Deep artificial neural networks applications in the biomedical sciences



Louis Falissard

St Catherine's College

University of Oxford

A thesis submitted for the degree of Master of Science by Research in Surgical Sciences

Under the supervision of Prof. Newton Howard and Ashok Handa

Trinity 2018

Deep artificial neural networks applications in the biomedical sciences

Louis Falissard

St Catherine's College

University of Oxford

A thesis submitted for the degree of Master of Science by Research in Surgical Sciences

Trinity 2018

Abstract

Deep artificial neural networks are a family of computational models that have led to a dramatical improvement of the state of the art in several domains such as image, voice or text analysis. These methods provide a framework to model complex, non-linear interactions in large datasets, and are naturally suited to the analysis of hierarchical data such as, for instance, longitudinal data with the use of recurrent neural networks.

In the other hand, the biomedical sciences are confronted with increasingly complex statistical analysis challenges that traditional methods oftentimes fail to handle properly. As deep neural networks were able to achieve so much in so little time in the field of artificial intelligence, it only feels natural to see whether these new analysis problems can be addressed with these methods. This thesis introduces the concept of deep artificial neural networks as applications in the biomedical sciences, with both a theoretical and practical approach.

List of abbreviations

MLP – Multi layer perceptron

ReLU – Rectified Linear Unit

LReLU – Leaky Rectified Linear Unit

RNN – Recurrent neural network

LSTM – Long Short-Term Memory

GRU – Gated Recurrent Unit

CNN – Convolutional neural network

Maxpool – Maximum pooling

Lasso – Least Absolute Shrinkage and Selection Operator

Adam – Adaptative Moment estimation

E3N – Etude Epidemiologique Aupres des Femmes de la MGEN

Kml – Longitudinal K-mean

CGI-S – Clinical Global Impression – Severity scale

FFT – Fast Fourier Transform

MFCC – Mel Frequency Cepstral Coefficients

List of figures

2.1 Margin classifier

2.2 Clustering algorithms

2.3 Semi supervised learning

2.4 Semi supervised learning\

2.5 Gradient descent

2.6 Non convex function

2.7 Multilayer perceptron

2.8 Multilayer perceptron

2.9 ReLU

2.10 Recurrent neural network

2.11 Regression on longitudinal data

2.12 Convolutional neuron

2.13 Convolution operation

2.14 Relationship between convolution operation and neuron

2.15 Padded convolution

2.16 Convolutional layer

2.17 Maxpool

2.18 Convolutional neural network

2.19 Visualization of convolutional

2.20 Bias/Variance tradeoff

2.21 Drop-out method

2.22 Drop-out method at inference

2.23 Initialization and gradient descent

2.24 Saddle points and gradient descent

2.25 Input normalization

2.26 Batch normalization

2.27 Grid search visualization

2.28 Random search visualization

2.29 Iterative random search visualization

2.30 Undercomplete autoencoder

2.31 Ladder encoder

2.32 Ladder autoencoder

2.33 Ladder combinator

2.34 Unsupervised learning with ladder networks

2.35 Semi supervised learning with ladder networks

2.36 Final inference with ladder networks

3.1 Recurrent autoencoder

3.2 E3N visualization

3.3 Qalitative comparison of E3N clusters

3.4 Quantitative comparison of E3N clusters

3.5 Simulated longitudinal dataset

3.6 Visualization of Kml results

3.7 Visualization of autoencoder results

4.1 Rectangular windowing

4.2 Hamming windowing

4.3 Autocorrelation operation

4.4 Fourier transform

4.5 Mel Filter Bank

4.6 Sequence-to-sequence models

4.7 Unimodal probability distribution

4.8 Binary search

4.9 Binary search tree

4.10 Recurrent neural networks and binary search for ordinal regression

4.11 Simulated ordinal dataset

4.12 Temporal evolution of CGI-S evaluations

4.13 Local evaluation model architecture

4.14 Temporal convolutional network\

5.1 Attention based model for speech recognition

List of tables

4.1 Inter rater agreement study results

4.2 Cohen kappa interpretation

Frequently used notations

Vectors are written in italic capital letters: X

Matrices are written in bold italic capital letters: $\boldsymbol{X}$

$\{X\}_{i,j}$ denotes the *i*th row of the *j*th column entry of matrix **X**

$(.)^T$ denotes the transpose of a matrix or vector

$\frac{dy}{dx}\ and\ \frac{\partial y}{\partial x}$ represents the differentiontion of a function y with respect to x

$*$ denotes the convolution operation

° denotes the Hadamard (element wise) product

$\|.\|$ represents the distance metric, which is typically chosen as the Euclidean distance, unless specified otherwise

Table of Contents

1 Introduction 1

2 Machine learning and deep artificial neural networks 2

2.1 Introduction to machine learning 2

2.1.1 Background 2

2.1.2 Supervised learning 3

2.1.3 Unsupervised learning 4

2.1.4 Semi-supervised learning 5

2.2 Neural networks 9

2.2.1 Linear regression 9

2.2.2 Logistic and softmax regressions 10

2.2.3 Optimization and Gradient descent 12

2.2.4 Multilayer perceptron 15

2.2.5 Backpropagation algorithm 20

2.2.6 Recurrent neural networks 24

2.2.7 Convolutional neural networks 26

2.3 Practical aspects of neural networks 39

2.3.1 Model evaluation 39

2.3.2 Regularization 44

2.3.3 Practical aspects of neural network optimization 48

2.3.4 Normalization 56

2.3.5 Hyperparameters tuning 60

2.4 Unsupervised learning and autoencoders 66

2.4.1 Autoencoder 66

2.4.2 Undercomplete autoencoder 68

2.4.3 Denoising autoencoder 69

2.5 Ladder networks, an example of semi supervised neural network 70

2.5.1 Ladder networks 72

2.5.2 Autoencoder architecture 73

2.5.3 Unsupervised and supervised cost functions and optimization 77

2.5.4 Model evaluation and final model inference 82

2.5.5 Evaluation 83
3 Deep clustering of longitudinal data in epidemiology 84
3.1 Background 84
3.2 Method 86
3.2.1 Overall 86
3.2.2 Recurrent autoencoder and deep longitudinal clustering 87
3.2.3 Embedding validation 88
3.2.4 Analysis 89
3.3 Results 91
3.3.1 Real-data application 91
3.3.2 Simulated data application 93
4 Voice analysis and automated detection of suicidal risks in crisis hotlines 96
4.1 Background 96
4.2 Methodology and data acquisition 98
4.2.1 Suicidal risk evaluation from crisis hotline recordings 98
4.3 Data preprocessing 104
4.3.1 Manual preprocessing with the Audacity software 105
4.3.2 Signal processing-based reduction of dimensionality 106
4.4 Analysis 119
4.4.1 Ordinal regression models 121
4.4.2 Local evaluation model 131
4.4.3 Contextual evaluation model 137
5 Conclusion and future work 140
6 Bibliography 147
7 Appendix 153
7.1 Longitudinal clustering experiment code 153
7.1.1 Model 153
7.1.2 Training 154
7.2 Ordinal regression experiment code 156
7.3 Voice analysis and suicide intent detection experiment code 159
7.3.1 Data preprocessing code 159

7.3.2 Data processing for local model code 160
7.3.3 Ladder model code 165
7.3.4 Temporal convolutional network for global modelling problem code 173
7.3.5 Losses code 175
7.3.6 Training for local modelling problem code 178
7.3.7 Training for the global modelling problem code 184

1 Introduction

The beginning of the 21st century witnessed the raise of the information age. As society became more and more digitalized, a growing amount of data coming was collected from heterogenous sources, oftentimes providing important new insights in various fields of study. Among other things, this wide availability of big datasets has been a major contributing factor in the raise of artificial intelligence and machine learning technologies (1), whose use has become hegemonic in most people's everyday life. More specifically, the democratization of deep artificial neural networks in machine learning based predictive modelling has had a significant impact in the analysis of complex, highly non-linear phenomenon, and has been at the basis of a wide range of innovations from self-driving cars to speech recognition technologies (2). Although biomedical sciences have been impacted by the "Big data" era, deep learning applications in this field have so far been surprisingly scarce. Indeed, apart from promising application in dermatology (3) or radiology (4), the biomedical science academic community remains quite unfamiliar with deep learning methods. Several reasons can be summoned to explain this apparent immiscibility. For instance, artificial neural networks methods are often regarded as too close to computer sciences for the medical scientist, who will prefer to use more classical statistical tools. In addition, deep learning models have a reputation for being "black box" machines (5), with close to no interpretability in their decision process, which is often incompatible with a scientific approach. However, artificial neural networks based models are actually closely related to statistical methods that are widely used in the medical sciences (6). As such, they can be used in a whole variety of statistical analysis problems typically found in fields such as epidemiology. In addition, their

ability to model complex phenomenon such as language, image, or voice analysis (2) make them promising candidates for the implementation of powerful decision aids tools, which might soon become necessary, as the amount of data for a given patient becomes too important for a medical practitioner to process alone, for instance. This thesis is meant as a bridging attempt between the fields of medical sciences and deep learning. First, on a theoretical level, most of the traditional deep artificial neural networks based machine learning algorithms are introduced and defined as statistical modelling techniques. Finally, on a practical level, the diversity of use that this powerful family of learning algorithm is illustrated with two practical applications on two different field of the medical sciences:

- Epidemiology, with the development of a clustering method for longitudinal data in cohort studies
- Psychiatry, with the implementation of a voice analysis based predictive models for the automated assessment of suicidal risk in crisis hotline

2 Machine learning and deep artificial neural networks

2.1 Introduction to machine learning

2.1.1 Background

Machine learning is a field of study focusing on giving computers the ability to perform predefined tasks without giving them explicit instructions on how to do so (7). Formally, in a well-defined machine learning problem, "A computer program is said to *learn* from experience E with respect to some task T and some performance measurement P, if its experience on T, as measured by P, improves with experience E" (8).

This broad definition leads to several possible mathematical interpretations, and, consequently, to an equal number of subfields of machine learning, usually overlapping with several other disciplines from computer sciences, artificial intelligence, mathematics, and specifically statistical analysis. Indeed, a profound relationship seems to link the concepts of experience as defined from a machine learning point of view, and of data analysis, which is ubiquitous in statistics.

The statistical analysis of real dataset being at the core of modern biomedical sciences, it only seems natural to focus on machine learning subfamilies that blend elements of statistical analysis. Some of the most advanced deep neural networks can for instance be thought of as advanced expansion of fairly classical tools used in statistical analysis in the biomedical sciences(6).

In such a scope, machine learning can be divided into three main categories, supervised, unsupervised and semi-supervised learning.

2.1.2 Supervised learning

Supervised learning is among the most common machine learning disciplines. It shares deep roots with statistical analysis. For instance, linear regression and linear discriminant analysis, two widely known statistical modelling techniques, can be considered as supervised machine learning techniques. The goal of supervised learning techniques is to, given a dataset, model the relationship between one or several independent variables (the input or endogenous variables) and a predefined dependent variable (9) (the output or exogenous variables).

Two main distinct types of supervised machine learning can be identified, namely given the output's variable nature:

- Classification for categorical output variables: The learning algorithm is asked to find a surface function of the input variables that best separates the output variable's different states
- Regression for quantitative variables: The learning algorithm is asked to find a function of the input variables that best fits the output variables

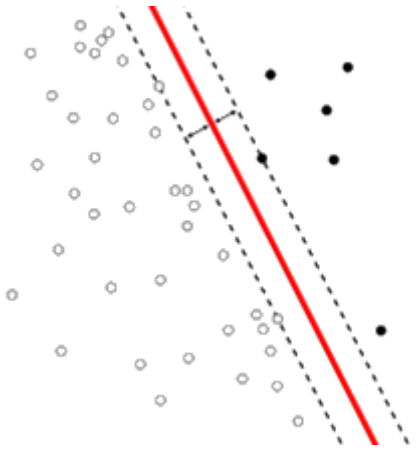


Fig. 2.1 Example of supervised learning method: Margin classifier (10). The learning algorithm determines the straight line that separates the black and white dots that maximize the distance from the lines to the data points

These methods can easily be linked to the previously seen definition of machine learning:

- Experience E: The gathered dataset
- Task T: Correctly estimate the output variables given the input variables
- Performance P: algorithm's accuracy on the data

2.1.3 Unsupervised learning

Supervised learning makes a difference between variables in a given dataset. Namely, supervised algorithms are trained to, given the input variables, explain or predict the output variables. No such difference is made between variables in unsupervised learning. The dataset

(or a subset of variables selected from it) is instead taken as a whole and searched by the learning algorithm for potentially interesting structures or properties. Clustering algorithms, for instance, are designed to identify within the dataset groups of samples who share common traits. A complex problem rising from the nature of unsupervised learning algorithms is the difficulty encountered when trying to define the concept of quality or relevance, of a structure extracted from a dataset by a given algorithm. As a consequence, most unsupervised learning algorithms rely on strong assumptions on the investigated dataset, and their performances are usually assessed through the use of heuristics. As such, their performance cannot, as of today, be assessed fully objectively (11).

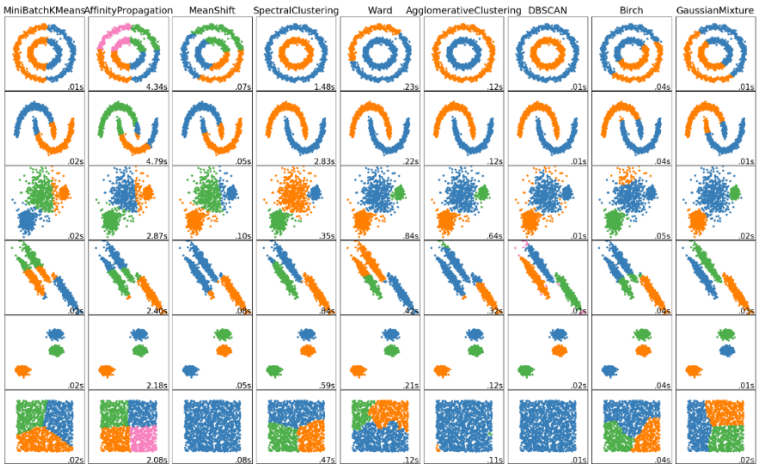


Fig 2.2 Example of clustering algorithms applied on several toy problems (12) Each algorithm groups data points (here shown in different colors) according to a predefined heuristic, which can become quite hurtful to the clustering quality if ill-chose

2.1.4 Semi-supervised learning

More than a completely different approach to machine learning, semi-supervised machine learning is rather a combination of supervised and unsupervised learning. Semi-supervised approached are often used in supervised situations where the number of labelled example (samples with both input and output variables available) is low, but the number of unlabeled

examples (samples with only input variables available) is high. In many situations, gathering a dataset of input variables is fairly straightforward, but the collection of their associated labels requires the input of a human expert, for instance a diagnosis decision taken by a medical practitioner.

The main idea behind semi-supervised learning is to boost the performance of a supervised learning algorithm by using the dataset's inherent structure, inferred from an unsupervised learning approach. For instance, a classification problem might be helped by a preliminary procedure, as subjects in the same groups might also show similar output variables (13).

For a better understanding of the concept underlying semi-supervised learning, a toy example of a semi-supervised binary classification problem, as shown in figure 2.3, can be considered. This example shows a very weakly supervised problem, where only one data point per class is paired with its corresponding label (the black and white points). The quantity of unlabeled points (in grey) is however significantly larger.

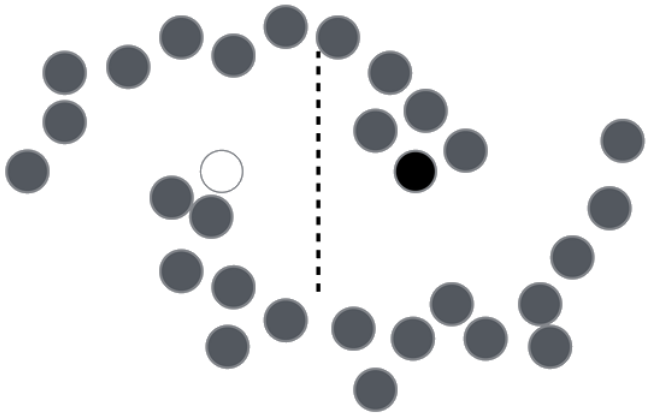


Fig 2.3 Simple example of a semi supervised problem (14). Only one labelled example of each class is available, but many unlabeled examples are available

The surface decision extracted by the supervised algorithm poorly classifies the actual dataset, which is not surprising considering the lack of information available regarding the output variable. The dataset, however, presents a strong structure in the form of two perfectly separated, non-spherical clusters.

One possible approach to make use of the unlabeled data points would be to alternate to procedures:

- Identify the labeled points nearest neighbors, and labelled them identically
- Estimate a new decision boundary with the selected learning algorithm, this time with the new artificially labelled examples
- Stop when no unlabeled neighbor is present anymore

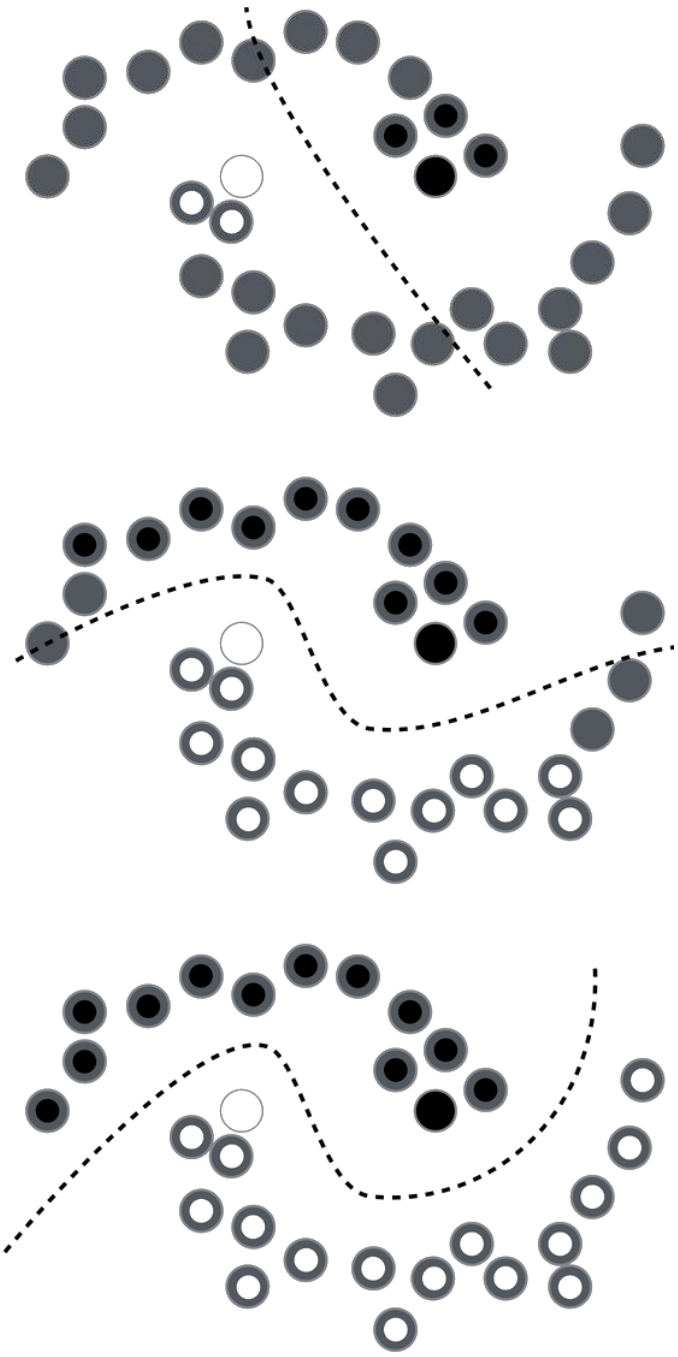


Fig. 2.4 Application of semi supervised learning to the previous example at different iteration steps (14). As the two-steps procedure proceeds, the boundary decision gets closer to what could be expected as the true boundary

As shown in figure 2.4, as the dual procedure iterates, the decision boundary's complexity raises, as well as its consistency with the dataset. Conditional to the two clusters actually

sharing their output variables' state, the supervised learning algorithm's performance has been significantly improved, compared to what can be achievable in a solely supervised setting.

2.2 Neural networks

Artificial neural networks are a family of machine learning algorithms loosely inspired by biological neural networks. They are nowadays crucial components in most artificial intelligence related technologies such as speech or image recognition, self-driving cars or machine translation (2). Their current popularity comes from their unprecedented ability to learn highly complex and nonlinear concepts by composing together a multitude of much simpler functions (6).

Artificial neural networks are quite versatile and can be adapted to a number of learning paradigm. Indeed, although they are mostly used for supervised learning, where they share a number of common characteristics with well-known statistical methods such as linear or logistic regression they can be adapted to perform in unsupervised or unsupervised settings.

2.2.1 Linear regression

Linear regression is a widely used modelling technique consisting of modelling dataset with a linear, multivariate approximation of the real, studied phenomenon (15). This simple approach also shares similarities with regressive feedforward neural networks and make as such a good introduction to understand them.

Let $\{(X_i, y_i)\}_{0<i<N+1}$ be a set of n observations (typically a multivariate dataset collected for an experimental study), with X_i and y_i being the i^{th} sample's input and output variables, respectively.

The objective of linear regression modelling is to find $\hat{y}$, a linear function of the input variable parameterized in (W, b) that best predicts the output variables, the concept of best predictive approximation being formalized by a cost function L, that quantifies how close to the observed dataset the approximation is (the least square function in the case of linear regression).

$$\hat{y}_{W,b}(X) = (W)^T \cdot X + b$$

$$L(W, b) = \frac{1}{2N} \cdot \sum_{i=1}^{N} \left(\hat{y}_{W,b}(X_i) - y_i\right)^2) = \frac{1}{2N} \cdot \sum_{i=1}^{N} ((W)^T \cdot X_i + b - y_i)^2)$$

The regression model is extracted by the data by finding $\hat{y}_e$ such that:

$$\hat{y}_e(X) = (W_e)^T \cdot X + b_e$$

$$with \quad (W_e, b_e) = \underset{(W,b)}{\operatorname{argmin}}(L(W, b))$$

In the case of linear regression, the parametric solution (W_e, b_e) can be determined either analytically, or through the use of an optimization algorithm such as gradient descent. Finding the parameters that minimize a specific cost function is common to a number of machine learning algorithms, for which it is called the training process. Linear models are known to be easy to fit, as well as their good generalization behavior. They however fail to capture any non-linearity, as well as variable interactions.

2.2.2 Logistic and softmax regressions

Logistic and softmax regression models are regressive models dedicated to the analysis of a binary, categorical output variables (two states variables such as 0/1, ill/healthy, pass/fail) and multimodal categorical variables (number of states >2), respectively (16). They are widely

used in statistical analysis as part of the family of generalized linear models and can be used from a machine learning perspective to perform binary linear classification.

The process of extracting a logistic regression model from a dataset is fairly similar to what is done in linear regression models. The objective of a linear regression model is to find a parametric function linking the input and output variables by minimizing a cost function of the model's parameters. However, the output variable's categorical nature in logistic regression requires a subtle change in the problem's formalization. Instead of trying to directly link the input and output variables, the objective is to link the input variables to the probability of the output variable being in one state or another, implicitly obtaining the output variable by taking the most probable option.

Let $\{(X_i, y_i)\}_{0<i<N+1}$ be a set of n observations (typically a multivariate dataset collected for an experimental study), with X_i and y_i being the i^{th} sample's input and categorical output variables, respectively.

The objective of logistic regression is to find a parametric function of the input variables that outputs an estimation function h of the output variable's probability to be in one state or the other, such that the output variable is best predicted, the concept being quantified through the formalization of a cost function L called cross entropy.

$$h_{W,b}(X) = \frac{1}{1 + e^{-((W)^T \cdot X + b)}}$$

$$\hat{y}_{W,b}(X) = \begin{cases} 1 & if \;\; h_{W,b}(X) > 0.5 \\ 0 & if \;\; h_{W,b}(X) < 0.5 \end{cases}$$

$$L(W,b) = -\frac{1}{N}\sum_{i=1}^{N}[\, y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\,]$$

The regression model is extracted by the data by finding h_e, and consequently $\hat{y}_e$, such that:

$$h_e(X) = \frac{1}{1 + e^{-((W_e)^T \cdot X + b_e)}}$$

$$with \quad (W_e\,, b_e) = \underset{(W,b)}{\operatorname{argmin}}(\, L(W, b))$$

The model's parameters $(W_e\,, b_e)$ can be determined through the use of an optimization algorithm such as gradient descent.

The process of fitting a softmax regression to a dataset with a M-states multimodal categorical variable is essentially the same as for a logistic regression, with a slight adaptation of the parametric function to account for the different variable states:

$$\forall\, j \in [\![1;\, M]\!], \qquad h_{W_j,b_j}(X) = \frac{e^{-((W_j)^T \cdot X + b_j)}}{\sum_{i=1}^{K} e^{-((W_i)^T \cdot X + b_i)}}$$

$$\hat{y}_{W,b}(X) = \underset{j \in [\![1;\, M]\!]}{\operatorname{argmax}}(\, h_{W_j,b_j}(X))$$

Similar to linear regression, logistic and softmax regressions are considered linear classification algorithms and cannot take account of potential non-linearity and variable interactions in the dataset.

2.2.3 Optimization and Gradient descent

In both linear and logistic regression, model fitting requires to find the minimum of a function of the model's parameters, called the cost function. The problem of finding the minimum of the least square cost function in a linear regression model is relatively straightforward, as it can be explicated analytically. This is not necessarily the case when using logistic regression. As a consequence, the use of optimization algorithms is usually necessary to obtain an approximate, computational solution.

A number of optimization algorithm are available to the practitioner, who chooses from them according to the problem at hand in a case by case manner. Gradient descent is a well-known optimization algorithm that is consistently used in machine learning to minimize cost functions.

Gradient descent is an iterative optimization algorithm that explores functions in a search for a local minimum by making use of its first-order derivatives in the form of the function's gradient. The gradient of a function f from $\mathbb{R}^n$ to $\mathbb{R}$ is defined such that $\forall(x_1, -, x_n) \in \mathbb{R}^n$:

$$\nabla(f(x_1, -, x_n)) = (\frac{\partial f}{\partial x_1}, -, \frac{\partial f}{\partial x_n})$$

The general idea behind gradient descent is to first evaluate the function and its gradient at a random location, identify the direction in which the function is "going down the most", and take a step in that direction. After a number of steps, the gradient will evaluate close to 0, meaning that the location in which the function is being evaluated is "flat", meaning that there is no immediate local neighborhood to explore providing lower values of the investigated function. As a consequence, the final position is a local minimum. More formally, for a function f from $\mathbb{R}^n$ to $\mathbb{R}$, $a_0 \in \mathbb{R}^n$ a random vector and two real number γ and S, a gradient descent algorithm is based on iterating:

$$\forall n \in \mathbb{N}, \quad a_{n+1} = a_n - \gamma \nabla f(a_n) \quad as\ long\ as\ \nabla f(a_n) > S$$

With sufficiently low values for S, (a_n) progressively converges toward a function's local minimum. During the practical implementation of a machine learning algorithm, the step size (or learning rate in machine learning terminology) γ needs to be fine-tuned by the machine learning practitioner. Its value has a significant impact on the optimization process, as it sets

the size of the steps taken by the algorithm at each descent iteration. As a consequence, it needs to be chosen carefully. Indeed:

- Too big a learning rate value will lead to an unstable gradient descent. The steps taken at each iteration are too large to make use of the local information given by the gradient computation. A gradient descent with too big a learning rate might not converge, and can sometime, especially with deep neural network, diverge
- Too small of a learning rate will lead to an exceedingly slow optimization process. Each step will be taken in the right direction, but the new point will be located right next to the previous one, thus making close to no progress in finding a minimum.

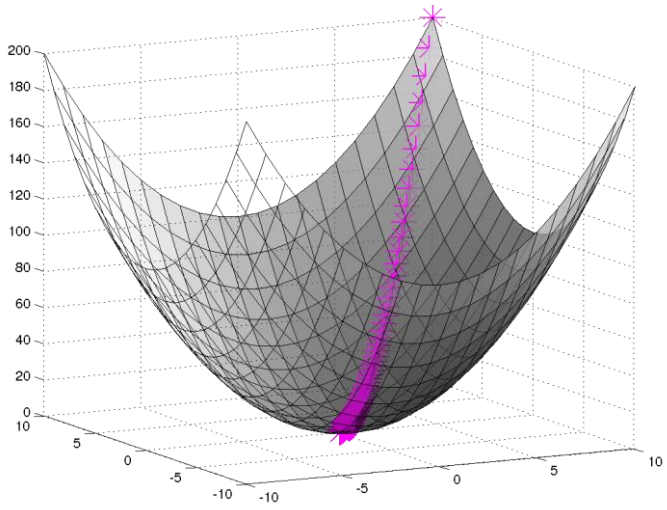


Fig 2.5 Example of gradient descent on a convex function. The gradient descent's steps (in purple) begins at a random point on the function's surface. Each step iteratively brings it closer to the function's minimum, where the gradient become null.

Finding a function's minimum using gradient descent requires certain hypothesis from the investigated function. Mainly:

- The investigated function needs to be defined and differentiable to perform gradient descent (these hypotheses can be weakened, but neural network all present those properties)
- The investigated function needs to be convex for the algorithm to find the function's global optimum. In other cases, gradient descent cannot provide any warranty regarding the identified local optimum's quality

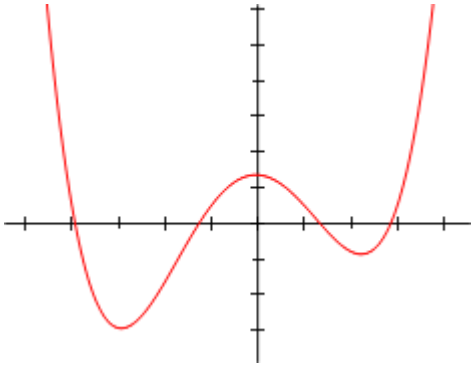


Fig 2.6 Example of non-convex function presenting two local minima. The left-hand minimum is the global optimum. However, a gradient descent algorithm initialized with a positive value will fall into the right-hand minimum, which is suboptimal

2.2.4 Multilayer perceptron

Multilayer perceptron are among the most widespread deep learning models (6). They can be used in a variety of modelling tasks, such as regression or classification, where then can be seen as nonlinear expansions to respectfully traditional mean square linear regression and logistic regression.

The idea behind feedforward neural networks is to fit a linear model to a transformed set of observations $\{(\phi(X_i), y_i)\}_{0<i<N+1}$ where not only the model's parameters, but also ϕ, are learnt from the data.

The traditional approach to enable a neural network model to learn non-linear interaction from the data is to inject linear combination of the investigated parameters into simple, nonlinear functions, whose outputs are then either used to perform a linear modelling task, or as inputs to be injected in another set of nonlinearities.

The toy network presented in figure 1 is an example of a feedforward neural network with one hidden layer, used in a regression setting. As in linear regression, the objective is to fit a set of N observations $\{(X_i, y_i)\}_{0<i<N+1}$ with a parametric, non-linear approximation:

$$\hat{y} = w_{11}^3 \cdot f_1(w_{11}^2 \cdot x_1 + w_{12}^2 \cdot x_2 + b_1^2) + w_{12}^3 \cdot f_2(w_{21}^2 \cdot x_1 + w_{22}^2 \cdot x_2 + b_2^2) + b_1^3$$

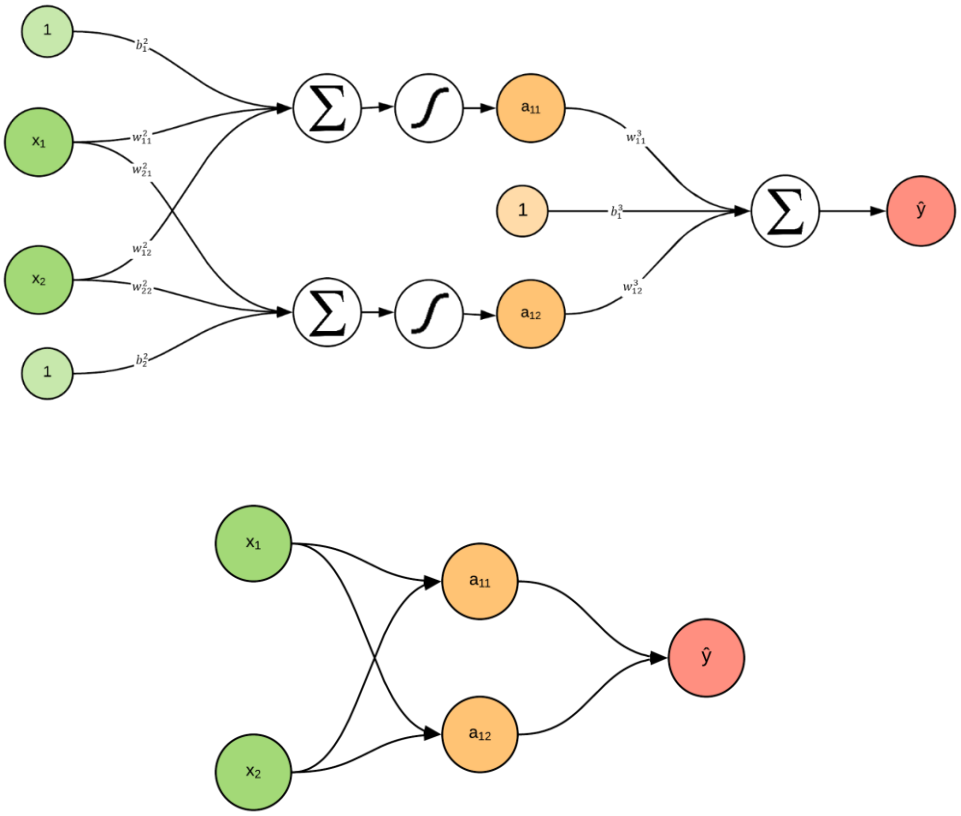


Fig. 2.7: Top: simple example of a feedforward neural network in regressive settings. Bottom: The same network in its traditional, simplified

representation. The biases are omitted and the linear combinations and injections in nonlinearities are done implicitly for ease of notation

While the parameters are typically obtained from the minimization of the square loss function, two differences arise from the introduction of parametric non linearities in the model:

- The parametric solution that minimizes the cost function cannot be expressed in closed form, and must be estimated through gradient based optimization
- The cost function's gradient is not straightforward to compute
- The resulting optimization problem loses its convexity, and the warranty of finding the global minimum

The entire family of feedforward neural network can then be derived from this toy example, mainly through the variation of the cost function and three hyperparameters:

- Cross entropy or least square cost function for classification and regression respectively
- Number of composed function in chain (number of layer)
- Number of newly build feature per layer (number of neurons)
- Type of nonlinearity applied to the neurons output (i.e. choice of functions f^1 and f^2)

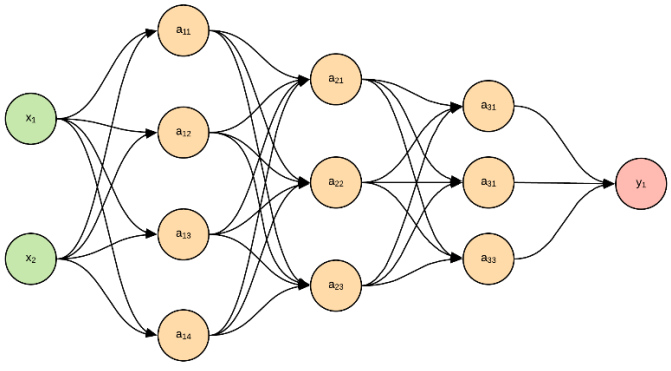


Fig. 2.8. Neural network model with three hidden layers (in orange). The green and red nodes are respectively the model's input (endogenous variables) and output (exogenous variables

Increasingly complex models can be built by tuning these hyperparameters, thus designing a model most fitted to the problem's complexity.

Theoretically, any type of nonlinearity f can be applied to the neurons' outputs. In real life applications, however, practitioners usually choose between a limited number of options that have been empirically proven to perform well, with the most notable example being the linear rectified unit (17) defined for $x \in \mathbb{R}$ as:

$$ReLU(x) = \begin{cases} x & if\ x > 0 \\ 0 & otherwise \end{cases}$$

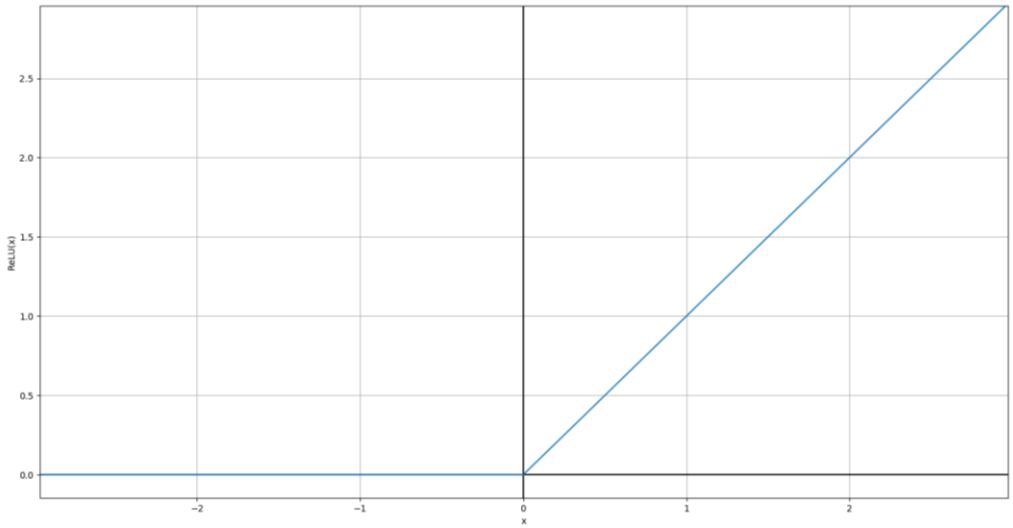


Fig 2.9 Plot of the ReLU and LReLU non linearities. Both functions are identical for positive real numbers

While the linear rectified unit is currently one of the most widely used nonlinearity in the deep learning academic literature, a lot of work has been done toward trying to improve on

its design, and has given birth to a family of linear rectifier type units, as for instance the leaky rectified linear unit (LReLU) (18) defined for $x \in \mathbb{R}$ as:

$$LReLU(x) = \begin{cases} x & if\ x > 0 \\ 0.1\ x & otherwise \end{cases}$$

The LReLU only differs from the traditional ReLU for a negative x, where it is given a strictly negative slope. The rationale behind this subtle change is related to the optimization process and more specifically the network's gradient shape. Indeed, a traditional ReLU displays a null derivative for any negative x, which can potentially hurt the gradient descent process. The LReLU prevents this problem by giving the nonlinearity a relatively small (but nonzero nonetheless) slope on the $\mathbb{R}^-$ domain. The subsequently obtained nonlinearity displays a closely similar behavior as the ReLU, but offers a nonzero derivative, potentially improving the gradient based optimization procedure used during the learning process (18).

In addition to the nature of the output variable, the only difference between a deep neural network in regressive or classification setting is its associated cost function, which does not in itself have any impact on a neural network's architecture. As a consequence, and for simplicity of writing, every neural network further described will be so in regressive settings. Its classifier counterpart can systematically be derived by changing the least square objective function by the cross-entropy cost function.

2.2.5 Backpropagation algorithm

In a similar fashion to statistical parametric models, the extraction of a neural network-based model from an investigated dataset is done through the computation of the solution minimizing a predefined cost function across the entire family of investigated parametric model (such as for instance the set of all straight line in univariate linear regression models). Although simple models such as linear regressive models allow for a closed form expression of these optimal solutions, they are significantly more complex to derive when it comes to artificial neural networks such as the multilayer perceptron. A number of approaches have been devised over the years to automatically tune the model's parameter to best fit the observed data, typically through gradient based optimization methods.

As mentioned before, finding a given function's minimum through a descent-based optimization algorithm relies on an iterative process involving the computation of the function's gradient at each step. The strongly composed nature of the multi-layer perceptron's parametric form, especially when given a large number of layers, can make the computation of its gradient's manual implementation cumbersome. In addition, the network's architecture, which can significantly vary during the practical implementation of a neural network, has a direct influence on the gradient's shape. As a consequence, manual implementation of the gradient's computation is time-consuming and prone to errors. The backpropagation algorithm (19) was developed as a general method allowing for efficient cost function's gradient computation in any kind of feedforward neural network such as the multilayer perceptron.

Whether for classification or regression analysis, a machine learning's objective function is built to quantify the difference between the investigated dataset's output variable and its

proposed approximation obtained through the model, for every available subject independently. Coming back to the linear regression analysis for $\{(X_i, y_i)\}_{0<i<N+1}$ a set of N observations, the cost function L was defined such that:

$$L(W,b) = \frac{1}{2N} \cdot \sum_{i=1}^{N} \left(\hat{y}_{W,b}(X_i) - y_i\right)^2 = \frac{1}{2N} \cdot \sum_{i=1}^{N} ((W)^T \cdot X_i + b - y_i)^2$$

Conceptually, this cost function defines the idea of distance between the observed data samples and the model by observing the difference between every single sampled example and their associated predictions, to then average over the whole dataset. As a consequence, both the cross entropy and least square cost function are written as a sum over a sample-wise error function applied to every example present in the dataset (and potentially a multiplicative factor):

$$L(W,b) = \alpha \sum_{i=1}^{N} L_{ind}(W,b,X_i,y_i)$$

The cost function's gradient can thus be written, because of the gradient's linear properties:

$$\nabla(L(W,b)) = \nabla\left(\alpha \sum_{i=1}^{N} L_{ind}(W,b,X_i,y_i)\right)$$

$$= \alpha \sum_{i=1}^{N} \nabla(L_{ind}(W,b,X_i,y_i))$$

As a consequence, computing the entire cost function's gradient can be considered as equivalent to computing the individual error function's gradient for every candidate present in the dataset. This interesting property shared by many of the traditionally used cost functions is used in the backpropagation algorithm to provide an efficient procedure to

compute their gradients by evaluating gradient contributions from every sample and combining them into the global gradient. The interest of this approach is dual:

- The sub gradients can be evaluated in a fairly straightforward approach using derivative chain rules
- As the computation of gradient contribution by every sample is completely independent, they allow for a powerful parallel implementation that significantly reduce the required computation time

For any observation (X_i, y_i) from the investigated dataset, given a multi-layer perceptron with n hidden layers of $\left(m^{[1]}, -, m^{[n]}\right)$ neurons respectively, parameterized with weights and biases real-valued vectors $\left(W^{[1]}, -, W^{[n]}, b^{[1]}, -, b^{[n]}\right)$ and non-linearities $(g^{[1]}, -, g^{[n]})$ and a cost function L adapted to the investigated modelling problem, the computation of the gradient of L with respect to the weight and biases $\left(\frac{\partial L}{\partial W^{[1]}}, -, \frac{\partial L}{\partial W^{[n]}}, \frac{\partial L}{\partial b^{[1]}}, -, \frac{\partial L}{\partial b^{[n]}}\right)$ can be obtained with the following procedure:

- Use the current parameter values to compute the model's approximation of the current data sample's output variable (forward propagation), while keeping track of every layer's pre-activation vector (the neurons' outputs before injection in a nonlinearity) $(Z^{[1]}, -, Z^{[n]})$
- Use the approximated values to compute the cost function's value
- Directly compute $\frac{\partial L}{\partial A^{[n]}}$, the cost function's derivatives with respect to the network's last layers activations.

- Starting with the n[th] layer and proceeding in a backward fashion, compute the cost function's derivatives with respect to the l[th] layer's parameter using $Z^{[l]}$, $\frac{\partial L}{\partial A^{[l]}}$ and the following set of equations (19):

$$\frac{\partial L}{\partial Z^{[l]}} = \frac{\partial L}{\partial A^{[l]}} \odot g^{[l]\prime}(Z^{[l]})$$

$$\frac{\partial L}{\partial W^{[l]}} = \frac{1}{m^{[l]}} \frac{\partial L}{\partial Z^{[l]}} \cdot (A^{[l-1]})^T$$

$$\frac{\partial L}{\partial b^{[l]}} = \frac{1}{m^{[l]}} \sum_{i=1}^{m^{[l]}} (\frac{\partial L}{\partial Z^{[l]}})_i$$

$$\frac{\partial L}{\partial A^{[l-1]}} = {W^{[l]}}^T \cdot \frac{\partial L}{\partial Z^{[l]}}$$

An interesting property of this procedure, that gives it its name, lies in the fact that at each gradient descent iteration, the neural network currently trained is used twice:

- Explanatory variables are injected in the network's inputs and are propagated through the entire network to obtain output estimations. This step is called forward propagation
- The cost function's gradient is injected at the end of the network's and is propagated in backward manner through the network to obtain derivative with respect to every parameter. This step is called backward propagation.

In present times, most deep learning programming libraries incorporate automated procedures to compute a neural network's gradient using the backpropagation procedure and allow for parallel implementations given appropriate hardware (typically high-end Graphics Processing Units).

2.2.6 Recurrent neural networks

Feedforward neural networks can in theory be used to analyze sequential observations of a variable $(x^{(1)}, x^{(2)}, \dots, x^{(n)})$, for instance by feeding every timesteps directly in a neural network. However, this approach can turn out to be rather cumbersome. Indeed, trying to fit a densely connected feedforward network to a sequence of data results in an exponential increase in the number of parameter in the model with the increase of the data sequence's length. In addition, feedforward neural networks are clearly not able to model sequences of different lengths, which prevents their use in a number of longitudinal data analysis problem.

Recurrent neural networks are a family of neural network that specialize in the analysis of longitudinal data (20). The main idea behind the elaboration of a recurrent neural network is to devise a model that shares its parameter across all timestep within the data. Instead of feeding the whole sequence to a neural network, each timestep in the data is sequentially fed to the neural network, which is given a retroactive loop to enable him to account for past iterations.

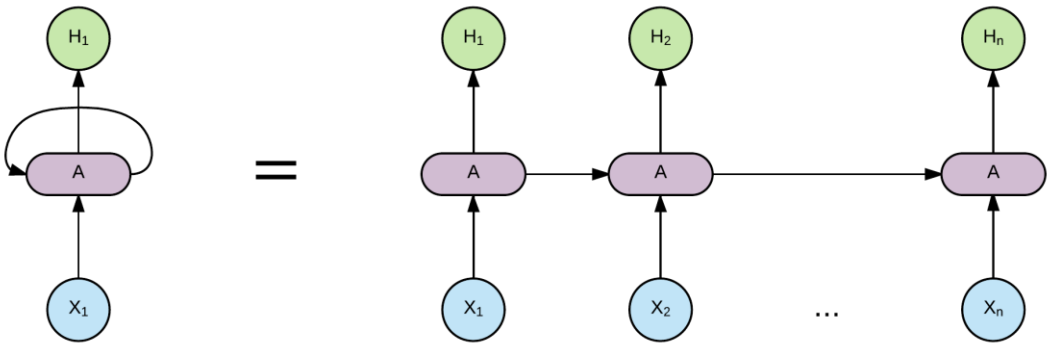


Fig. 2.10 Recurrent neural network architecture, with on the right its looped form and on the left its unrolled representation

This family of neural network can be used in a variety of settings, such as, for instance, non-linear regression on sequential data. The objective is to fit N observations $\{(\boldsymbol{X}_i, y_i)\}_{0<i<N+1}$ with a parametric approximation such that:

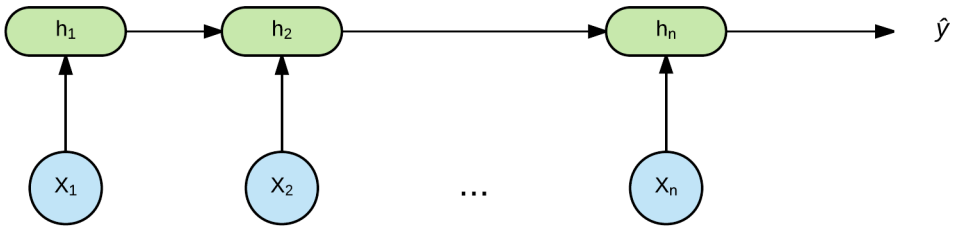


Fig. 2.11 Recurrent neural network for regression on sequential data

$$\forall i \in \mathbb{N}, i \in [1, n]$$

$$h_i = f(x_i, h_{i-1}) \qquad with\ f\ a\ parametric\ family\ of\ functions$$

$$\hat{y} = \mathrm{W}\,.\,h_n$$

Similar to a feedforward neural network, $\hat{y}$ is finally approximated from the recurrent neural networks last output layer through the fitting of a traditional linear regression, whose parameter are obtained by the optimization of a least square cost function on both W and the parameter of the parametric function f, which was historically chosen to be a feedforward neural network.

However, the use of a feedforward neural networks for the f function leads to several issues in the model fitting process that essentially render the optimization problem intractable (21).

Several architectures have been devised to counter these problems, such as the Gated Recurrent Unit (22), or The Long-Short-Term-Memory (LSTM) (23) cell, the latter being one of the most wildly used.

The Long-Short-Term-Memory cell (23) is a variation of a traditional recurrent network, and defines the parametric function f defined for the proposed approach to regression on sequential data as:

$$f_t = \sigma_g(W_f \cdot x_t + U_f \cdot h_{t-1} + b_f)$$

$$i_t = \sigma_g(W_i \cdot x_t + U_i \cdot h_{t-1} + b_i)$$

$$o_t = \sigma_g(W_o \cdot x_t + U_o \cdot h_{t-1} + b_o)$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \sigma_g(W_f \cdot x_t + U_f \cdot h_{t-1} + b_f)$$

$$h_t = o_t \circ \sigma_h(c_t)$$

With :

- x_t as the input vector
- h_t as the output vector
- c_t as the cell state vector
- $W, U, and\ b$ the cell's parameter matrices and vector
- σ_g the logistic function
- $\sigma_c\ and\ \sigma_h$ the hyperbolic tangent function
- $\cdot$ the dot product
- $\circ$ the Hadamard product

2.2.7 Convolutional neural networks

Recurrent neural networks allow for the modelling of sequential datasets by sharing their neurons' parameters across all timesteps and the addition of a "memory" to the network. The idea of several neurons sharing the same parameters is a powerful concept that can be

adapted to a number of model architectures, in order to extract powerful predictive models from structured datasets. Convolutional neural networks offer a different approach to parameter sharing that was developed for computer vision task and the analysis of images as real valued 2-dimensional grids (24). Indeed, multilayer perceptron-based models could be fit to such a data type, but similar to sequence datasets, the high number of variables contained in an image make for a combinatoric explosion which usually prevent their use.

To prevent this problem, convolution based neural networks focus on the modelling of local information, by limiting the neurons' access to the input variables to small sub-grids (typically 2*2 to 5*5 areas) of the entire matrix. Each neuron is then copied across the entire image, to output not only one scalar activation, but a 2-dimensional array called an activation map. Similarly to multilayer perceptrons, every neuron's activation map can then be injected into additional, similar neural layers.

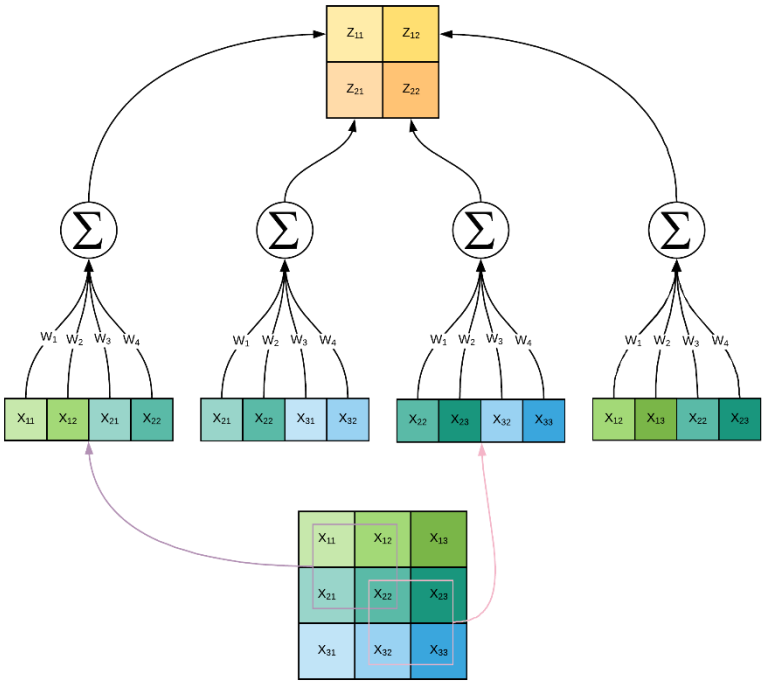


Fig. 2.12 Example of a convolution neuron applied to a 3x3 matrix. The neuron has access to limited 2x2 sub grids but sweeps the entire input grid.

As a consequence, the neuron outputs several real values, instead of a single one

Although copying a neuron with limited connections to its input data presents a powerful way to make use of the idea of parameter sharing for the analysis of grid-like objects, its practical implementation as is can quickly become quite cumbersome. Indeed, naively copying the same neuron across the entire input data structure makes for a substantial amount of memory used to store redundant information. However, the transformation linking the input grid and the resulting activation map depending on one neuron's parameter is actually equivalent to a well-known signal processing tool, called cross-correlation. Through a slight modification of the traditional representation of an artificial neuron, this transformation can be used to implement a computationally efficient version of the aforedescribed neural architecture. In the deep learning academic literature, the cross-correlation operation is wrongly called a convolution operation. Although different transformations, the two share similarities and symmetries that make them yield similar results when used in neural network. As a consequence, the neural networks making use of this sub grid approach are called convolutional neural networks, in spite of using a cross-correlation operation. In order to prioritize consistency with the deep artificial neural network literature, the following formal description of convolutional neural network will make use of this designation, and the convolution operation defined further will be equivalent to the signal processing's cross correlation.

Even though convolutional neural networks were specifically designed for the analysis of image-based datasets as 2-dimensional grids, it can easily be adapted for use on sequential datasets (25), by considering a sequence of a variable as a grid of length the sequence's length

and unitary width. Its applications in fields such as text or vocal analysis-based classification is actually a growing area of research (26).

2.2.7.1 Convolution operation

The convolution operation is a function takes two grid objects as entries, with one usually smaller than the other and outputs another grid that gathers the results of the sum of the element wise products of the smaller grid and every similar sized sub-grid from the bigger grid. Formally, for $\boldsymbol{A} \in \mathbb{R}^{n\times n}, \boldsymbol{B} \in \mathbb{R}^{m\times m}$ with $(n, m) \in \mathbb{N}^2$ such that $n > m$, the convolution of $\boldsymbol{A}$ and $\boldsymbol{B}$ noted $\boldsymbol{A} * \boldsymbol{B}$ is a matrix of size $(n - m + 1) \times (n - m + 1)$ defined as:

$$\forall i, j \in [\![1, n - m + 1]\!], \ \ (\boldsymbol{A} * \boldsymbol{B})_{i,j} = \sum_{k=1}^{m}\sum_{l=1}^{m}(\boldsymbol{A})_{i+k-1,j+l-1}(\boldsymbol{B})_{k,l}$$

To better understand the properties of this operation, as well as its relationship with the neural architecture of the convolutional neural network, the same example used previously can be considered. Let X and W be 3 by 3 and 2 by 2 real valued matrices, respectively. The convolution of X and W can then be written as a 2 by 2 matrix such that:

$$\forall i, j \in [\![1, 2]\!], \qquad (X * W)_{i,j} = \sum_{k=1}^{2}\sum_{l=1}^{2}(X)_{i+k-1,j+l-1}(W)_{k,l}$$

$$(X * W)_{i,j} = \ (X)_{i,j}(W)_{1,1} + (X)_{i,j+1}(W)_{1,2} + (X)_{i+1,j}(W)_{2,1} + (X)_{i+1,j+1}(W)_{2,2}$$

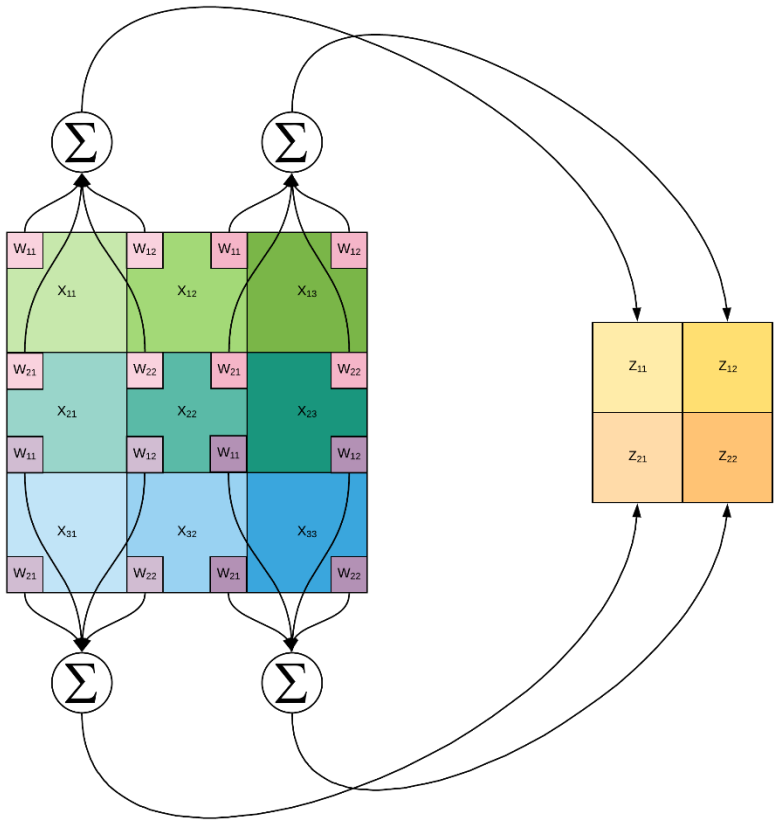


Fig 2.13 Visualization of the convolution of 3 by 3 and 2 by 2 matrices $\boldsymbol{X}$ and $\boldsymbol{W}$, and its similarity to the convolution neuron architecture

On another hand, when considering a 3 by 3 grid matrix input variable $\boldsymbol{X}$ and a neuron given limited access to 2 by 2 sub grids from it (and as such having a 4-dimensional weight vector), the neuron's pre-activation map can be written as a 2 by 2 matrix such that:

$$\forall i,j \in [\![1,2]\!], \qquad (\boldsymbol{X} * \boldsymbol{W})_{i,j}$$
$$= (\boldsymbol{X})_{i,j}(\boldsymbol{W})_1 + (\boldsymbol{X})_{i,j+1}(\boldsymbol{W})_2 + (\boldsymbol{X})_{i+1,j}(W)_3 + (\boldsymbol{X})_{i+1,j+1}(\boldsymbol{W})_4$$

By comparing the two previous equations, the neuron's pre-activation map can easily be identified as being the convolution of the input grid by a grid composed of the neuron's weight parameters, and as a consequence can be computed as such.

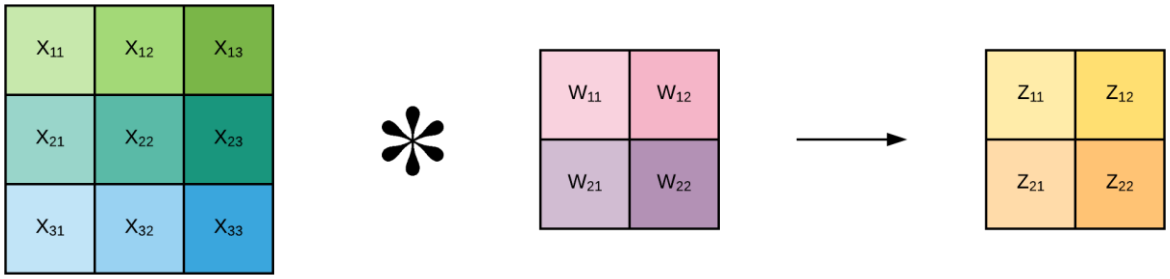


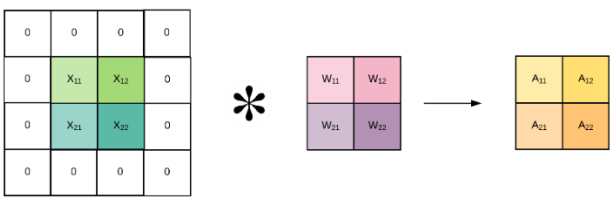
Fig 2.14 Same neuron as in figure 2.13, written as a convolution product

When applied in artificial neural networks, the convolution operation is usually extended in two distinct ways in order to provide the practitioner with more flexibility:

- Stride variations allow for the reduction of the output's dimensionality, by allowing more space between each copied neuron. As a consequence, some of the input matrixes sub grids are not observed by neurons. Formally, for $\boldsymbol{A} \in \mathbb{R}^{n\times n}, \boldsymbol{B} \in \mathbb{R}^{m\times m}$ with $(n,m) \in \mathbb{N}^2$ such that $n > m$, the convolution of A and B with stride s noted $\boldsymbol{A} * \boldsymbol{B}(s)$ is a matrix of size $\left\lfloor \frac{n-m+1}{s} \right\rfloor \times \left\lfloor \frac{n-m+1}{s} \right\rfloor$ defined as (27):

$$\forall i,j \in \left[\!\left[1, \left\lfloor \frac{n-m+1}{s} \right\rfloor \right]\!\right], \quad (\boldsymbol{A} * \boldsymbol{B}(s))_{i,j} = \sum_{k=1}^{m}\sum_{l=1}^{m} (\boldsymbol{A})_{i*s+k-1,j*s+l-1} (\boldsymbol{B})_{k,l}$$

- 0 padding can be added to the input matrix, in order to artificially raise its dimension, thus raising the convolution's output dimensionality. In practice, padding is typically used to set the convolution's output to be of same size as the input matrix.



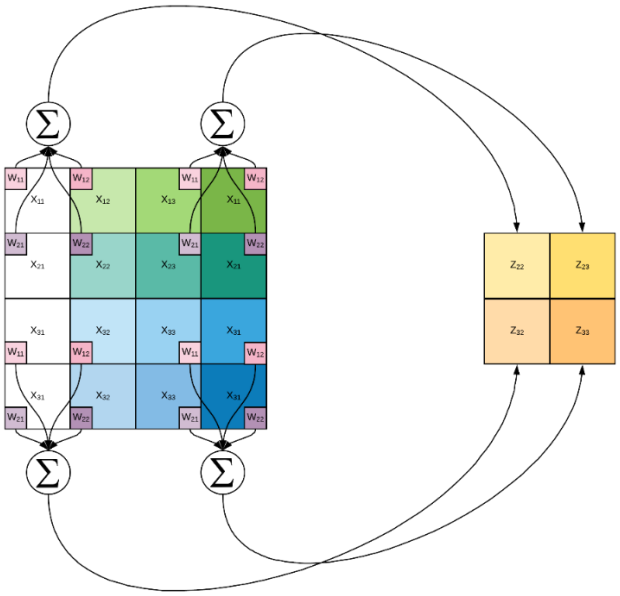


Fig. 2.15 Top: Convolution with padding. The result of the padded convolution of A and B has the same dimension as A. Bottom: Convolution with stride of 2. As a consequence of the increased translation step, each element of the input grid is accessed once

Finally, the concept of convolution can easily be expanded to 1-dimensional, 3-dimensional or n dimensional grid (with n any positive integer) structures, by adapting the parameter vector to a similar structure. As an example, the convolution of two 3-dimensional grid structures $\boldsymbol{A} \in \mathbb{R}^{n\times n}, \boldsymbol{B} \in \mathbb{R}^{m\times m}$ with $(n, m) \in \mathbb{N}^2$ such that $n > m$, can be defined as:

$$\forall i,j,k \in [\![1, n-m+1]\!], \quad (\boldsymbol{A} * \boldsymbol{B})_{i,j,k} = \sum_{l=1}^{m}\sum_{q=1}^{m}\sum_{s=1}^{m}(\boldsymbol{A})_{i+l-1,j+q-1,k+s-1}(\boldsymbol{B})_{k,l,s}$$

Its behavior can easily be linked to the aforedefined concept of neuron by the same identification argument used for their 2-dimensional counterparts.

2.2.7.2 Convolution layer

A convolution layer is essentially similar to the concept of neural layer in the multilayer perceptron, with the neurons previously defined as input variables linear combinations replaced with convolutional neurons. Opposite to traditionally defined neurons, their

convolutional counterparts present some hyperparameters (namely their number of connections, stride and padding). Although some advanced neural architecture allows for the joint use of neurons with different hyperparameters in the same neural layer, a traditional convolution layer typically uses a set of neurons tuned with the same hyperparameters, for both simplicity of implementation and limitation of the number of hyperparameters. As a consequence, a convolutional neural layer can be implemented through the following steps:

- Define a set of neurons (as a set of small real valued matrices)
- Compute the convolution of the input variable and each neuron, resulting in as many pre-activation maps as there is neurons in the layer
- Inject each pre-activation map in an element-wise nonlinear function (typically a ReLU nonlinearity)
- Concatenate every activation map into a 3-dimensional grid.

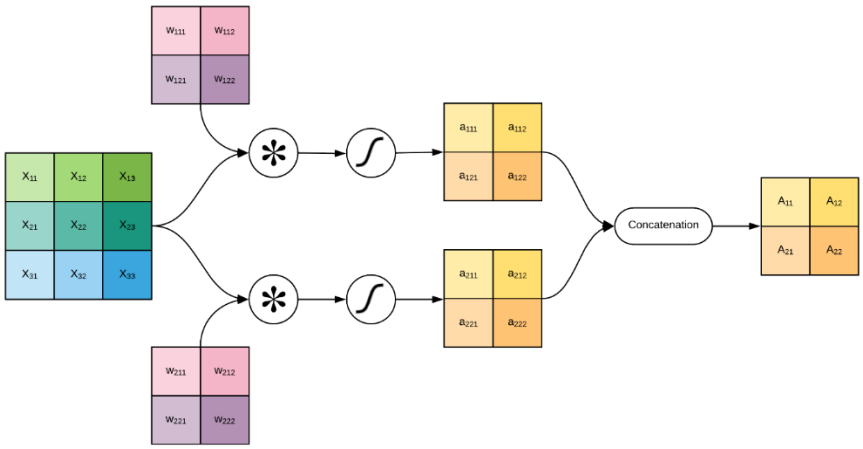


Fig 2.16 Example of a convolution layer with two neurons applied to a 3 by 3 matrix. The layers final output is a 2 by 2 by 2 grid (first and second axes represent the location of the pattern, the final axis resulting from the concatenation of each neuron's convolution output)

In a similar fashion to the multilayer perceptron, convolution layers are typically applied back to back to allow for the modelling of powerful nonlinear relationships characterizing the investigated dataset. However, they present a peculiarity that does not make the implementation of a sequence of layer as straightforward. Indeed, the output of a traditional neural layer is sensibly similar to its input, in the sense that both of them are simple real valued vectors. Convolutional layers, however, due to the final concatenation step, typically output data structures of dimensionality bigger than their input (N+1-dimensional grids for N-dimensional grid inputs). To prevent for this undesirable phenomenon potentially leading to an unreasonable number of unnecessary parameters, convolutional layers receiving the output of previous ones will typically apply to their input a 3-dimensional neuron of depth (last dimension) equal to their input's ones, resulting in a stabilization of the grids' dimensionality across layers.

2.2.7.3 Maximum pooling

Successive convolutional layers can be used to progressively decrease the local information contained in grid-like data structures by including it in the neurons' representation of the investigated dataset. Essentially, each layer combines local information from the previous activation maps, and so on until the input grid variable. The use of convolutional layers alone in convolutional neural network can end up being considerably expensive, specifically with same padding types. Indeed, in these conditions, the size of each activation map does not decrease across layers and make for a considerable amount of computations to perform at each step. Similarly, with a valid type of padding the size of activation decreases relatively slowly across layers (it follows an arithmetic progression). Max-pooling layers were designed in order to efficiently increase the decreasing rate of activation map sizes across layers, in order to speed up computations, typically by outputting the maximum value of its input

matrix's each adjacent sub grid (28). Similar to convolutional layers, several sizes of filter and stride can be used, but most max-pool implementations pool maximum values from 2 by 2 sub grids with a stride of 2 in both directions, essentially taking the maximum value from every adjacent and disjoint 2 by 2 sub grid from its input matrix, as can be seen in figure X.

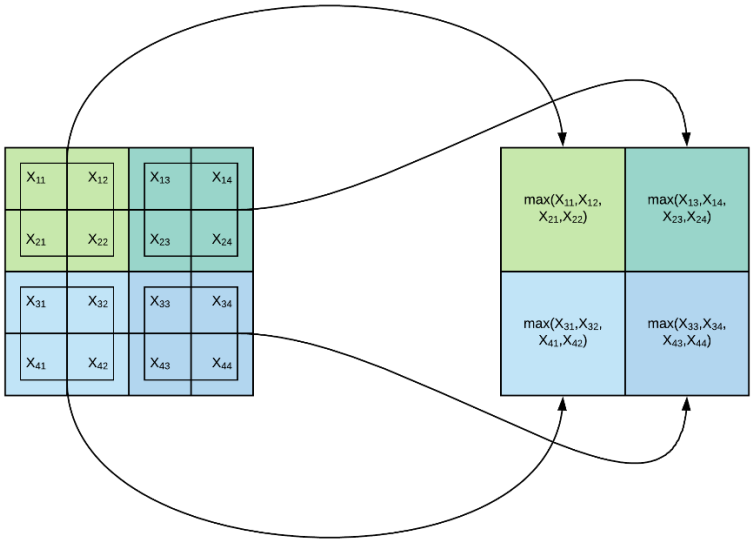


Fig 2.17 Example of max-pool layer of filter size 2x2 and stride 2 applied to a 4 by 4 matrix. The resulting 2x2 grid contains the maximum values from every disjoint 2 by 2 sub grid from the input matrix.

Although the theoretical justification of maximum pooling layers is still not fully understood, a heuristic explanation of their behavior is quite well received in the academic community. Typically, a convolutional neuron is expected to output high values when it detected the pattern it has been trained to recognize. As a consequence, the max pooling transformation allows for a significant decrease of representation sizes (following a geometric progression) while keeping for each neuron's activation map the most important information, as well as its location in the input grid.

2.2.7.4 Convolutional neural network

The concepts of convolutional and maximum pooling layer are typically used together to build what are caller convolutional neural networks, which are a family of parametric models that currently hold state of the art performances on a variety of tasks ranging from machine vision to voice detection. Convolutional neural networks typically incorporate the concepts of convolution and maximum pooling layers into a multilayer perceptron, allowing for the implementation of powerful models of grid like data structures by in the following way:

- Inject the input data into a succession of convolution and maximum pooling layers
- Once the model's representation sizes have sufficiently decreased, inject the last layer's activation map into a standard, multilayer perceptron linking it to the output variable
- Fit the model through cost function optimization with gradient descent and backpropagation, similar to a traditional multilayer perceptron

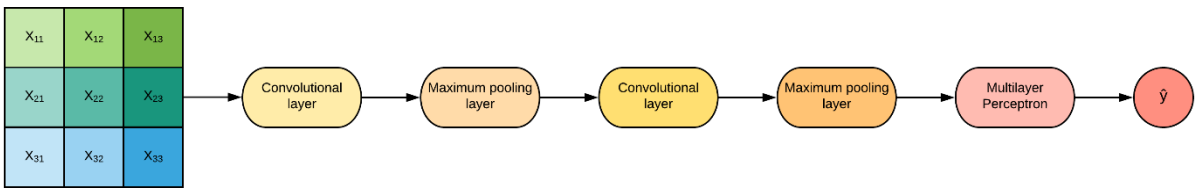


Fig 2.18 Example of a typical convolutional neural architecture. The grid shaped input variable is injected into a succession of convolutional and maximum pooling layers whose final output is injected directly into a multilayer perceptron

The main rationale for using a multilayer perceptron as the last part of a convolutional neural networks is related to the way convolutional neural layers build dense, hierarchical representations of the investigated dataset's input variables. Essentially, each network's layer

converts the local information it is injected with into independent, increasingly complex concepts describing growing sub grids of the input variable. As a consequence, successive layer's activation maps slowly reduce in size until reaching a threshold where internal representation become unlocalized and dense enough for the use of standard, multilayer perceptrons.

This phenomenon was actually illustrated for better understanding in (29), which essentially introduced a method for visualizing the type of input grids maximizing a given neuron in a deep convolutional neural network trained for image classification. Figure 2.19 shows examples of such visualization for neurons selected from different layers, which exhibit the concept of successive, hierarchical representations inherent to deep neural networks:

- The first layer's neurons (top) typically respond to simple, unspecific patterns (colored blobs, linear patterns)
- The intermediate layers (middle) use the simple representations established in previous layer to respond to more complicated concepts
- The final layers (bottom), through successive refinements of representations, manage to encapsulate highly advanced concepts (such as a dog's face) into a single neuron

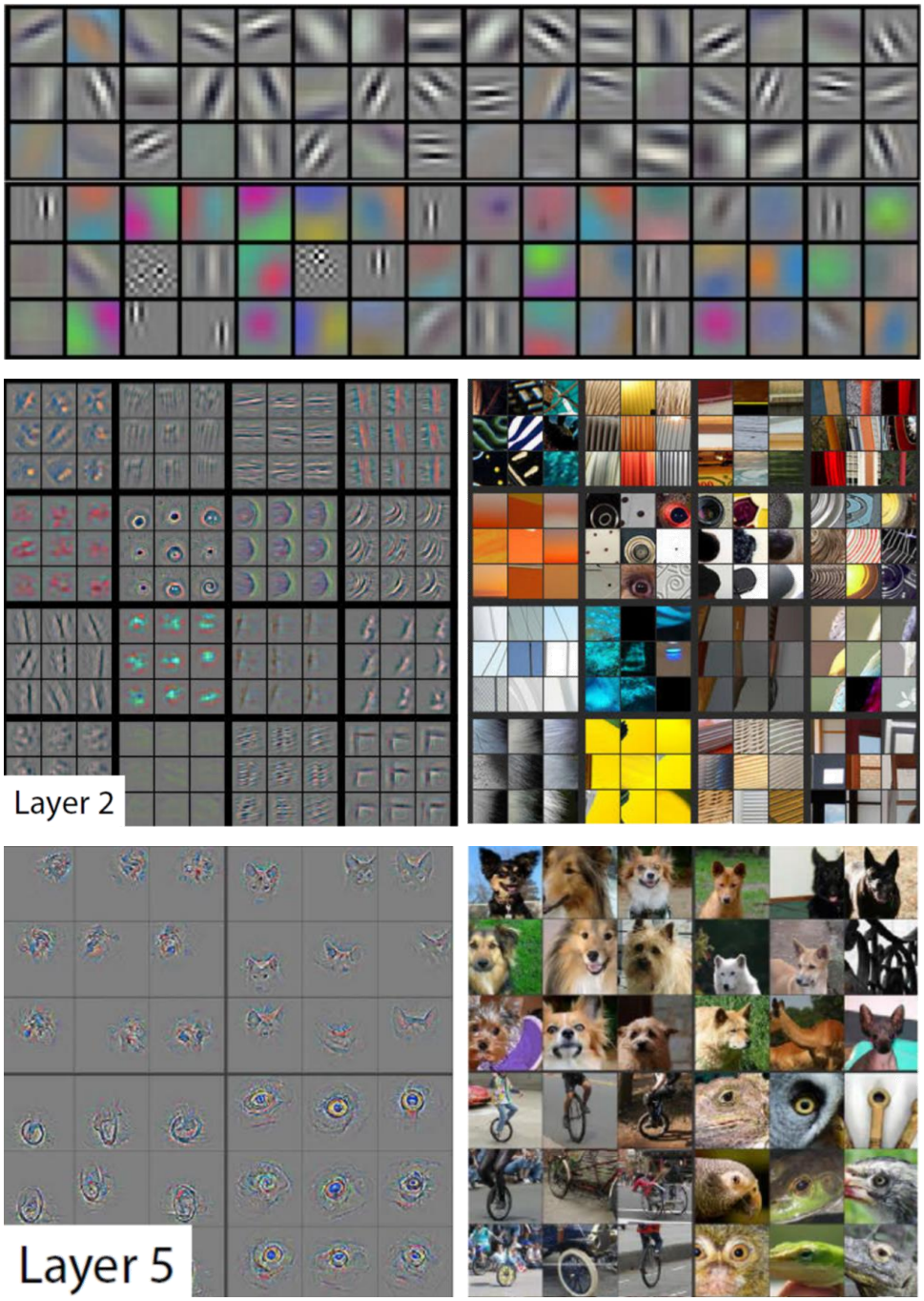


Fig. 2.19 Visualization of maximum activation inputs for several layer depths (29)

2.3 Practical aspects of neural networks

Although the theoretical basis of artificial neural network-based learning algorithms is fairly simple, their practical implementation often results in catastrophic failures (such as, for instance, gradient explosion) without careful considerations (30). As a consequence, a set of practical rules were derived from experiments as well as practitioner insight to improve their overall performances.

2.3.1 Model evaluation

The evaluation of a supervised model's performance is paramount to the practical implementation of a deep neural network. Indeed, machine learning based models are typically used for their predictive capabilities and are consequently expected to not only correctly model the dataset, but also any additional example. However, as the model can only learn from the gathered dataset during the training process, its ability to generalize to never before seen cases cannot be guaranteed from its performance on the sole training data. In practice, because of their ability to model close to any relationship between input and output variables, deep neural networks have a tendency to derive models that are too specific to the investigated dataset. As a consequence, they require a careful analysis of their performance.

2.3.1.1 Training, development and testing sets

The learning process of an artificial network typically consists of minimizing its prediction error on the investigated dataset. As a consequence, the estimation of the model's performance on the data used for training is inherently biased. Indeed, on this specific set of examples, the model usually shows good predictive capabilities, because it was actually chosen amongst an infinite set of different models as the one with best predictive capabilities. It is consequently standard practice to divide the investigated dataset into two distinct,

independent subgroups. One group will be used for optimizing the model's parameters, and the second one will be used to assess the model's performance (31). As the algorithm does not have access to the second group during training, its derived model is not optimized to properly model them and allow for the extraction of an unbiased estimation of the model's predictive power. The group of subjects used for the parameter's optimization is usually called the *training set*. Its counterpart used for model evaluation is called the *testing set*.

Although the training and testing sets are sufficient to obtain an unbiased estimate of a given model's performance, the practical implementation of a machine learning based model is usually done by fitting an important number of different models to the investigated dataset in the search for the best one. For instance, a practitioner could try a lot of different multilayer perceptrons on a given dataset, by varying the number of layers in the networks, or the number of neurons per layers. This search for the best available model can also be seen as an optimization process, that biases the model's evaluation. Indeed:

- Trying to compare the performance of two models on the training set is not reliable, as a bigger network will typically (but not necessarily) yield better training accuracies than a smaller one, without any indication on the actual gain in performance
- Trying to compare the performance of a set of models on the test set adds bias to the estimation. Indeed, once again, the final model will be chosen as the one with best accuracy on the test set, and as a consequence the argument used to illustrate the biased aspect of the training accuracy can yet be used on the test set; the model's performance on the test set is "good" because it was defined as the best among every proposed solution

To counter this additional undesirable bias, an additional group is made from the dataset, called the *validation set* or *development set,* and is used to choose the best model, before using the test set to obtain an unbiased estimation of the latter's performances.

To conclude, the standard methodology to derive a good model from a machine learning algorithm is the following (32):

- Divide the investigated dataset into three distinct parts, the training set, the development set and the test set
- Use the training set to optimize the learning algorithms used during the investigation
- Use the development set to choose the best model among the different ones considered during the investigation
- Use the test set to obtain an unbiased estimation of the final model's performance

An important question during the separation of the investigated dataset into training, development and test set is the percentage of data samples to allocate to each of the sets. Indeed, the model's parameters optimization usually requires a substantial amount of example to behave properly. However, the computation of the models' performances, whether for model selection of final estimation, needs also be reliable.

A standard practice for datasets presenting a sample size less than a ten thousand is to allocate 60% of the samples to the training set, and an equal 20% for both the development and test sets.

2.3.1.2 Bias variance dilemma

During the practical implementation of a machine learning algorithm, two typical unwanted phenomena can arise (33):

- The model's training error is low, but its test error is significantly higher
- Both the model's training and test error are significantly higher than expected by the practitioner

These two phenomena are respectively called over and underfitting. Underfitting is typically observed when trying to model complex relationships with simple models. The model then does not have enough expressive power to identify the investigated dataset's patterns properly, and consequently shows poor predictive ability on both the training and testing set. Overfitting, on the other hand, is a result of using a family of algorithms with so much explanatory power, that they yield unstable models, meaning that two separate datasets gathered from the same phenomenon will be explained by significantly different models. Indeed, the algorithm is powerful enough to account for the random noise typical of statistical dataset, usually to the detriment of the test error.

Figure 2.20 shows a simple example of these concepts in the context of a univariate polynomial regression problem. The left-hand plot shows an attempt to fit the dataset with a linear regression model (polynomial regression of degree 1) and is consequently unable to account for the data's non-linear nature. In contrast, the right-hand figure consists of a polynomial regression model of degree 15. The model has enough explanatory power to obtain a fit close to perfect to the noisy training dataset but yielding an unstable approximation function that equally fails to understand the true underlying function. The centered image shows the polynomial regression best suited for the investigated dataset and could be obtained for instance from a model selection scheme using training, development and test sets.

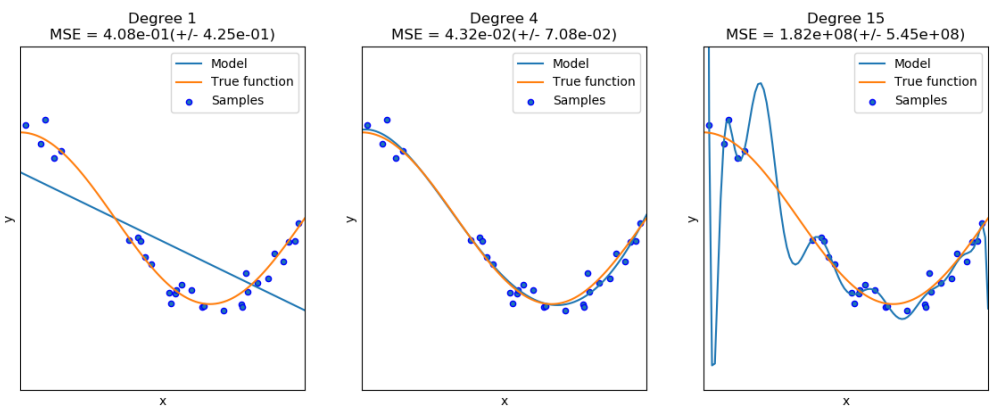


Fig. 2.20 Examples of under and overfitting of a non-linear regression problem. Right: Overfitting model Left: Underfitting model, Center: Finetuned model

This phenomenon relying on the difference of explanatory power between different types of models is called the bias variance dilemma. Typically:

- An algorithm with high bias will typically make too much assumption on the relationship it is trying to model, as in the linear regression example, where a straight line is fitted to a curve
- An algorithm with high variance, however, will be able to model a large number of relationships, but with that gained explanatory power comes the potential to be data-specific, and as a consequence to overfit the dataset

The model selection scheme discussed previously, and for which the development set is paramount, is typically used to fine tune this tradeoff to the investigated dataset.

In practice, deep neural network only underfit the dataset in exceptional circumstances but have a strong tendency toward overfitting that usually requires the use of additional techniques to reduce their variance, such as regularization.

2.3.2 Regularization

Deep artificial neural networks are universal approximators, meaning that they can essentially approximate any continuous function. As a consequence, the neural network's family of models' variance is significantly high, and they are sensibly prone to overfitting, even on very large datasets. Regularization schemes are a variety of techniques used in practical applications to limit this variance, and as such overfitting, typically by preventing the network from learning too complex of a function. Regularization techniques for neural networks are typically divided into two distinct approaches:

- The addition of a penalty term in the cost function (34)
- The drop out method, which is specific to artificial neural networks (35)

2.3.2.1 Penalization methods

The concept of regularization through the addition of a penalty term in the cost function is widely used in both machine learning and statistical analysis. A typical example of such an approach can typically be observed when performing Ridge or Lasso regression (36,37). The rationale behind penalization regularization scheme is to limit the explanatory power of the learning algorithm, by preventing it of outputting artificially too complex of a function. For instance, one can notice from empirical observations that overfitting functions are typically not "smooth". They instead tend to oscillate rapidly between observation to fit the dataset as well as possible. As a consequence, the practitioner might want to constrain the learning algorithm into prioritizing "smooth" function against "non-smooth" ones, even if the latter makes a better job at approximating the training data.

Although quite ambiguous, the concept of a model's "smoothness" can be formally expressed in a relatively straightforward manner. Indeed, for a given parametric model, "non-smooth"

functions will typically present with higher parameter values than “smooth” function. As a consequence, by constraining the model’s parameter values to remain low can limit how “non-smooth” the model can eventually become during the learning process. This can typically be achieved by the addition of a constraint on the model’s parameters in the cost function. Considering a ridge linear regression model on a dataset $\{(X_i, y_i)\}_{0<i<N+1}$ of N observations, the new, regularized cost function can be defined as:

$$L(W, b) = \frac{1}{2N} \cdot \sum_{i=1}^{N} ((W)^T \cdot X_i + b - y_i)^2) + \lambda \left(\|W\|_2{}^2 + b^2 \right) \quad with\ \lambda \in \mathbb{R}^+$$

The cost function’s first term is identical to that of the standard linear regression one. The second term constitutes the regularization constraint. Where a standard linear regression’s model parameters are optimized in order to best fit the data, a ridge regression’s objective is to jointly minimize the model’s fit to the data as well as its squared parameter weights. As a consequence, the model will prioritize parametric solutions that might not best fit the data, but that keep its parameters to low values, typically leading to “smooth” solutions.

The λ value is to be considered a model hyperparameter and denotes the regularization scheme’s importance during the model fitting process. It allows for the practitioner to choose how much importance is to be allowed to the “smoothness” constraint. Typically:

- High values of λ will guide the model toward emphasizing its smoothness requirement. In such conditions, the model’s fitness to the data might be overlooked, which can lead to underfitting
- Low values of λ will result in a cost function close to its unregularized counterpart, and result in overfitting

As a consequence, the fine-tuning of the λ value is paramount to a successful regularization, and its best value is typically determined from testing on the development set.

2.3.2.2 Drop out method

Although penalty-based regularization methods constitute a reasonably simple approach to preventing overfitting which can be applied in a wide variety of situations, its rationale fails to encapsulate the inherent reason for overfitting in neural networks. By using its highly connectionist structure, a big enough artificial neural networks can explain any observed dataset, simply by organizing its neurons and their high number of connections to each other into highly complex data representations. As the network grows bigger, the number of connection between neurons substantially increases, which often lead to overfitting by the conception of features specific to the dataset, thus failing to generalize to new examples. The drop out method is a regularization method that was created in order to prevent neurons to build their features by relying on highly specific combinations of connections, by stochastically shutting down some of the network's neurons during training (35). Essentially, for each training step during the network optimization process, the modelled output variable inference process will be performed by a random subnetwork of the actual implemented neural network. Each neuron builds its representation to become robust to the random absence of input features coming from the previous layer, which prevents them from building artificially too complex and observation specific representations.

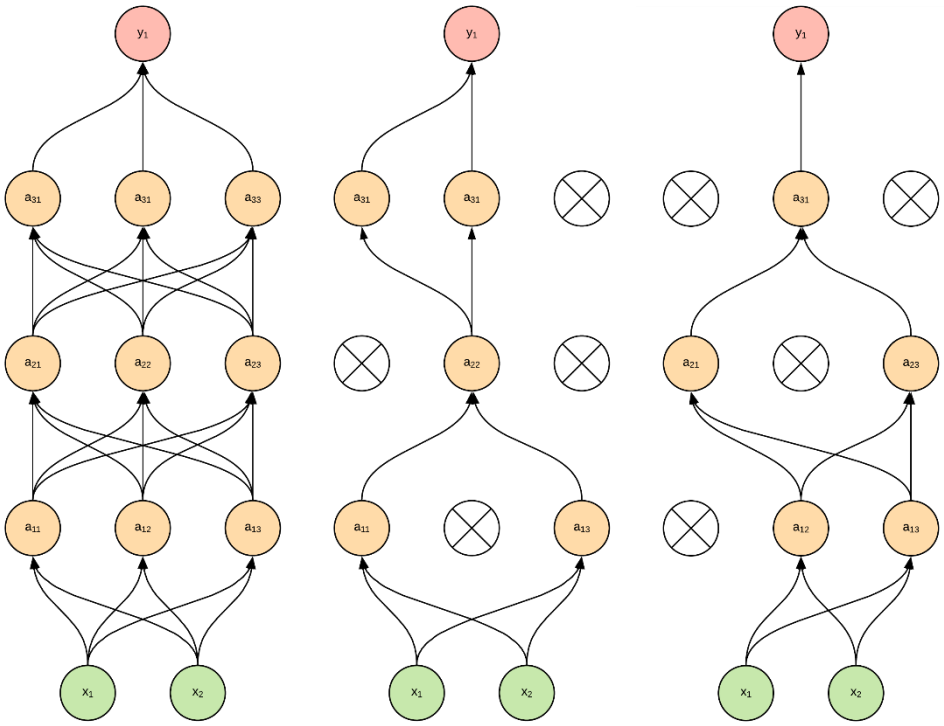


Fig. 2.21 Three layers perceptron with drop out regularization, in its complete version and during two distinct training operations. Neurons are shut down randomly in each layer for each iteration, and the output variable is inferred from the obtained subnetwork

Several implementations of the drop out method have been devised over the years, the traditional approach being:

- Define a neural network for the investigated modelling problem
- During every training iteration, randomly choose several neurons to shut down (each neuron has a shutdown probability of p)
- Mute the selected neurons' outputs to 0
- Use the partial network to infer the output variable
- Backpropagate the gradient through the subnetwork
- Perform the gradient optimization step

- After descent convergence, use the entire network for model evaluation, with every neuron's output multiplied by its drop out probability distribution's expectation

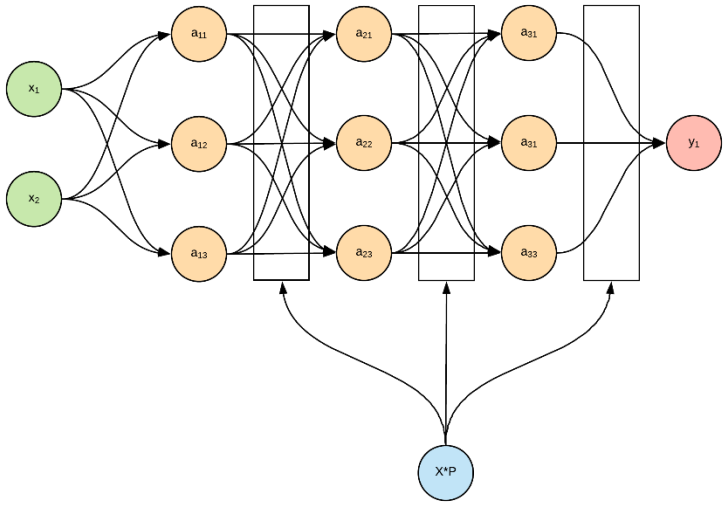


Fig. 2.22 Same neural network as in Figure X. at test time of for true model inference. The neurons' outputs are all used (no drop out) but are multiplied by the drop out probability distribution's expectation

2.3.3 Practical aspects of neural network optimization

The process of optimizing an artificial neural network's parameters is typically difficult (38). Not only does it require the use of gradient based methods on non-convex cost functions which, as mentioned previously, can behave sub-optimally, but neural network's natural properties tend to make said cost function quite unfit to be explored with gradient based methods. Typically, a naïve implementation of an artificial neural network optimized by an equally simple gradient descent algorithm will fail to find a correct parametric model for the investigated dataset. Instead, two different undesirable phenomena will usually arise during optimization:

- The cost function's gradient will become significantly larger than its previous values, causing the gradient descent algorithm to diverge
- The cost function's gradient will become so low that the gradient procedure will stop making any progress, without having reached a local minimum

These two complications are known together as the vanishing and exploding gradient problem (30) and are the reason artificial neural network were discarded by the majority of the academic community as interesting machine learning models until the 2010s. Even though this problem is inherent to deep neural networks, it can in practical cases be overcome enough to obtain really powerful predictive models, typically by using careful parameter initialization schemes and more subtle versions of the gradient descent algorithm.

2.3.3.1 Parameter initialization

The first step required during the optimization of a parametric model with a gradient descent procedure is to initialize the model's parameter, typically randomly. The algorithm typically uses this initialization as a starting point and begins its search for a minimum from it.

When it comes to convex optimizations problems, this starting point does not have any effect on the obtained solution. Indeed, assuming that the algorithm's step rate is correctly tuned, the algorithm is guaranteed to converge toward the function's global minimum. However, this desirable property does not remain true when gradient methods are applied to non-convex functions such as deep neural network's cost functions. Indeed, as can be seen in figure 2.23, two gradient descent procedures initialized slightly differently can converge

toward significantly different solutions. As a consequence, properly initializing a gradient descent in non-convex optimization problems, and as such, in the process of training a deep neural network, has a tremendous impact on its results. In fact, the new artificial neural network parameter initialization procedures introduced in the late 2000s and early 2010s are considered a major factor in the first successful practical implementations of deep learning algorithms.

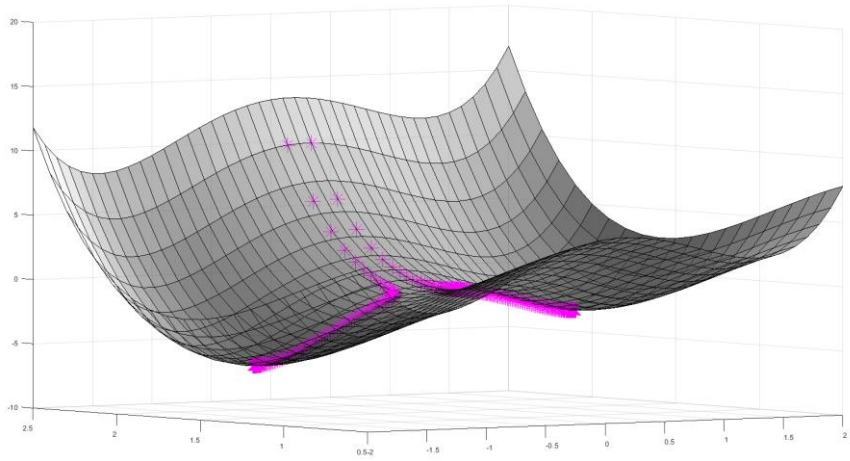


Fig 2.23 Two gradient descent procedures on a non-convex function, with different initialization (starting point). Even though the two procedures are initialized closely, the resulting local optima are sensibly different

Historically, the first parameter initialization scheme resulting in the successful training of deep neural networks was obtained through the use of unsupervised layer-wise pretraining (39). However, recent work has led to the creation of initialization schemes that outperform such pretraining both in performance and computational costs, such as, for instance, the Xavier initialization procedure (38).

The Xavier initialization scheme is a probabilistic parameter initialization method that relies on heuristic considerations on information flows within a deep neural network using

hyperbolic tangent nonlinearities. For an L-layered neural network, the i^{th} layer's parameters are sampled from a random normal distribution with 0 mean and variance σ^2 such as:

$$\sigma^2 = \sqrt{\frac{1}{n^{i-1}}} \quad with\ n^{i-1}\ the\ number\ of\ neurons\ in\ the\ network's\ i-1^{th}\ layer$$

In a similar fashion, the i^{th} layer of a deep neural network using ReLU nonlinearities is to be initialized by sampling parameters from a random normal distribution with 0 mean and variance σ^2 such as:

$$\sigma^2 = \sqrt{\frac{2}{n^{i-1}}} \quad with\ n^{i-1}\ the\ number\ of\ neurons\ in\ the\ network's\ i-1^{th}\ layer$$

2.3.3.2 Advanced gradient-based optimization method: Adagrad

While a proper parameter initialization can significantly improve the quality of a neural network parametric model solution extracted from a given dataset, using the classical implementation of a gradient descent procedure during the learning process can result in substantial practical difficulties:

- As the dataset and the neural network get bigger, the gradient gets increasingly more expensive to compute, both in time and memory, and rapidly becomes practically intractable on machines with limited computing power
- Even with proper parameter initialization, the cost function's surface remains difficult to explore. The gradient descent can end up "stuck" on a function's neighborhood with a gradient close to 0 that is not a local optimum (degenerate critical point or saddle points), as is shown in figure 2.24

Two major concepts have been introduced to adapt gradient descent methods to these difficulties:

- Stochastic (or mini-batch) gradient descent
- Gradient descent with adaptative learning rates (Adam optimization)

For the practical optimization of a deep neural network's cost function, these two improvements on the traditional gradient based procedure are usually combined into powerful optimization algorithms.

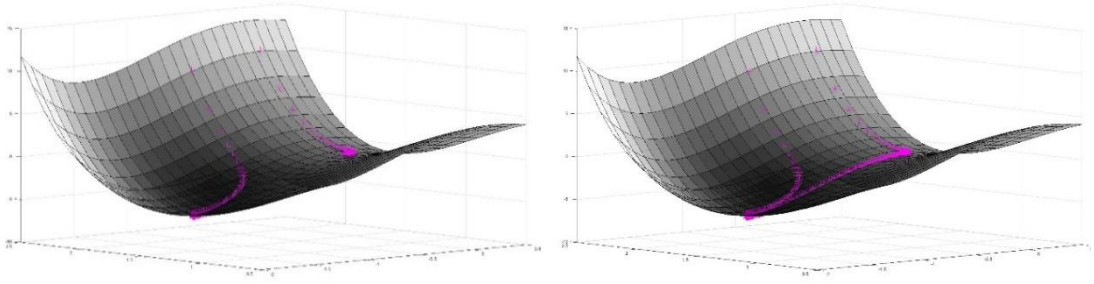


Fig 2.24 Two standard gradient descent procedures with different initialization. Left: one procedure has converged toward a minimum, the other is stuck on a saddle point. Right: The second descent algorithm "got out" of its saddle point, and converged to the same local optima as the first version, but took a significantly larger amount of time

2.3.3.2.1 Mini-batch gradient descent

Theoretically, the gradient of any given neural network's associated cost function can be obtained through the use of the backpropagation algorithm. By making use of the cost functions' inherent additive nature, its gradient computation is divided into sub gradient computations at the sampled individual level. Although an efficient method to automatize the computation of the gradient process, this method presents a significant drawback when

applied to dataset with large sample sizes. Indeed, it requires the capacity to store and compute as many gradients as there are examples in the dataset, usually resulting in memory overflow and unreasonably long computation times.

Mini-batch gradient descent offers a solution to this physical problem by approximating the cost function's gradient through the use of probabilistic sampling (40). For a given iteration of the descent procedure, the model's cost function is estimated from a small number of subject (a mini-batch) randomly sampled from the dataset. As a consequence, this approximation of the cost function's gradient can be computed through backpropagation on a limited subsample of examples, significantly diminishing its computational cost. By successive random sampling at each step, the gradient descent algorithm can then be approximated with limited computational costs.

A typical implementation of a mini batch gradient descent with mini-batch size m during the fitting of a linear regression model on a dataset $\{(X_i, y_i)\}_{0<i<N+1}$ of N observations, can be described as follows:

- Define a random permutation σ of $[\![1, N]\!]$ that will be used to shuffle the dataset
- For each gradient descent iteration step s, compute an approximation of the cost function

$$L_s(W, b) = \frac{1}{2N} \cdot \sum_{i=m*s}^{m*(s+1)} \left((W)^t \cdot X_{\sigma(i)} + b - y_{\sigma(i)}\right)^2)$$

- Compute the approximated cost function's gradient (through the backpropagation algorithm in the case of artificial neural network based models), and use it to update the parameters (standard descent step)

- Once every subject from the investigated dataset has been used for cost function approximation (in other words when $m * s > N$), reiterate the process with a new random permutation
- Continue the process until convergence

The number of individuals to sample for cost function estimation is considered a model hyperparameter and has a significant impact on both the method's computational efficiency and the outputted minimum's quality. Indeed, low mini-batch size will require lower computational requirements when computing the objective's gradient, but in the other hand typically make for a more unstable optimization algorithm, as the statistical estimation of the cost function gets poorer with few sampled examples. In practice, typical mini batch size values lie between 50 and 200 examples.

2.3.3.2.2 Adaptative Moment estimation optimisation (Adam)

Although mini-batch gradient descent allows for the approximation of a usually prohibitively expensive gradient for the optimization of artificial neural networks, it alone isn't enough to allow for optimization of the difficult functional surfaces typically observed in deep artificial neural network learning processes. As aforementioned, the gradient descent's ability to converge towards a local minimum is strongly conditioned on the proper tuning of its hyperparameter, the learning rate. Indeed, as it quantifies the size of the step to be taken in the direction defined by the gradient, strong learning rate values yield an unstable optimization algorithm that fails to stop when arriving close to the found local optima, and weak values result in an unreasonably slow optimization algorithm. Unfortunately, the fine tuning of this hyperparameter is sometimes not efficient on its own to obtain a good optimization algorithm. Indeed, using the same step size value to update every parameter

implicitly makes the assumption that the investigated functional surface has roughly the same properties in every direction, which might not be the case, especially when it comes to neural network cost functions. The Adam optimization method (41) was introduced specifically to address this issue and provides a gradient based descent algorithm which adapts its learning rate to each parameter individually. Its definition is substantially similar to the traditional gradient descent's one, with for only difference a modified update rule and consists, for a function f from $\mathbb{R}^n$ to $\mathbb{R}$ and four real number η, ε, β_1 and β_2, of the following:

- Initialize $a_0 \in \mathbb{R}^n$ randomly and $[m_0, v_0] \in \mathbb{R}^n \times \mathbb{R}^n$ to $[0, 0]$
- Iterate through the parameter update rule at step $t \in \mathbb{N}$:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla f(a_{t-1})$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)\big(\nabla f(a_{t-1}) \odot \nabla f(a_{t-1})\big)$$

$$\hat{m}_t = \frac{m_t}{(1 - \beta_1)^t}$$

$$\hat{v}_t = \frac{v_t}{(1 - \beta_2)^t}$$

$$\forall i \in [\![1, n]\!], \qquad (a_t)_i = (a_{t-1})_i - \frac{\eta}{\sqrt{(\hat{v}_t)_i} + \varepsilon}(\hat{m}_t)_i$$

$$with\ (x)_i\ the\ i^{th}\ parameter\ of\ vector\ x$$

- Stop iterating when a_t has converged toward a local minimum

Once again, η, ε, β_1 and $\beta_2$are to be considered model parameters. Although the tuning of η has a significant impact on the optimization procedure's quality, ε, β_1 and β_2 are usually chosen to be respectively 10^{-8}, 0.999 and 0.9 (41).

2.3.4 Normalization

As aforementioned, improving on the gradient descent algorithm used during the training of deep neural networks, through careful initialization and refined update rule, can significantly improve the quality of the obtained parametric model, as well as speed up convergence time. However, using a better optimization algorithm is not the only way possible to improve the process of searching for the minimum of a non-convex function. Indeed, despite the substantial advances introduced in gradient based optimization procedures, a deep neural network's cost function remains substantially difficult to optimize. A different approach to overcome this inherent complication is, instead of improving on the optimization algorithm, to try to modify the cost function itself into a simpler, but equivalent one, that can then be optimized in a more straightforward manner. This idea has led to two main methods that significantly improve the quality of deep neural network models, input and batch normalization.

2.3.4.1 Input normalization

The cost function of a deep neural network being built upon samples from the investigated dataset, its shape and nature highly depend on the latter. As a consequence, applying simple transformation to the dataset can potentially have a positive effect on the model's cost function, by for instance improving its compatibility with gradient based optimization methods. Although many options are available to the practitioner to preprocess a dataset, the most widely used method of data preparation for the implementation of deep neural network is input normalization.

As aforementioned, one major difficulty with optimizing neural network models comes from the heterogeneity of their cost function across directions, which lead to the implementation

of adaptive learning rates gradient methods. Input normalization works toward an additional improvement on the optimization process, this time by modifying the cost function itself to be more homogenous across directions (42).

The process in itself is fairly straightforward and consist of normalizing every input variable from the investigated dataset to be of mean 0 and variance 1 by applying to them both an offset and a linear scaling estimated from the dataset. For a dataset with m input variables $(x_i)_{0<i<m+1}$, input normalization can be performed by applying the following affine transformation to each input variable:

$$\forall i \in [\![1, m]\!], \qquad f(x_i) = \frac{x_i - \mu(x_i)}{\sigma^2(x_i)}$$

$$with\ \mu(x_i)\ and\ \sigma^2(x_i)\ the\ mean\ and\ variance\ of\ x_i, estimated\ from\ the\ dataset$$

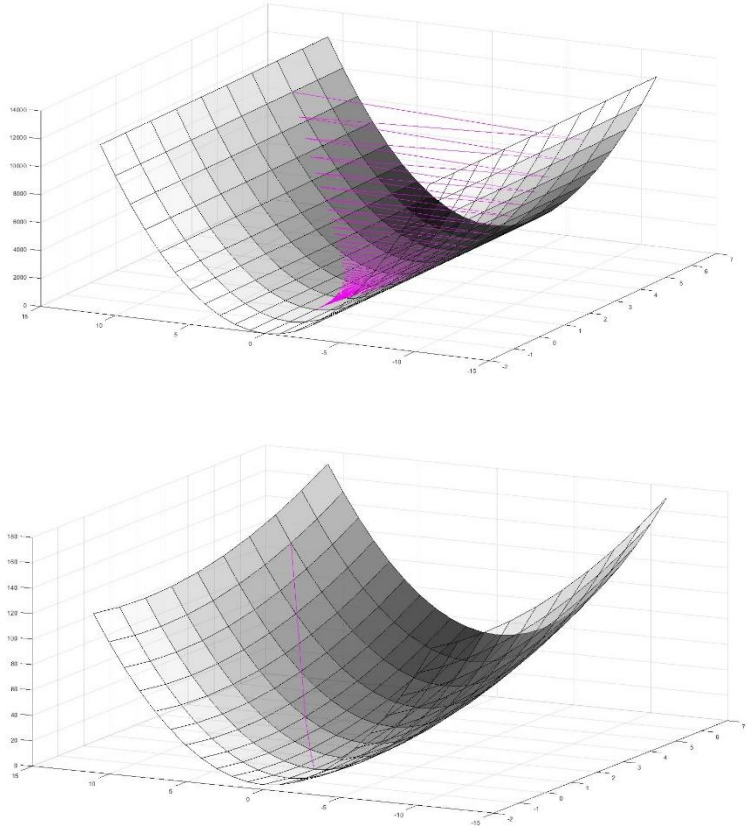


Fig. 2.25 Standard gradient descent on a linear regression model's cost function before and after input normalization. Top: unnormalized function with learning rate of 0.0098. Convergence is reached in 280 iterations, and higher learning rate lead to divergence. Bottom: normalized function with learning rate of 0.3. Convergence is reached in <10 steps

The positive effect of input normalization on the optimization of a model's cost function through gradient based methods can be visualized in figure 2.25, which illustrates an example of linear regression model's cost function and its dedicated optimization process both before and after input normalization. Before normalization, the cost function presents a strong slope heterogeneity across directions. As a consequence, the maximal allowed value for the descent procedure's learning rate is strongly limited to small values, to prevent gradient instability and divergence. The same cost function after input normalization presents a functional surface with a slope that is more homogenous across directions, which allows for much higher learning rates, and converges much faster to a local optimum.

Although this phenomenon is also typically addressed by the implementation of adaptive learning rate descent methods such as the Adam optimization algorithm, a joint use of both methods typically increases the convergence speed of the training process in artificial neural network-based models.

2.3.4.2 Batch normalization

Although Batch normalization shares some conceptual and practical similarities to input normalization, its main rational is quite different.

The power of deep neural network models comes from its ability to use its layers to build increasingly complex representations of the investigated dataset, by letting each layer build

on its predecessor. During the learning process, each layer adapts itself to the previous layer's outputs in order to build the best representation at its level, thus contributing in improving the model's overall quality. However, the previous neural layer is also modifying itself, for the exact same reasons. As a consequence, for a given layer, the parameter update fulfills two purposes:

- Adapting the layer to the changes made in the previous layer's representation
- Adapting the layer to improve the model's performance on the training set

Batch normalization was introduced to reduce the need for each layer to adapt to changes in its predecessors and "focus" on the modelling task at hand, by fixing each layer output's mean and variance to 0 and 1, respectively, typically with an affine transformation similar to input normalization (42). However, unlike input normalization, which is a rather straightforward procedure, batch normalization requires more thoughts in its practical implementation. Indeed, as the learning process progresses, the network's layer outputs are continually changing, and with them their first order statistics. As a consequence, the normalization step is updated at every mini-batch gradient descent iteration, and the mean and variance are estimated for every layer from the currently used batch of training examples.

Although this normalization procedure allows the model to learn on a more stable version of its internal representations, fixing the mean and variance of each layer's outputs might actually hurt the model in itself, by preventing each layer to make use of its nonlinearities correctly. As a consequence, a final affine transformation is typically applied to the normalized outputs, with its offset and scaling factors considered model parameters that are trained by the model through the gradient process.

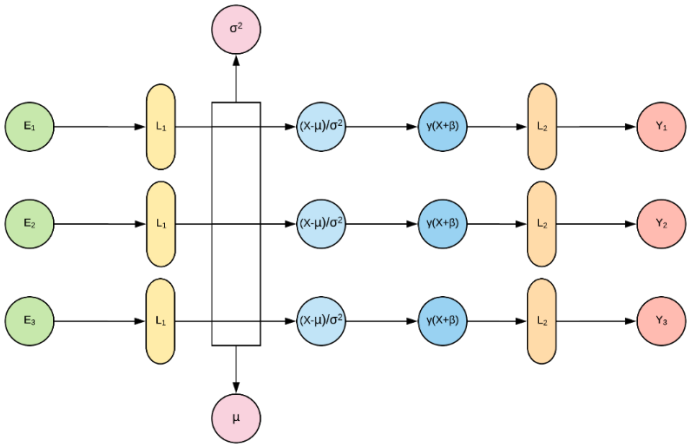


Fig 2.26 Example of a two layer perceptron with batch normalization between the first and second layers, with a batch of 3 examples (E_1, E_2, E_3)

For a given model' layer l with m output neurons $(a_i)_{0<i<m+1}$ and two vectors of parameters $(\gamma_i)_{0<i<m+1}$, $(\beta_i)_{0<i<m+1} \in \mathbb{R}^m$, the batch normalization process during an iteration of the mini-batch gradient descent process can be formally defined the following way (42):

- Estimate mean and variance for the layer's every output neurons from the current iteration's batch
- Normalize the batch outputs to null mean and unitary variance in a similar fashion as input normalization
- Rescale the outputs by applying the following parametric affine transformation:

$$\forall i \in [\![1, m]\!], \qquad f(x_i) = \gamma_i \, (x_i + \beta_i)$$

2.3.5 Hyperparameter tuning

Deep artificial neural network models present a substantial number of hyperparameters:

- The learning rate
- The batch size

- The number of hidden layers
- The number of neurons per layer
- The regularization hyperparameters
- Other model specific hyperparameters (stride parameter in convolutional layers for instance)

Finding appropriate values for this set of hyperparameter is paramount to the successful practical implementation of a deep learning based algorithm to a given modelling problem. As mentioned before, the estimation of the hyperparameter values needs to follow a specific methodology, to avoid overfitting and biased estimation of model accuracy:

- One model with a given set of hyperparameter is extracted from the training dataset
- The best set of hyperparameter is inferred by testing several combinations on the development set
- The best model's accuracy is estimated on the test set

Although most hyperparameter tuning in deep artificial neural network follows this methodology, the question of how to come up with potentially good sets of hyperparameters to try on the modelling task, has yet to be defined.

Several procedures have been devised over the year to explore the hyperparameter space to discover proper model settings, varying from powerful, but computationally expansive solutions to simpler search procedures, with the two most widely used approaches being grid and random search.

2.3.5.1 Grid search

The grid search approach to hyperparameter optimization is one of the most widely used method in machine learning. It consists of limiting the space exploration to a subset of values for every hyperparameter. The resulting value space forming an approximation of the original in the shape of a grid-like structure. More formally, for a given model with n hyperparameters $(\lambda_i)_{0<i<n+1} \in \mathbb{R}^n$ a grid search can be implemented as following:

- For each hyperparameter λ_i, define a number of candidate to sample m_i and a bounded interval to sample them from
- For each hyperparameter λ_i , divide the selected interval into $m_i - 1$ regular intervals $\left([(\lambda_i)_1,\ (\lambda_i)_2], -, \left[(\lambda_i)_{m_i-1},\ (\lambda_i)_{m_i}\right]\right)$ and select $S_{i,m_i} = ((\lambda_i)_j)_{0<j<m_i+1}$ as the subset of λ_i values to explore
- Train a model with the training set for every hyperparameter combination $(\lambda_{c1}, -, \lambda_{cn}) \in S_{1,m_1} \times - \times S_{n,m_n}$
- Select the best set of hyperparameter as maximizing the model's accuracy on the development set
- Test the model's accuracy on the test set

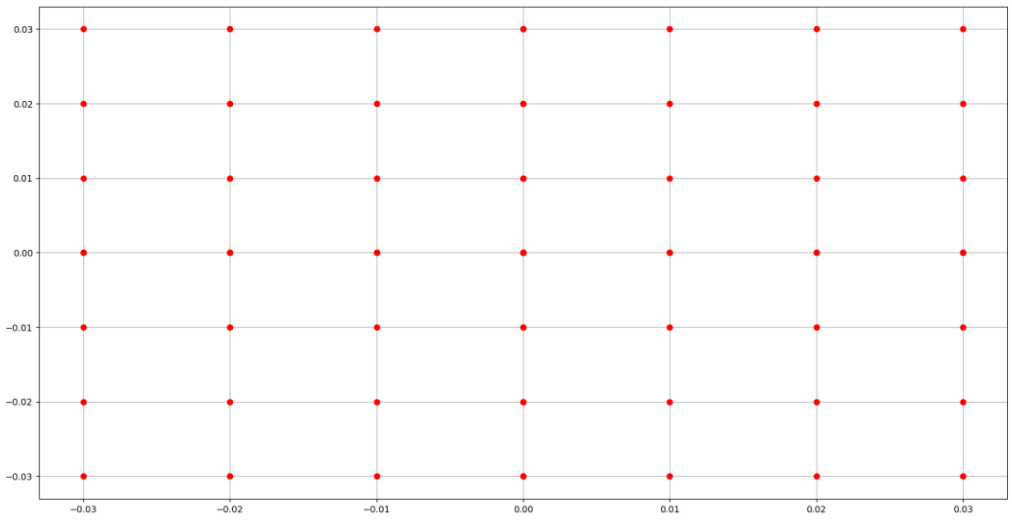


Fig 2.27 Grid search-based exploration of [-0.03, 0.03]^2 with 7 samples per axis

The grid search process can be refined by iterating several grid searches on increasingly smaller areas of the hyperparameter space, each one identified as a promising hyperparameter region from the previous grid search.

Although grid search methods can easily be implemented in parallel, significantly speeding the hyperparameter tuning process, the computational cost of deep artificial neural networks typically prevent it on reasonably sized machines. In addition, the quality of grid-based exploration approaches tends to substantially decrease in high dimensional spaces that can typically be observed in deep neural network hyperparameters.

2.3.5.2 Random search

Grid search methods methodically explore the entire bounded subspace that it is provided. As a consequence, and specifically in deep artificial neural network which present a high number of hyperparameter, these processes are substantially computationally expansive. In addition, grid search methods are specifically adapted to tuning hyperparameters with

comparable impact on the model's performances. However, it has been found that the influence of hyperparameters on the quality of the final algorithm is substantially heterogenous; oftentimes, some hyperparameters have close to no influence on the model. As a consequence, random searches-based exploration methods have been found to significantly outperform their grid-based counterparts in the finetuning of hyperparameters in deep artificial network-based models.

the optimization of a given model's hyperparameter through random search is fairly straightforward to implement, and consists simply in sampling random combination of hyperparameters, assessing their quality on the development set and selecting the combination yielding best model performances. More formally, for a given model with n hyperparameters $(\lambda_i)_{0<i<n+1} \in \mathbb{R}^n$ a grid search can be implemented as following:

- Define a number of hyperparameter set to sample m
- For each hyperparameter λ_i , define a bounded interval to sample from
- Sample m hyperparameter set from their respective sampling intervals, typically through a uniform random distribution
- Train a model with the training set for every hyperparameter combination
- Select the best set of hyperparameter as maximizing the model's accuracy on the development set
- Test the model's accuracy on the test set

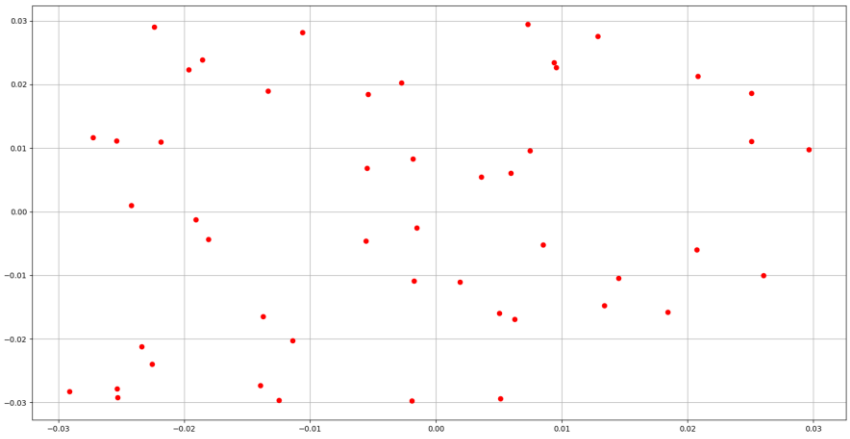


Fig 2.28 Random search based exploration of [-0.03, 0.03]^2 with candidates sampled from a bivariate uniform random distribution. The same number of candidates were sampled as in the previous grid search visualization example

In a similar manner to grid search methods, a random search process can be refined by iterative random searches in increasingly smaller areas of the hyperparameter space identified from the previous search result as promising.

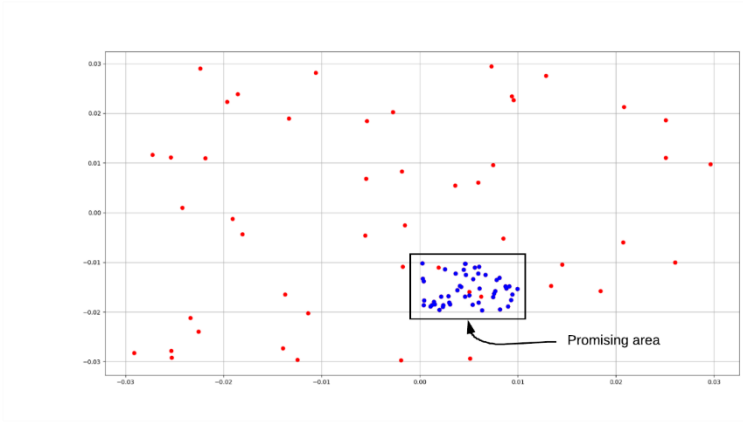


Fig. 2.29 Refined random search. A first random search (in red) is performed. An area within the search is selected as promising (for instance because hyperparameter sampled from it all yielded good model performance), and a new random search (in blue) is performed on selected area

The superiority of random search based hyperparameter tuning compared to grid-based approaches mostly can be explained from a combination of two phenomenon:

- Random search investigates a significantly higher number of different values for each parameter than grid search
- Some hyperparameters might not have any significant effect on the model's performance

A simple example can be considered in order to better visualize the impact of these two properties on the resulting obtained set of hyperparameter. Let $(\lambda_1, \lambda_2, \lambda_3)$ be a set of three hyperparameters with λ_3 having close to no significant impact on the model's predictive quality.

A grid search hyperparameter optimization procedure with n values sampled for each hyperparameter will require to fit n^3 distinct models. However, due to λ_3's absence of effect on the model's performances, only n^2 *efficient* hyperparameter (λ_1 and λ_2) settings will have been tested.

A random search process with n^3 random sampled hyperparameter sets, however, will have tested n^3 distinct combination of λ_1 and λ_2, thus resulting in a finer space exploration and, usually, better model performances.

2.4 Unsupervised learning and autoencoders

2.4.1 Autoencoder

Both recurrent and feedforward neural network are defined as supervised learning algorithms. The model is built by setting its parameters in order to optimize its ability to

approximate the model's exogenous variables. However, clustering procedures are typical examples of an unsupervised machine learning algorithm, in which a structure within the observed data is to be extracted.

Autoencoders are a family of neural networks that adapts the concepts of supervised learning through feedforward and recurrent networks to enable their use in unsupervised settings.

The objective of autoencoders is to use neural networks' ability to define complex, nonlinear interaction to embed a dataset into a richer representation, which can then be used to perform several tasks such as dimensionality reduction, data compression or visualization.

An autoencoder is typically built from two separate neural networks, an encoder f and a decoder g. The encoder takes a set of N observations $\{X_i\}_{0<i<N+1}$ as inputs, and outputs for each of them a vector h_i.

The decoder then takes the encoder's output as entries and tries to rebuild the corresponding input data. The two parts of the autoencoder are jointly optimized in order to minimize a reconstruction error, typically the mean squared cost function (43):

$$L(X) = \frac{1}{2N} \cdot \sum_{i=1}^{N} (g(f(X_i)) - X_i)^2$$

The idea behind autoencoders is for the encoder to encapsulate in its output as much information as possible about the data for the decoder to be able to reconstruct them with optimal accuracy, learning along the way interesting features and properties about the investigated dataset.

A simple implementation of an autoencoder is however rarely useful. Indeed, the easiest way for the decoder and the encoder to minimize the reconstruction error is to both converge toward identify functions. Indeed, for any dataset sample X:

$$g = f = Id \;\Rightarrow\; L(X) = \frac{1}{2N} \cdot \sum_{i=1}^{N} \big(g\big(f(X_i)\big) - X_i\big)^2 = \frac{1}{2N} \cdot \sum_{i=1}^{N} (X_i - X_i)^2 = 0$$

As a consequence, the difficulty in implementing an autoencoder is to find ways of preventing the encoder and the decoder simply learning an identity function (6). This can be done by constraining the network's explanatory power, typically by adding a regularization constraint during the learning process, thus making the learning task "difficult" for the neural network.

The study of autoencoders is a growing area of research, that lead to the elaboration of entire families of algorithms, such as the undercomplete or denoising autoencoders.

2.4.2 Undercomplete autoencoder

The undercomplete autoencoder is the most straightforward example of autoencoder managing to successfully capture salient features in dataset. From a practical point of view, its implementation does not differ in any way to a naïve autoencoder. The main rationale consists of constraining the number of output neurons in the encoder's output to be less than the number of input variables in the dataset. As a consequence, to perform the reconstruction task, the encoder, forced to compress the data in a smaller representation, will need to capture the data's most informative characteristics (6).

As such, an undercomplete autoencoder can be seen as an expansion of kernelized principal component analysis in which the kernel function is learnt from the dataset. In fact, an undercomplete autoencoder with linear decoder and mean squared error cost function will learn the same data decomposition as a linear principal component analysis (6).

In spite of its relative simplicity, the undercomplete autoencoder presents several drawbacks:

- The requirement for the embedding to be of dimension inferior to the input data can result in a loss of information that might impair the analysis
- Both encoder and decoder are limited in their size for an interesting embedding to be extracted. Indeed, with too big of an encoder or decoder, the autoencoder can potentially overfit, and learn a perfect mapping that does not capture any information on the dataset (overfitting)

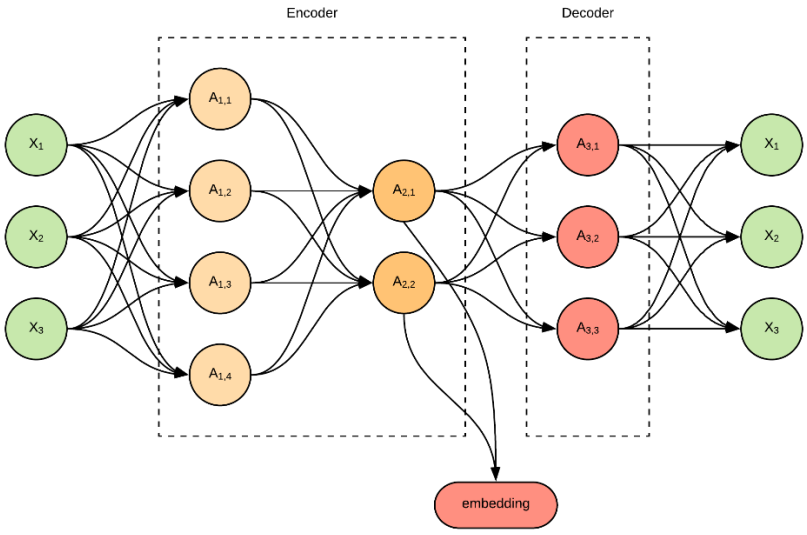


Fig. 2.30 Example of undercomplete autoencoder. The investigated dataset is composed of 3 variables. The two-layers encoder outputs two variables, which forces the encoder to compress the data efficiently for the one-layer decoder to reconstruct the data properly

2.4.3 Denoising autoencoder

Although fairly straightforward to implement, undercomplete autoencoders present some limitations on the type of embedding they can extract from a dataset. Denoising autoencoders are another type of unsupervised artificial neural networks that manage to overcome some of these drawbacks (6).

The concept of denoising autoencoder (44) is to prevent the learning of identity functions for the encoder and decoder by slightly changing the task at hand. An additive probabilistic noise (typically a gaussian random noise of mean 0 and predefined variance) is added to the input data before injecting it into the encoder. The reconstruction error L, however, is computed using the clean input data.

$$\tilde{X} = X + \mathrm{N}(0, \sigma)$$

$$L(X) = \frac{1}{2N} \cdot \sum_{i=1}^{N} \left(g(f(\tilde{X}_i)) - X_i\right)^2$$

As a consequence, the model learns during the optimization process to clean the corrupted input data from its additive noise. Contrary to their undercomplete counterpart, a denoising autoencoder's extracted embedding need not have a dimensionality lesser than the input data's. In addition, the additive noise can be considered as a form of model regularization, and both the decoder and the encoder's architectures can be made significantly larger, thus allowing for the identification of potentially more complicated nonlinear relationship in the dataset. The noise's variance, however, constitutes an additional model hyperparameter that needs to be carefully fine-tuned by the practitioner.

2.5 Ladder networks, an example of semi supervised neural network

Artificial neural networks are currently the state of the art in modelling complex, non-linear interactions within a dataset. Their flexibility has allowed their use in a range of diverse applications with never achieved before results, such as in image and speech analysis, or natural language processing. One of their major drawback, especially in supervised learning, is the requirement for a significantly high sample size for them to perform correctly. Indeed, practitioners usually consider that deep neural networks should be used cautiously when

investigating datasets with less than 5000 labelled examples. Nowadays, a typical supervised deep learning dataset's sample size will sit in the hundreds of thousands.

This constitutes a major drawback when it comes to the biomedical sciences, where the collected dataset's sample sizes rarely go to these orders of magnitude.

As an example, when considering a project of implementation of an automated diagnosis algorithm. Even though the digitalization of information enables for the collection of potential endogenous variables on a massive scale, the output variable here needs to be evaluated by an expert medical practitioner, which is tedious in the best-case scenario and simply intractable otherwise.

However, these situations, presenting massive availability of unlabeled data, and scarcity of labelled example, fit perfectly the semi-supervised machine learning paradigm.

Although fundamentally different, supervised and unsupervised neural network algorithms share two interesting common traits:

- Both learn through the optimization of a cost function
- Both are specialized in building powerful representations of the investigated dataset

As a consequence, it only seems natural to investigate how to use the abilities of autoencoder to build powerful internal representations of the data without any supervision to help a supervised neural network to learn to make powerful predictions on a partially supervised dataset.

The study of semi supervised neural networks is a relatively new field of study in deep learning, and a number of architecture have been proposed over the years, the ladder network and its variations being the current state of the art (45).

2.5.1 Ladder networks

Ladder networks (46) constitute a family of semi-supervised neural learning algorithms that efficiently combine unsupervised and supervised learning into one composite learning process by using some of the latest concepts introduced in the deep learning academic literature, such as batch normalization and skip connections. Conceptually, a ladder network is a particular type of denoising autoencoder, whose encoder is also used to make predictions on the dataset's output data. Its learning process is based on the joint optimization of two separate cost functions, for $\left\{\{(X_i, y_i)\}_{0<i<N+1}, \{X_j\}_{0<j<M+1}\right\}$ a dataset of N labelled observations and M unlabeled ones:

- An unsupervised, reconstruction cost $L_{unsup}(X)$ similar to what can be seen in traditional autoencoders
- A supervised cost function $L_{sup}(X, y)$ adapted to the modelling problem, indicating the encoder's ability to model the output data from the input data

The encoder can then be used on every available example, labelled or not, by optimizing the joint cost function:

$$\begin{cases} L(X,y) = \lambda\, L_{sup}(X,y) + (1-\lambda)\, L_{unsup}(X) & for\ labelled\ examples \\ L(X,y) = (1-\lambda)\, L_{unsup}(X) & for\ unlabelled\ examples \end{cases}$$

$\lambda \in [0; 1]$ is a model hyperparameter expressing the importance of the supervision task compared to the unsupervised one. Typically:

- Lambda values close to 0 prioritize the unsupervised cost function to the supervised objective. The network's ability to approximate the output variable is then usually

limited, as the representations learnt from the data are not specific to the problem enough

- Lambda values close to 1 prioritize the supervised cost function. As a consequence, the neural network focuses on correctly estimating the output variables from the input variables, in a similar fashion as a traditional supervised network. However, overfitting can occur without sufficient labelled examples

The fine-tuning of the λ hyperparameter is paramount to the network's semi supervised learning process' efficiency.

As a conclusion, the implementation of a ladder network can be summed up in 5 steps:

- Autoencoder implementation
- Supervised cost function definition
- Supervised cost function definition
- Joint cost function optimization
- Model validation

2.5.2 Autoencoder architecture

The ladder network's autoencoder is inspired by the aforedescribed concept of a denoising autoencoder. A denoising autoencoder learns properties from the dataset by taking as inputs the dataset's explanatory variables, corrupted with added random noise, and tries to output their uncorrupted counterparts. The ladder's autoencoder is built in a similar fashion, with two main differences:

- Random noise is added to the network between every layer of the encoder
- The decoder is asked to reconstruct every uncorrupted layer from the encoder

2.5.2.1 Encoder architecture

A ladder network's encoder component is fairly to implement. Any type of feedforward neural network (multilayer perceptron and convolutional neural networks, for instance) with a batch normalization operation on every layer can be converted into their encoder version by addition of a random noise before the batch normalization step.

Theoretically, in a similar fashion to traditional implementations of denoising autoencoders, several types of additive random perturbations can be used to corrupt the data flow. However, specificities in the decoder architecture make the adaptation of the network to more exotic noise distribution less straightforward. As a consequence, most practical implementations of ladder network use a gaussian random noise of mean 0 for perturbation. The noise's variance is considered another model hyperparameter and can be shared across the whole network or fined tuned for every layer.

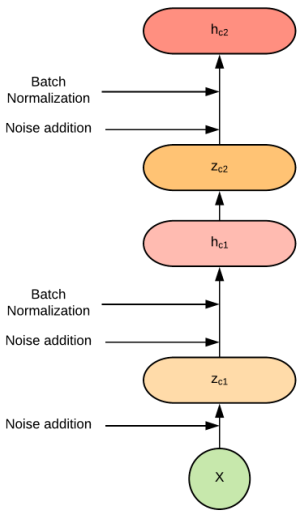


Fig 2.31 Example of a two layers ladder's network encoder component. Noise addition and batch normalization are used between every pre-activation (here z_{ci}) and activations (here h_{ci})

2.5.2.2 Decoder architecture

If the ladder network's encoder component can easily be derived from its traditional denoising counterpart, the definition of the decoder's architecture is less straightforward.

Indeed, where the encoder and decoder of a regular autoencoder only share connections over the network's embedding, the ladder network's decoding architecture includes skip connections from the encoder between each layer. Therefore, the decoder's architecture is constrained by the encoder one's such that for an encoder with N hidden layers:

- The decoder has an equal number of layers
- The decoder's i^{th} layer has the same number of neurons as the encoder's $N - i^{th}$ one (to allow for computation of a reconstruction error)

Each layer of the decoder is computed in a similar fashion. As an example, the ith layer of a N-layers decoder will be built such that:

- A linear transformation is applied to the *i-1*th layer's outputs. The result of said transformation needs to have the same dimensionality as the encoder's $N - i^{th}$ layer's output
- The linear transformation's output is batch normalized into an output u_i
- The batch normalized output is combined with the encoder's $N - i^{th}$ layer's output (after noise addition and batch normalization) to form the i^{th} layer's output
- The process is repeated until reaching the encoder's first layer

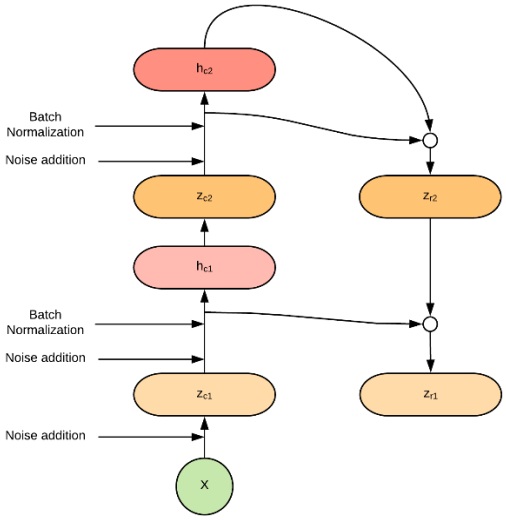


Fig 2.32 Example of a two layers ladder's network decoder architecture. The skip-connections between every layer from the encoder to the decoder give the network its name. The ∘ symbol denotes the combination function

Several methods can be used for the combination of the encoder and decoder's signals, that ranges from simple addition and injection into a nonlinearity to complex, probability theory inspired solutions. Although the original definition of ladder network followed the latter approach, it was recently outperformed by an element wise combination scheme based on a variation of the standard multilayer perceptron caller the augmented multilayer perceptron (47).

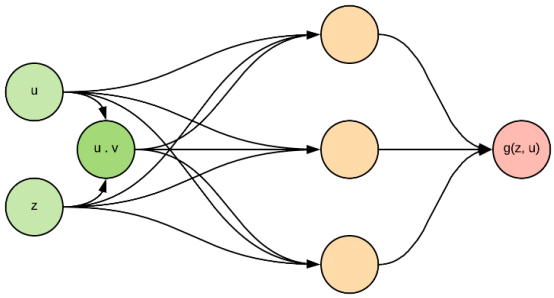


Fig 2.33 Example of an augmented multilayer perceptron with one hidden layer of three neurons

The augmented multilayer perceptron architecture is essentially a parametric function g combining two real valued inputs into one real valued output through the use of a multilayer perceptron taking as entries the two input values as well as their product. As an example. An augmented multilayer perceptron with 3 hidden layers of 2 neurons each is defined as:

$$\forall\,(u,v) \in \mathbb{R}^2, g(u,v) = \boldsymbol{W}_3 \cdot LReLU(\boldsymbol{W}_2 \cdot LReLU(\boldsymbol{W}_1 \cdot [u,v,u \cdot v] + b_1) + b_2) + b_3)$$

With:

- $\boldsymbol{W}_3, \boldsymbol{W}_2 \in \mathbb{R}^2 \times \mathbb{R}^2$
- $W_1 \in \mathbb{R}^2 \times \mathbb{R}^3$
- $b_3, b_2, b_1 \in \mathbb{R}^2$

The augmented multilayer perceptron's parameters initialization process is however slightly different than what is expected from a traditional perceptron. Indeed, its original authors recommend a random initialization following a normal distribution with zero mean and a variance to be considered as a model hyperparameter.

2.5.3 Unsupervised and supervised cost functions and optimization

Once a ladder network is properly implemented, the learning process can begin. To allow for semi supervised learning, the ladder network is taught through the optimization of a weighted average of two distinct cost function, one for the unsupervised task, that can be applied to both labelled and unlabeled examples, and one to model the dataset's output variable, that is exclusive to the labelled examples.

2.5.3.1 Unsupervised cost

As previously mentioned, the ladder network shares strong similarities with a denoising autoencoder. As a consequence, its unsupervised cost function will be defined in the same fashion, with a reconstruction error typically based on the least square distance.

However, the network is built for the decoder not to denoise the input variables, but every layer within the network's encoder, thus resulting in several layer-wise reconstruction costs available, one for the encoder's every layer, to be jointly optimized. A straightforward implementation of the latter would be for instance to derive it from a similar procedure to the denoising autoencoder, by computing the mean square loss function between the encoder layer and the decoder layer. However, this task is made trivial by the introduction of the skip connection between the two layers. In the same way as an overcomplete autoencoder would behave, the decoder could in such a set up easily learn an identity function to maximize the layer wise reconstruction set up, which would prevent the whole network of learning any concept of interest from the investigated dataset.

To counteract this undesirable effect, the reconstruction errors are not built by comparing the encoder and decoder, but rather by comparing the decoder's outputs to a completely uncorrupted version of the encoder:

- The decoder's outputs are computed as defined before
- The input data is injected into the encoder neural network without any corruption by additive random noise
- The reconstruction cost for each layer is defined as the mean square error between the decoder's outputs and the truly uncorrupted encoder counterpart's ones

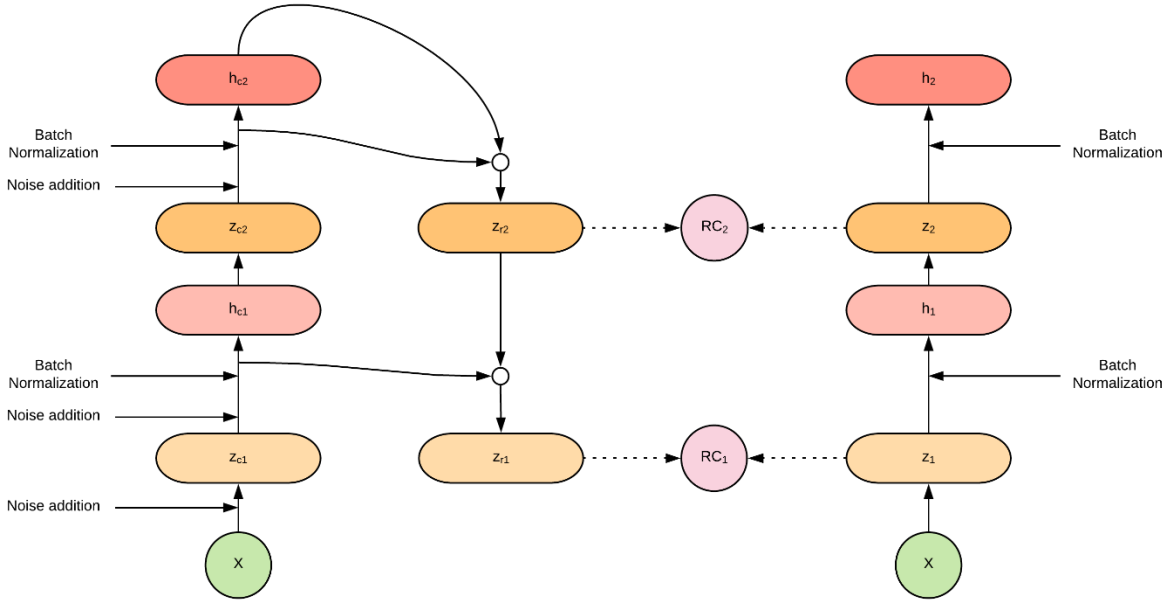


Fig 2.34 Example of layer-wise, local reconstruction costs for a ladder network with two hidden layers. The encoder is used twice, one with injection of additive noise to compute decoder outputs, and one without to compute the reconstruction costs (RC_1 and RC_2)

The ladder network's unsupervised cost function being based on a combination of several, local denoising objectives, a similar approach to the weighted average used to fuse supervised cost and unsupervised cost can be used, in order to give the model prior knowledge about the importance of unsupervised learning in each of the encoder's layer. Typically, for an architecture with a K-layers encoder applied to a semi supervised dataset $\left\{\{(X_i, y_i)\}_{0<i<N+1}, \{X_j\}_{0<j<M+1}\right\}$, the unsupervised cost function can be written as:

$$L_{unsup}(X) = \sum_{i=1}^{K} \lambda_i \cdot RC_i(X) \qquad with \quad \sum_{i=1}^{K} \lambda_i = 1$$

With $\forall i \in [\![1, K]\!]$:

- $\lambda_i \in \mathbb{R}$ a model hyperparameter
- $RC_i(X) \in \mathbb{R}$ the reconstruction cost of the i^{th} layer evaluated on data sample X

In a similar fashion to what was observed in the weighted compound cost function, the influence of the hyperparameters $(\lambda_i)_{0<i<k+1}$ have a powerful interpretation. Indeed, for a given layer:

- Lambda values close to 0 diminish the importance of the reconstruction task for the layer. As a consequence, the layer will be free to organize its representation to better its performance on the supervised task
- Lambda values close to 1 augment the importance of the reconstruction task for the layer, and it will prioritize extracting a general representation of the data

Although any kind of value combination can be tried for $(\lambda_i)_{0<i<k+1}$, a powerful heuristic can be derived from the empirical conclusions drawn from visualizing a convolutional neural network's hierarchical representations. Indeed, as aforedescribed, neural networks tend to create general representations of the data in the lowest layers of a neural network, and higher layers tend to adapt these low-level representations into more sophisticated concepts that become more specific to the supervised learning task with each successive layer.

The idea of semi-supervised learning, specifically in deep learning, being to use widely available but unlabelled example to capture general features of the dataset to be used to improve the modelling task, it seems only natural to use higher lambda values for the encoder's lower layers, and lower lambda values for its highest layers, thus working together with the way neural network naturally process information. Although not theoretically backed, this heuristic consistently gives the best performance results for ladder networks tested on standard, semi supervised benchmark datasets (47).

2.5.3.2 *Supervised cost*

So far, the ladder network has been described as an unsupervised learning algorithm derived from the denoising autoencoder family. To enable its use in semi-supervised settings, an additional supervised cost function must be added to the global learning objective, guiding the network into making a good predictive model, while retaining insightful information from the unsupervised objective. The result of the autoencoder being the encoder's output, it is only natural to use it as a basis for the supervised task. As a consequence, the supervised learning process in a ladder network is rather straightforward:

- Use the network's encoder to predict the output variable
- Compute the supervised loss on the dataset's labelled part

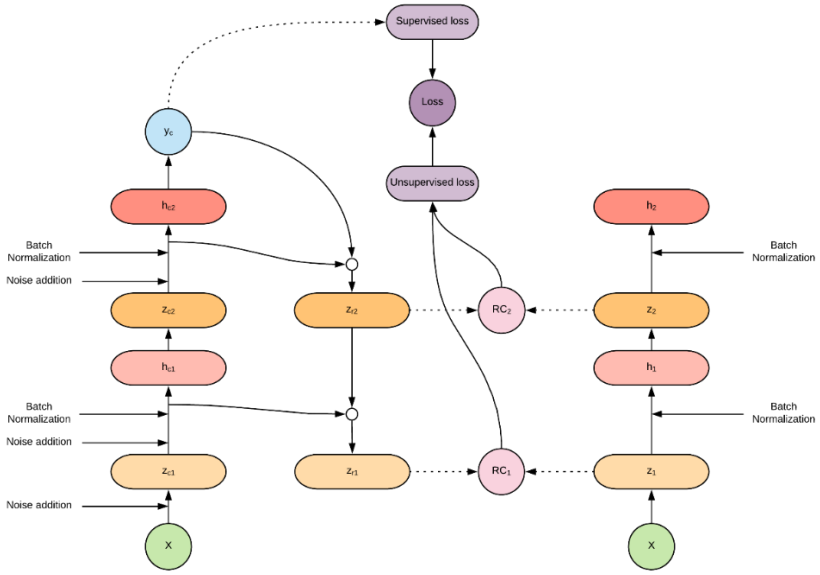


Fig 2.35 Full ladder network in learning setup

It is worth mentioning that the supervised loss could have either been obtained from the predicted output derived from the encoder, or from its uncorrupted counterpart used in the

computation of the unsupervised loss. However, the additive noise corruption used in the encoder can also be seen as a regularization scheme for the supervised learning process, and a such should be helpful in avoiding overfitting in the overall model.

2.5.4 Model evaluation and final model inference

The question of evaluating a semi-supervised machine learning model is much closer to the evaluation of supervised algorithms to the evaluation of their unsupervised counterparts. Indeed, the network's end goal still remains to predict the output variable from the input variable with as much accuracy as possible, the unsupervised component of the model being used only to boost said accuracy. As a consequence, the model's performances will be evaluated with the same methods as would be done for a standard, supervised neural network. However, the impact of the semi-supervised architecture can be assessed, for instance by comparing, for a given architecture, model performance from an optimization on the sole supervised objective against performance with the dual objective function (with properly tuned hyperparameters).

For the specific case of a ladder network, a question remains for the evaluation of the model's performance. Should the network's ability to predict the output variables given the input variables be evaluated on the encoder, or on its uncorrupted counterpart? An intuitive answer to this question would be to choose the uncorrupted version of the encoder to output the network's actual decision. Indeed, as previously mentioned, the addition of noise in the data can be considered as a regularization scheme making the model more robust to small, local perturbations in the input data. However, in situations where the model is to be used to make the best prediction possible, the noise can only hurt the model's ability to link input and output variable.

As a consequence, the model's evaluation, as well as its predictions in a deployed set up should be done from the uncorrupted encoder. As the encoder-decoder component of the network is never used in the final inference process, it is unnecessary, and can be discarded after completion of the optimization procedure. The final network remaining is then a standard, feedforward neural network.

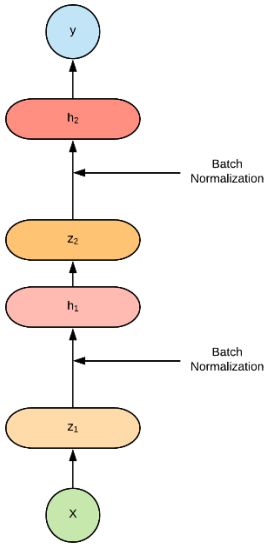


Fig 2.36 Final ladder network after parameters optimization in standard, inference mode. The encoder-decoder architecture is discarded after optimization and only a third of the whole model remains, making the network much smaller

2.5.5 Evaluation

As for unsupervised machine learning in general, evaluating the quality of an embedding extracted from an autoencoder is a complicated problem.

A straightforward manner of assessing for embedding quality is to use the implicitly supervised paradigm of autoencoder, and to assess the autoencoder's potential overfitting by comparing the mean reconstruction error on the training dataset and an evaluation dataset excluded beforehand, similarly to the standard methodology used for performance evaluation in supervised settings. This simple method presents however a potential

drawback. Indeed, although testing the autoencoder on an unseen dataset might identify possible overfitting phenomena, thus identify “bad” -as in non-representative of the dataset- embeddings, it does not allow for the assessment of a good embedding.

Another approach, commonly used in the evaluation of word embeddings in natural language processing, is to qualitatively assess the embedding’s quality for instance by using visualization algorithms such as the T-SNE projection algorithm, or by using nearest-neighbors methods:

- Select randomly a number of samples from the dataset
- Identify the selected samples’ nearest neighbors in the extracted embedding (according to the Euclidean distance, for instance)
- Evaluating the relationship in the input data between the neighboring groups identified

3 Deep clustering of longitudinal data in epidemiology

3.1 Background

The question of longitudinal data analysis in cohort studies is a topic of importance in epidemiology. In this particular type of study, variables are observed repeatedly over long periods of time. As a consequence, a proper statistical analysis of the gathered data needs to account for the variable’s evolution through time just as well as its value.

Sequence clustering procedures have been shown to perform particularly well in that task. The main idea behind cluster analysis is to group individuals together according to some predetermined similarity criterion, such that the resulting groups of individuals all display

similar properties, which can then be used by the analyst to identify and extract useful insight from the data.

Even though the notion of similarity between individuals was traditionally designed to exploit geometrical properties of multivariate, non-temporal datasets, it has been successfully expanded to longitudinal datasets through a range of measurements, such as, for instance, temporal Euclidian distance.

Several clustering procedures have been developed for the analysis of longitudinal data. The kml package (48) introduced in the open source statistical programming language R, for instance, relies on a K-mean algorithm coupled with a longitudinal distance measurement, which is traditionally defined as the aforementioned temporal Euclidian distance.

Another method, proc traj (49), was introduced in the software SAS. This approach differs significantly from the kml K-mean, and offers a likelihood approach based on gaussian mixture of parametric models fit beforehand on the phenomenon to be studied. Both approaches present advantages and drawbacks, and reasons to pick amongst the different available methods are not well defined.

As aforementioned, even though clustering is traditionally approached as a statistical analysis problem in epidemiology, it can equally be considered as an unsupervised machine learning problem. In addition, artificial neural networks provide natural frameworks for both the modelling of sequential datasets (for instance with recurrent neural networks) and the analysis of datasets in an unsupervised setting (with autoencoders). These two architectures can be combined into a recurrent autoencoder, which allows for the embedding of longitudinal data into a low dimensional, non-temporal representation while conserving the dataset's statistical properties.

The following experiments were designed to investigate the use of such methods on a real dataset, and compares its results to ones obtained with more traditional methods of longitudinal data clustering, namely the kml and proc traj procedures. A simulated dataset will also be used to explicit situations where deep learning could have a special interest over standard classical approaches.

3.2 Method

3.2.1 Overall

In order to investigate the potential application of neural networks to longitudinal data clustering, two weel established longitudinal data clustering procedures –the proc traj algorithm from SAS and the kml function from R- were selected for comparison against the proposed deep learning clustering architecture.

The kml procedure (48) is an adaptation of the K-mean clustering algorithm to longitudinal data.

Traditional K-mean clustering is an iterative algorithm consisting of two phases. First, k points $(m_1, m_2, \dots, m_k)$ are randomly initialized in the data space. A k partition is then defined from these points by attributing each data point to the nearest m_i, according to a predefined distance. The (m_i) are then updated to be their cluster mean. This step is then repeated until convergence of the cluster centers.

The kml procedure is essentially identical. The k points are replaced with randomly sampled sequences, and the distance is selected through the panel of existing distance for longitudinal data (eg. L1, L2, Fréchet distance, Dynamic Time Warping).

Proc traj (49) constitutes a different approach to the same problem, and consists of fitting a mixture of parametric models to the data through an Expectation-Maximization algorithm.

3.2.2 Recurrent autoencoder and deep longitudinal clustering

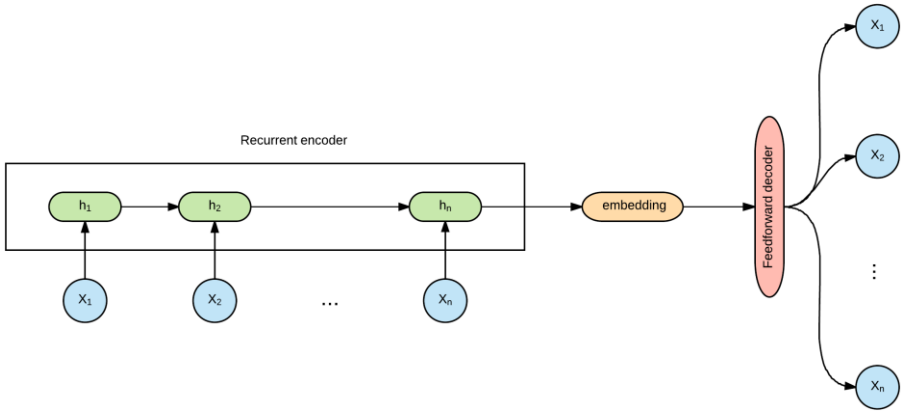


Fig. 3.1. Neural network used to perform longitudinal clustering: A recurrent neural network "reads" the studied longitudinal data and outputs a low dimensional vector. This vector is used as an entry for a multilayer perceptron, which then reconstructs the original longitudinal data

While undercomplete autoencoders are traditionally used with feedforward neural networks, they can be expanded to the analysis of longitudinal data, for instance by using a recurrent neural network as the encoder (50). The extracted embedding can then be used as a low-dimensional, non-temporal representation of the studied sequential data, on which traditional, multivariate clustering algorithms can be applied. Several architectures are available to the practitioner, among them the one described in fig. 5, selected to perform the experimentation to be reported:

- A recurrent neural network in the setting described above is used to sequentially read the data sequences
- The last hidden layer is converted into a low dimensional vector with a linear, feedforward neural network
- The low dimensional representation is fed to a feedforward neural network

- The feedforward neural network's output is then used to copy the data sequence through multiple linear regression (as much as there is values in the data sequences)
- All the model parameters, from the recurrent network, the feedforward networks and the linear regressions, are tuned by the optimization of a least square cost function
- The cost function is optimized with the Adam optimization procedure
- The function's gradient is computed with the backpropagation algorithm
- Once the optimization process is terminated, the data's extracted low dimensional representation is checked for internal consistency
- The low dimensional embedding obtained from the recurrent network's last hidden layer is used to perform visualization and multivariate data clustering.

3.2.3 Embedding validation

As described before, validating the quality of an autoencoder's extracted embedding is a complex process for which no definitive metric is known to this day. The validation method chosen for this study is an empirical validation by visualization of several local neighborhoods within the embedding, as described before:

- Select randomly a number of samples from the dataset
- Identify the selected samples' nearest neighbors in the extracted embedding (according to the Euclidean distance, for instance)
- Evaluating the relationship in the input data between the neighboring groups identified

3.2.4 Analysis

3.2.4.1 Proc traj

As SAS is not an open-source software, the proc traj procedure was not directly used in this experiment. Instead, for the real data experiment, the proc traj based data partition was retrieved from the results of the original study (51), and the procedure was discarded from use in the simulated dataset experiment . In this reference, the number of clusters was fixed to six, the value obtained through the optimization of a parsimonious index. It is noticeable that, in the present paper, the number of clusters retained for all three approaches was also fixed to 6 to ensure good comparability.

3.2.4.2 Kml

The partition obtained with longitudinal distance-based K-mean was obtained using the related R package kml. The algorithm was run 20 times with random initialization and the solution with maximum Calinsky-Harabasz criterion (52) was selected for comparison with the other procedures.

3.2.4.3 Recurrent Autoencoder

The recurrent autoencoder was manually implemented in Tensorflow (53), a dedicated python based library for deep learning. The selected architecture was defined as afordescribed. Hyperparameters (architecture of the recurrent encoder, number of neurons in the encoder, embedding dimensionality, number of layers in the decoder, number of neurons per layers in the decoder) were tuned toward a satisfying embedding according to the defined quality evaluation methodology. For purpose of visualization, and given the investigated sequences' length, the embedding dimensionality wat set to two.

Once a satisfactory bi-dimensional, atemporal embedding has been extracted from the dataset and validated, clustering was then extracted from it by traditional, Euclidian distance

based, K-mean clustering. Similarly, as for kml clustering, 20 partitions where extracted from random initialization, and was kept the best cluster set in regard to the Calinsky-Harabasz criterion.

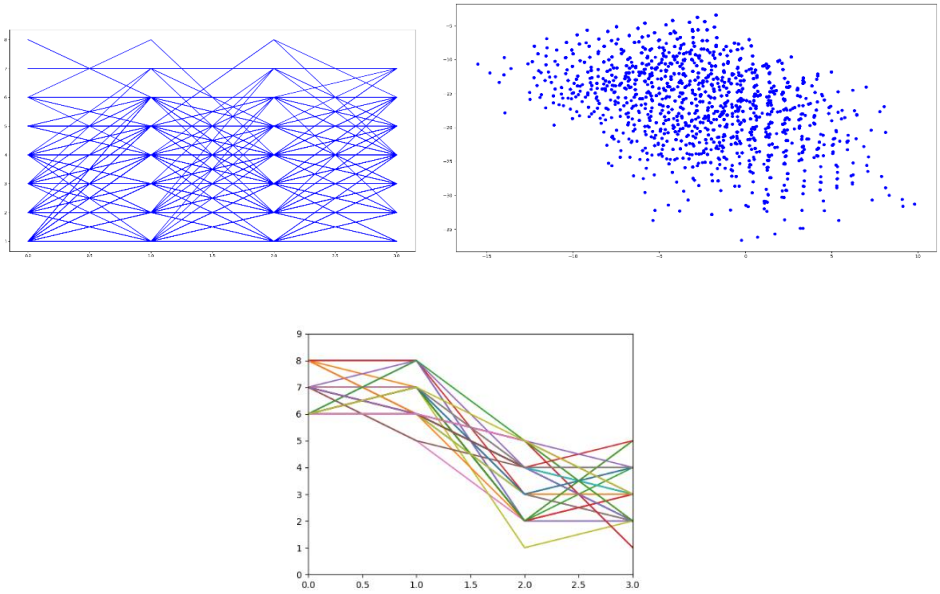


Fig. 3.2 Upward left: Longitudinal plot of the dataset's trajectories, Upward right: Plot of the corresponding bi-dimensional embedding, Bottom: longitudinal plot of one of the embedding's local neighborhood. All trajectories clearly share similarities

3.2.4.4 Coherence of the three partitions

To establish quantifiable coherence between the three partitions, the posterior probabilities of an individual to belong to a given cluster were computed for the three clustering procedures:

- As a gaussian mixture-based model, the proc traj procedure gives us natural access to the desired probabilities
- The kml procedure offers a built-in cluster membership probability tool

- The auto-encoding based clustering membership probabilities were computed on the extracted embedding, under an assumption of gaussian prior for each cluster, with mean and covariance computed on the cluster's members

The extracted probabilities' correlation matrix is finally represented graphically using a spherical representation (54).

3.3 Results

3.3.1 Real-data application

3.3.1.1 Dataset

The three aforementioned clustering procedures were compared with each other on a dataset obtained from the E3N cohort study (51), which describes the evolution of 80110 women's body shape across time. Measurements consist of an 8 levels ordinal scale representing "extreme thinness" to "obesity", and were established at 4 times in the subject's life:

- Around the age of 8 years
- At menarche
- At ages 20 to 25 years
- At ages 35 to 40 year

Longitudinal data clustering of these scale measurements has been used in a series of papers with proc traj. It is of interest to ensure this clustering is retrievable with the kml algorithm and the autoencoder as well.

3.3.1.2 Cluster confrontation

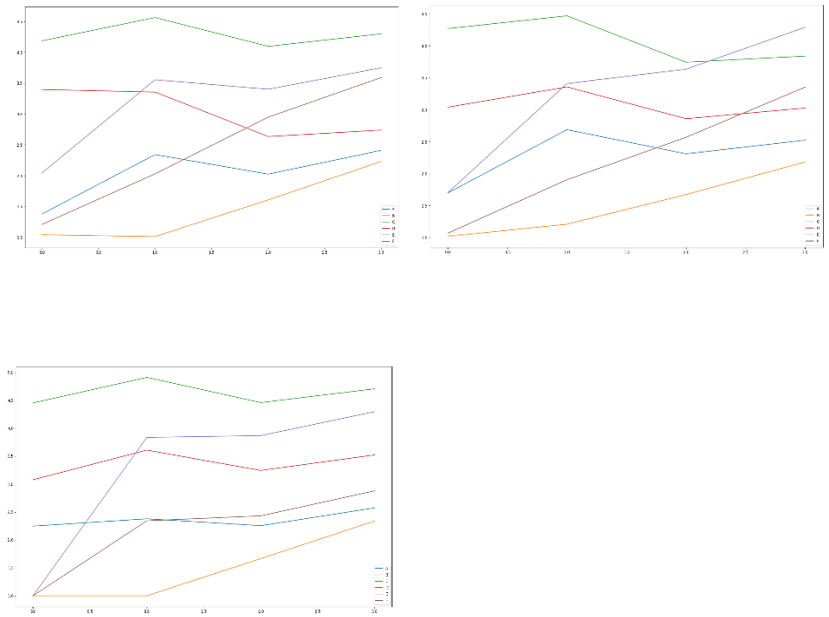


Fig. 3.3 Cluster Mean trajectories for each procedure (from left to right, autoencoder based clustering, kml, proc traj)

As shown in figure 3.3, the clusters mean trajectories obtained with the three procedures are similar enough to enable a potential cluster association within each clustering method. Although encouraging, this result alone only constitutes an empirical observation and cannot assert any satisfying conclusion on its own.

To establish quantifiable coherence between the posterior probabilities of an individual to belong to a given cluster within a given method, a spherical representation of the correlation matrix of these posterior probabilities is displayed in figure 3.4. It indicates that cluster membership probabilities among the three approaches are indeed correlated, thus ensuring stability within the investigated procedures.

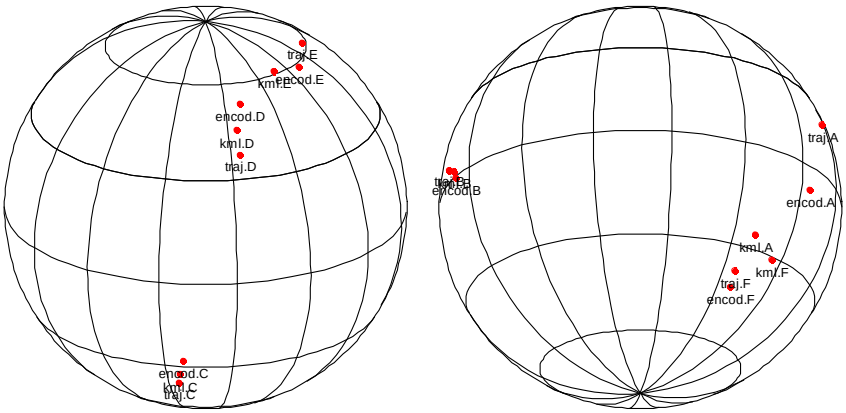


Fig. 3.4 Spherical projection of cluster belonging probabilities' correlation matrices. Traj, encod and kml points respectively link to clusters obtained with the proc traj, autoencoder and kml procedures. Letters link to visually similar clusters in the different approaches. For X = A, B, C, D or E encod.X, kml.X and traj.X are very close, this indicates that the clustering given by the 3 studied approaches are similar.

3.3.2 Simulated data application

3.3.2.1 Dataset

To illustrate situations where the autoencoder outperforms traditional clustering methods, the kml and autoencoder procedures were compared with each other on a simulated dataset corresponding to a hypothetical study of quality of life measurements amongst cancer patients. Patients are divided into 2 groups corresponding to 2 treatment allocations:

- Group A patients receive a treatment every 4 month. Their quality of life falls once treatment is administered because of its side effects, then increases because of treatment effects, then stabilizes because treatment is no longer efficient, then comes another sequence of treatment, thus leading to a sine shaped evolution of the patient's quality of life

  Group B patients receive continuous treatment, leading to stable life quality

The quality of life evaluation at time *t* was modeled such that:

$$QoL_A(t) = 5 \cdot \sin\left(\frac{\pi}{2} \cdot t + U(-2,2)\right) + 10 + N(0,1)$$

$$QoL_B(t) = 10 + N(0,1)$$

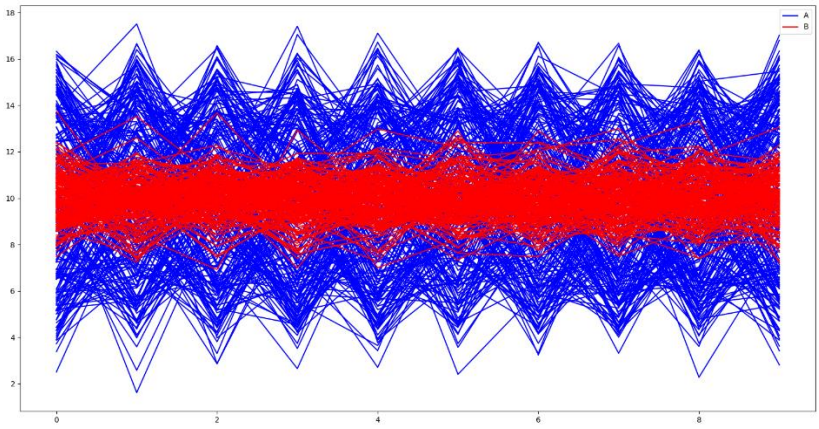


Fig. 3.5 Longitudinal plot of the simulated dataset. Patients in group A (in blue) have a sine shape evolution while patients from group B (in red) are stable.

3.3.2.2 Cluster confrontation

The methodology used to derive both clustering is as aforedescribed. For the kml procedure, the number of clusters was chosen to maximize the Calinski-Harabasz index. Proc traj was not used here because it is not designed to tackle periodic trajectories and so is not adapted to this type of data. As can be seen in fig. 10, the kml procedure fails to identify the two different groups and prefers it a partition of 7 clusters. However, it does manage to correctly group together individuals from the group B, and as such is still able to bring some insight about the dataset to the practitioner.

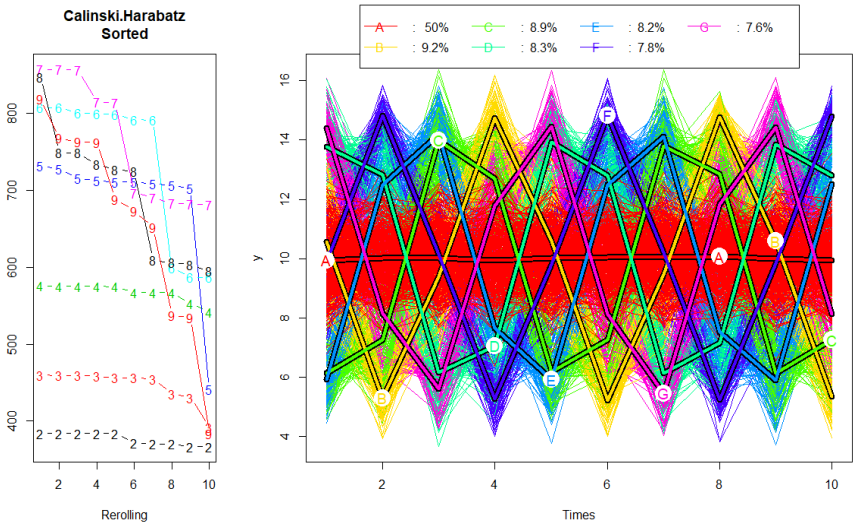


Fig. 3.6 Left: display of the Calinski-Harabasz indices for all ran clustering procedures. Right: Best extracted 7-partition, with mean cluster curve in bold and each individual subject grouped according to cluster belonging by color

Fig. 11 shows a plot of the autoencoder clustering procedure's extracted embedding. Groups A and B are clearly separated into two easily identifiable, non-spherical clusters. The said clusters can then be extracted through the use of different methods, such as spectral, or agglomerative clustering algorithms.

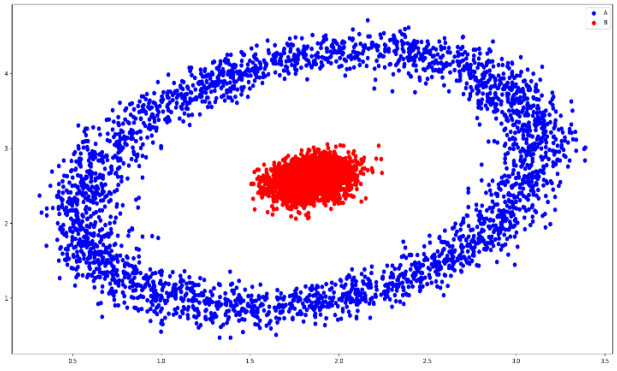


Fig .3.7. Bi-dimensional, atemporal embedding of the simulated dataset, with in blue embeddings corresponding to group A patients' curves, and in red embedding corresponding to group B.

Although the kml procedure managed to identify interesting patterns in this experiment, it was clearly outperformed by the autoencoder based approach. Indeed, the two subject groups crafted for the simulation are only correctly extracted from the bi-dimensional embedding provided by the recurrent autoencoder. One might object that K-means based procedure are not suited to the analysis of non-spherical clusters which the simulated dataset is clearly made of, and that other longitudinal clustering approaches might be better suited to the studied problem. However, with this experiment and the aforementioned one on the E3N dataset, the proposed autoencoder based approach was shown to perform correctly in both spherical and non-spherical clusters setups, thus proving its adaptability. It is also worth mentioning that the embedding itself, in this experiment, allows us to visualize useful information on the studied dataset. Indeed, the extracted non-spherical embedding might illustrate non-spherical relationships happening in the corresponding longitudinal dataset. Although not theoretically backed, this kind of insight can turn out to be useful, for instance by guiding the practitioner away from the use of spherical cluster based statistical methods such as K-Means or Gaussian mixture-based procedure, as well as cluster mean curve plotting as graphical results displays.

4 Voice analysis and automated detection of suicidal risks in crisis hotlines

4.1 Background

The previous section showed that artificial neural network can be used as statistical tools for the analysis of medical data in fields such as epidemiology or public health, where it can provide a significant improvement compared to traditional methods. This result is to be expected given these disciplines' strong statistical background. However, the current

enthusiasm for deep learning methods is mostly related to their ability to handle analysis problems that used to be intractable, such as voice, image or language modelling. As such, neural models can potentially provide medical practitioners with powerful technologies such as, for instance, decision aid algorithms based on a whole variety of complex, hierarchical data. Such applications are already emerging, typically in Radiology and Dermatology (3,4), where the power of convolutional networks for computer vision applications can be used to its full extent.

Surprisingly, the field of Psychiatry has been globally untouched by the raise of deep neural models. It should however constitute an area of choice for the development of such applications, as psychiatrists infer diagnosis mostly from discussions with their patients. As a consequence, the analysis of mental health patients' facial expressions, vocal patterns or language should, despite their significant complexity, contain information that might be recognizable by a powerful learning algorithm. Some experimental evidence actually supports that hypothesis (56), (57). However, most were derived from classical convex learning algorithm instead of the more recent and performant neural modelling techniques. It then only feels natural to investigate the potential use of deep neural networks in psychiatry, typically through a practical application. As an example, suicide crisis hotlines are useful tools in the management of patient under emotional and mental distress. However, crisis line counsellors can be exposed to difficult situation impairing their ability to help the caller. Consequently, giving counsellors objective, automated tools, with the ability to screen the caller's mental state, might help them in these situations.

This project focuses on the application of deep learning methods to the creation of such a tool by using artificial neural networks for voice analysis to model crisis hotline caller suicidal risk

4.2 Methodology and data acquisition

The definition and collection of a dataset is a crucial part of any experimental machine learning project. Indeed, the predictive power of any deep learning-based model is bound to the quality of the input and output variables' relationship. Although the amount of academic literature regarding voice analysis applications to suicidal intent or depression detection is limited, the inherent behaviour of machine learning algorithms provides with useful guidance in the design of an experiment. Most deep neural network based predictive models focus on finding the relationship between a set of input variable and an output variable. As a consequence, identifying potential candidates for both will subsequently define the whole experiment's methodology. The study being focused on suicide intent detection in crisis hotlines the choice of input variables is deterministically constrained to digital recordings of phone recordings from the latter. The definition of an explanatory variable, however, is not straightforward, and relates to deeply complex interrogations. Indeed, how does one define a suicidal risk?

4.2.1 Suicidal risk evaluation from crisis hotline recordings

The most straightforward way to derive an estimation of the suicidal risk is a frequentist approach. Indeed, by establishing whether a subject attempted suicide following his or her call to a hotline suicide, one can directly approximate the conditional probability of a given

subject to attempt suicide given the content of the corresponding recording, which is exactly the definition of a suicidal risk. However, this solution presents two major drawbacks:

- Suicide attempts are relatively rare events. As a consequence, gathering enough observations of both outcome can become prohibitive
- This information is typically not available. Indeed, crisis hotlines are not necessarily in constant relation with hospitals, and a significant number of callers remain anonymous

A possible solution to this problem is to use experienced psychiatrists as proxy for estimating this probability. Several criteria are available to evaluate a given subject's suicidal intent, among which the Clinical Global Impression – Severity Scale (CGI-S) which was chosen for this study. The CGI-S is a 7 levels scale requiring a clinician to rate the severity of the patient's illness, relative to clinician's experience with patients who have the same diagnosis according to the following scale (58):

1. Normal, not at all ill
2. Borderline mentally ill
3. Mildly ill
4. Moderately ill
5. Markedly ill
6. Severely ill
7. Among the most extremely ill patients

This criterion was decided upon jointly with a psychiatrist with expertise in psycho-traumatology and experience in suicide crisis lines, for four main reasons:

- The criterion doesn't require any additional information to be assessed (no psychometric scale, for instance)
- The criterion allows for the practitioner to express a clinical opinion potentially much more powerful than the information gathered in standardized forms
- The criterion is relatively easy to assess for an experienced practitioner
- The criterion can be assessed in a whole range of situations

Once selection criteria has been chosen, a methodology must be derived to define how the practitioner is going to label the data.

A straightforward method would be to evaluate each conversation's overall situation with a CGI-S. However, as a crisis line's phone conversation lasts for half an hour in average, the information contained into each evaluated sample would be too important to allow for the model to learn any meaningful information from the provided examples.

Consequently, it was decided to slice every recording into fixed length samples, for which evaluation will be performed sequentially during the listening.

As an example, a 30 minutes' phone recording would be divided into 60, 30 seconds length recordings. A psychiatrist will then listen to all the 60 samples in order and provide a CGI-S evaluation for each sample. However, this evaluation methodology leads to one issue. Indeed, forcing the practitioner to label a small sample of recording can lead to problem, such as, for instance, a 30 seconds slice of recording where the hotline caller is not talking. Whether the caller is not talking since the counsellor is speaking, or due to another reason, this type of situation leads to the practitioner being unable to correctly evaluate the audio sample.

The recording slice's length was established following two criteria.

- The slices cannot be over a minute and a half, otherwise the data's sequence length becomes too long for neural networks to capture long term dependencies
- 4 psychiatrists listened to several slices of different lengths (from 15 seconds to a minute and a half, with a 15 second increment), and chose unanimously the 30 seconds slice to be the most convenient one (enough information for evaluation, not too long so the caller's state can change)

Therefore, the final evaluation methodology was defined as follows:

- The practitioner listens to a phone recording, 30 seconds at a time
- For every 30 seconds of recording, the practitioner evaluates two different CGI-S indices
- The first CGI-S corresponds to the practitioner's evaluation of the situation in its globality, and is supposed to answer the question "How is the caller feeling at the time of the recording", given the whole conversation
- The second CGI-S corresponds to the practitioner's evaluation of the situation without the past context, and is supposed to answer the question: "does this recording of 30 seconds of audio contain markers of suicidal behaviour"

Following the definition of the methodology, SOS Kriz, a French Caribbean based suicide hotline was contacted and agreed to provide recordings. An authorization by the French National Commission on Informatics and Liberty (CNIL) to record the crisis hotline's recordings for research purpose was obtained, and an automated recording device was installed in SOS Kriz to record around 200 hours of crisis hotline phone conversations.

In order to evaluate the relevance of this evaluation methodology, an inter-rater agreement study was conducted with the help of 4 psychiatrists using the following methodology:

- 15 completely independent slices of 30 seconds of crisis hotline recordings were submitted to each practitioner
- The recording slices were taken at random times from random recordings. Each psychiatrist had the same 15 slices.
- Each psychiatrist was to quote them with the CGI-S criterion
- Interrater agreement was then evaluated though computation of a quadratic Cohen Kappa test (59) to account for the ordinal nature of the evaluation

	Judge 1	Judge 2	Judge 3	Judge 4
Judge 1	1	0.69	0.65	0.50
Judge 2	0.69	1	0.71	0.79
Judge 3	0.65	0.71	1	0.61
Judge 4	0.50	0.79	0.61	1

Table 4.1 Inter rater agreements measured though a quadratic Cohen Kappa score. Values range from moderate to strong agreement

As can be seen in Table 4.1, the inter-rater agreement values range between 0.5 and 0.8 with a mean value of 0.65. As no consensus exists on the interpretation of the Kappa statistic, assessing these results is not straightforward. One possible approach is to consider values according to the following table, proposed by (60) :

K value	Interpretation
<0	No agreement
[0, 0.20]	Very weak agreement
[0.21, 0.40]	Weak agreement

[0.41, 0.60]	Moderate agreement
[0.61, 0.80]	Strong agreement
[0.81, 1.0]	Agreement almost perfect

Table 4.2 Interpretation of the Cohen Kappa statistic according to (60)

According to this table, the data collected during the experiment indicates that the psychiatrists strongly agree on average, with some variability in agreement from one psychiatrist to another.

Additionally, an intra rater agreement experiment was attempted. The 4 psychiatrists were expected to evaluate twice 20 randomly selected 30 seconds audio files sampled from crisis hotline recordings, with a 4 months gap between the first and second evaluation. However, due to lack of availability only one psychiatrist was able to perform the experiment, which lead to a Cohen Kappa value of 0.79, showing strong stability of evaluation across time.

Following this preliminary study, around 40 hours of recording were randomly chosen and distributed to the 4 psychiatrists, who each evaluated around 10 hours of crisis hotline recording following the aforedefined methodology.

The problem of inferring suicidal risk from crisis hotline recordings can be defined from the resulting dataset as a two-fold predictive modelling problem:

- A first model predicting the CGI-S evaluation on individual 30 seconds slices of audio recording, from said slices of audio recording. As such, this problem will in the following sections be called the local evaluation model
- A second model predicting the global CGI-S evaluation from the local evaluation

The combination of these two models should allow in real-life use of the technology to provide the counsellor with the global CGI-S evaluation from the audio recordings in real time.

Finally, the experimental set up requires a performance metric to guide the architecture search process and evaluate the final model's performance. Given the application's requirement, a dual criteria performance metric will be used to guide the model selection process:

- To remain coherent with the preliminary experiments, the model's prediction performance will be assessed with a quadratic Cohen Kappa coefficient measuring agreement between the model's predictions and the ground truth. This performance metric is the standard approach for performance evaluation of ordinal regression models in the machine learning academic community (61)
- As the main purpose of the experiment is to assess the possibility of providing crisis counsellors with real-time information of the caller's mental state, the computational cost of the whole data processing pipeline, from raw audio to prediction, should be sufficiently low to allow for real-time processing. In practice, this criterion can be assimilated to limiting the decision process' entire computational cost to 30 seconds

4.3 Data preprocessing

In most cases, real data is not directly fit for machine learning based analysis. Either noise from the gathering process or intractable data format may make it impossible to inject observations directly into a learning algorithm. As a consequence, a preprocessing step is generally applied to the investigated dataset in order to transform it into a vectorized, machine learning compatible format. This general rule stays true to the investigated

modelling problem. Indeed, as collected from the crisis hotline, the gathered dataset is unfit for machine learning applications for two main reasons:

- The recordings themselves are not clean. Available audio tracks record both the caller and the counsellor, which can confuse the learning algorithm. In addition, recordings have some artifacts such as audio clicks due to the telecommunication technologies
- The slices of 30 seconds audio are too big to allow for computationally efficient analysis

As a consequence, a two-step preprocessing pipeline was applied to the recordings before any analysis. First, recordings were manually cleaned with a simple audio processing software, Audacity, and second, a range of voice features were extracted from the clean recordings to convert them into a more computationally tractable format.

4.3.1 Manual preprocessing with the Audacity software

Due to budget constraints, the phone recording device used for the recording of the crisis hotline conversations only provides a monophonic signal mixing both the caller and the counsellor's voice. Using the mixed signal directly to adjust a model would require of the algorithm to implicitly identify whose voice is the caller's, which seems highly unreasonable. As a consequence, the counsellor's voice was manually edited out of all recordings to provide recordings with only the caller's voice, using an open-source software specialized in audio file processing called Audacity (62).

The Audacity software also provides users with basic built-in digital sound processing functions such as click suppression and noise reduction, which were both used on the recordings to improve the audio recording's quality.

4.3.2 Signal processing-based reduction of dimensionality

The obtaining of clean recordings is followed by a decision with significant impact on the final model's predictive power. Indeed, three different approaches are available in the analysis of speech:

- Analysis of the speaker'swords. The model then focuses on what the caller is saying without including the acoustic information. This modelling problem then belongs to the field of natural language processing
- Analysis of the speaker's voice. The model focuses on how the caller is speaking, without taking into account the content of its speech. This modelling problem belongs to the field of audio analysis
- Analysis of both the caller's voice and verbatim. This modelling problem is a combination of the previous two and belongs to the field of multimodal analysis

Clearly, the last option is preferable, as it does not hide any information from the recordings to the prediction models. However, in this experiment's setting, the extraction of the caller's meaning is an extremely time-consuming process. Indeed, although highly performant speech-to-text algorithm are available nowadays for the automated transcription of audio recordings, the recordings were obtained from a French Caribbean crisis hotline in the island of Martinique. As a consequence, some recordings were spoken in Créole, a French dialect endemic to the island's inhabitant, and for which no speech recognition technology is available. In addition, French-speaking Martinique inhabitants tend to have a strong, unusual accent for which no specific voice model has ever been fit, which can lead to catastrophic failure of automated transcription technologies. Consequently, the caller's verbatim would require a manual transcription, at least for the 40 hours of evaluated audio recordings, if not

for the complete 200 hours dataset. Due to the extremely time-consuming nature of such a process, it was deemed incompatible with the project's timetable. It should however be the object of future work. As a consequence, the analysis of the crisis recordings will in this experiment be considered only as a voice analysis problem.

Theoretically, the cleaned recordings could be used directly for the implementation of a deep neural network-based regression analysis on sequential datasets, which was previously defined as the current modelling problem. In practice, deep learning models are not yet adapted to the direct analysis of voice signal in most cases. Indeed, 30 seconds of mono audio recorded at 44000Hz are digitally encoded into a univariate sequence of length bigger than a million which leads to two major difficulties:

- To this day, no hardware is capable of computing deep neural net fast enough to perform real time analysis on problems of that scales. As an example, Deepmind, a Google company specialized in deep learning research recently introduced a neural network dedicated to the direct analysis of audio signal without any preprocessing step (26). Running their algorithm on a few seconds of audio on their highly specialized hardware took up to several hours! In particular, the sequential nature of audio makes it hard to use parallel processing, which is the most widely used technique to speed up computation time in neural network-based algorithms
- The problem is highly dimensional. Even though they are locally correlated, the input variables are too numerous, and collecting a big enough dataset to avoid overfitting would be extremely expensive, if not unrealistic

As a consequence, most machine learning audio analysis-based technologies use a range of computationally efficient signal processing based techniques to significantly reduce the

dimensionality of the modelling problem in order to make it tractable with the hardware resources available to this day.

4.3.2.1 Windowing

Voice signals are locally periodic pressure oscillations of the air surrounding created by complex, synchronized movements of both the speaker's mouth, throat and vocal cord. The inherent periodical character of this type of signal allow for the use of powerful analysis tools in order to express the problem in a simplified form without losing much information. However, as the speaker modulates his or her vocal organs to create the different sounds required for the utterance of a sentence, the audio signal loses its periodicity. As a consequence, the recording needs to be examined at relatively short time scales (typically around 50 milliseconds) in order to be able to make use of the mathematical tools available for the analysis of periodical signals. Most voice analysis-based technologies thus examine small overlapping fixed-length windows of audio signal independently, under the hypothesis that the signal is periodical within them. This approach is called windowing, and is typically performed on a discrete signal s of length T as follows:

- A first window of typical length 50 milliseconds is taken from the beginning of the audio recording
- Audio features are extracted from this subsample of audio, under the assumption that the signal on that timescale is approximately periodical
- The process is iterated across the whole audio recording, by shifting the window at each timescale by a fixed length, typically 25 milliseconds, thus allowing a 50% overlapping between windows

The interest of windowing-based feature extraction can be assessed instantly. Indeed, considering the problem of analyzing audio recordings of 30 seconds length sampled at 44000Hz, the dimensionality of the problem evolves from:

- One quantitative variable observed sequentially 1,320,000 times
- N quantitative variables (as many as there are features extracted per windows) observed sequentially 1200 times

As a consequence, the dimensionality of the sequential analysis problem is considerably reduced, at the cost of introducing multivariate sequential observations.

Although quite efficient in reducing the dimensionality of a sequential statistical analysis problem, windowing presents an important drawback by introducing border effects. Indeed, by selecting only windows of signal, the latter is, from the window's point of view, abruptly cut at its border, which introduces unwanted, artificial harmonics in the signal, and can consequently impair the feature extraction process' quality.

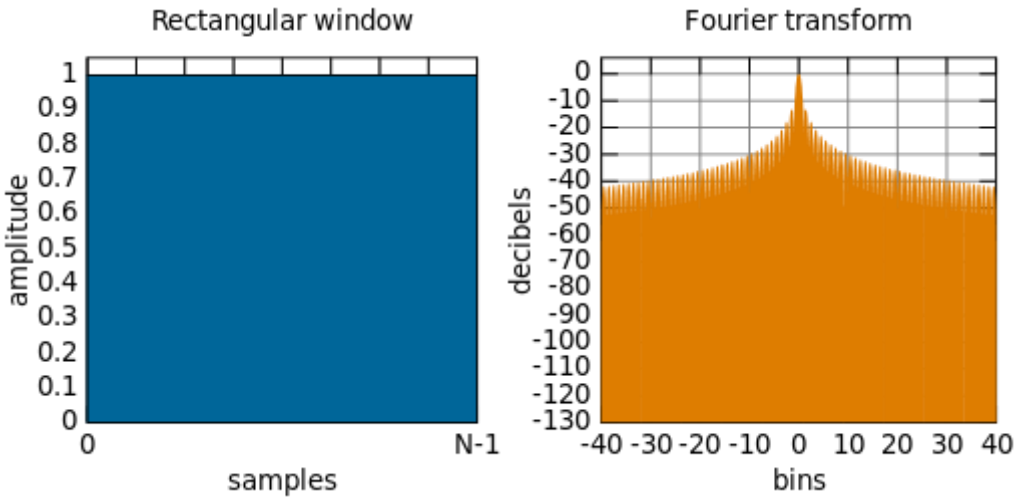


Fig. 4.1 Frequency response of a simple, rectangular window. Spectral noise of intensity around -40dB is added across all frequency, with a concentration around 0 (63)

In order to counteract these undesirable side effects, the signal subsamples extracted with a traditional windowing process are typically multiplied with a window function. Several window functions are available to the signal processing practitioner to choose from depending on cases, the most widely used being the Hamming window function, defined as follows for a signal s of length T :

$$\forall t \in [\![0, T-1]\!], \qquad w(t) = 0.53836 - (1 - 0.53836) \,.\, cos\left(\frac{2\pi t}{T-1}\right)$$

The subsequent windowed signal is then obtained by element-wise multiplication:

$$\forall t \in [\![0, T-1]\!], \qquad s_{win}(t) = w(t) \,.\, s(t)$$

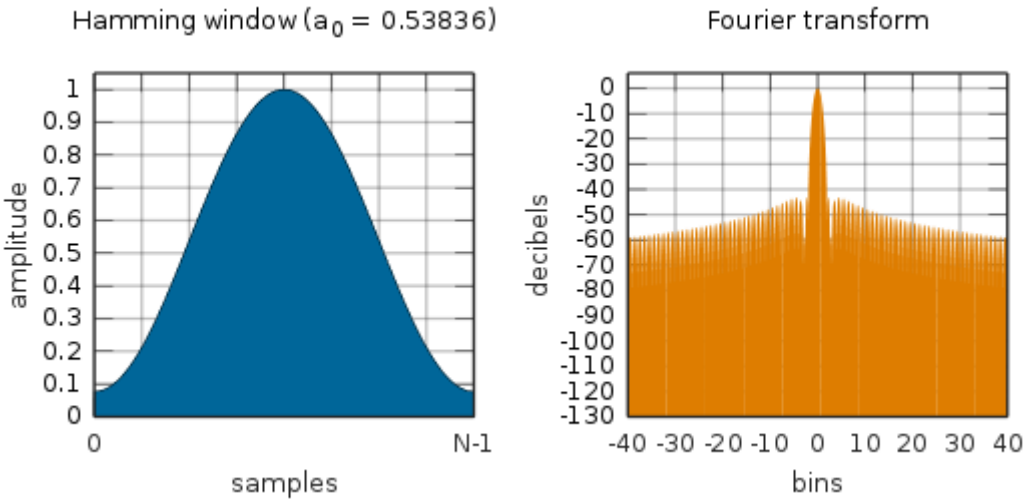


Fig. 4.2 Frequency response of a Hamming window. The spectral noise is sensibly lower with an intensity of -55dB, especially around 0 (64)

4.3.2.2 Time domain features

Time domain features are characteristics of the signal that are directly derived from the signal itself considered as a discrete function of time, as opposed to analysis tools exploiting more complex representation of the signal, such as its Fourier transform, for instance.

4.3.2.2.1 Pitch and fundamental frequency

A speaker's pitch is information of utmost importance when designing voice analysis-based systems. First, an extremely desirable property of such application is speaker invariability. A model showing better performance on male speaker compared to female speaker would indeed be problematic. Consequently, adjusting a predictive model while confounding on the speaker's voice pitch allows for models that are more robust. In addition, the speaker's pitch evolution across time as the conversation unfolds might contain information that, by itself, could boost the model's predictive power.

On a physical point of view, a speaker's pitch can be assimilated to the signal's local fundamental frequency. Several methods are available to the practitioner to extract a periodical signal's fundamental frequency, all based on computational heuristics rather than truly theoretically supported. One simple and efficient extraction method can be derived from the signal's auto-correlation (65). The autocorrelation of a signal, discrete or continuous, is defined as the cross correlation of said signal (as defined in 2.2.7.1 with convolution networks) with itself, and can be written formally for a discrete signal s of length T such that:

$$\forall \tau \in [\![1, T]\!], \qquad ACF_s(\tau) = \sum_{t=1}^{T} s(t).s(t-\tau)$$

The autocorrelation function of periodical signal s presents an interesting property: Its peaks are found at the harmonic periods of s. The fundamental period (and thus fundamental

frequency) can then be empirically extracted by selecting the autocorrelation's argument of the maxima in a predefined interval of periods.

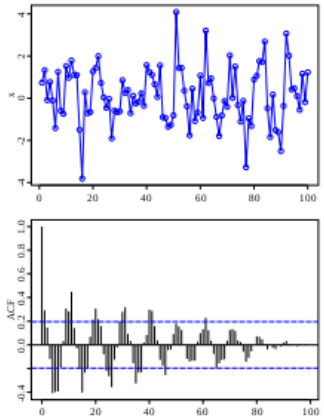


Fig. 4.3 Sine function corrupted by additive random noise (top) and its autocorrelation function (bottom). The second peak (the autocorrelation function's global maximum is always 0) is located at the function's fundamental period

4.3.2.2.2 Zero crossing rate

Although the zero-crossing rate might be one of the simplest feature to derive from audio signals, it has been identified as a useful indicator in several tasks related to automated mental disorder diagnosis from voice analysis (66). It is, defined as a signal's frequency to change from negative to positive values and conversely (67). For a discrete signal s of length T, the zero-crossing rate can be formally defined as:

$$zcr(s) = \frac{1}{T-1}\sum_{t=1}^{T-1} 1_{\mathbb{R}^-}(s(t) - s(t+1))$$

4.3.2.2.3 Energy

The energy of a given signal is a well-known concept in signal processing. It is defined, for a discrete signal s of length T as its squared amplitude sum (67):

$$E(s) = \sum_{t=1}^{T} s(t)^2$$

4.3.2.2.4 Loudness

Although quite straightforward to define, the aforementioned notion of energy is one based on purely physical and mathematical considerations and does not account for how human actually perceive a signal as energetic (or loud). Although the computation of a signal's loudness can get quite computationally expensive, a simple approximation of this concept can be derived in the following fashion for a signal s (67):

$$E_{i,approx}(s) = \left(\frac{E(s)}{E_0}\right)^{0.3}$$

With:

- $E(s)$ the energy of signal s
- E_0 a reference intensity (defined as $E_0 = 10^{-6}$) in the Opensmile library (68)

4.3.2.2.5 Linear predictive coding coefficients

Linear predictive coding coefficients are a powerful tool in speech analysis (69) that uses the coefficient of an autoregressive linear predictive model fit on the investigated signal as features for subsequent analysis. As a consequence, the number of features obtained a given signal through this method can vary considerably (with the depth of the autoregressive model) but is fixed to 7 in the Opensmile library. Formally, for a discrete signal s of length T, the linear predictive coding coefficients can be derived the following way (67):

- Fit the following linear model to the signal:

$$\forall t \in [\![8, T]\!] \quad s(t) \approx \hat{s}(t) = \sum_{i=1}^{7} a_i \, s(t-i)$$

- Use the adjusted coefficients a_i as features for the signal

In addition to the linear predictive coefficients, it is common practice to adjust predictive models on their first order temporal derivative, whose computation for a discrete signal s of length T is straightforward:

$$\forall t \in [\![1, T-1]\!], \qquad \Delta s(t) = \frac{s(t+1) - s(t)}{\delta t}$$

$$with\ \delta t\ the\ true\ time\ difference\ between\ s(t+1) and\ s(t)$$

4.3.2.3 Frequency domain features

As aforementioned, voice signals can be considered as locally periodic oscillatory signals. Under the hypothesis that the windowing functions used during the feature extraction process are small enough, the investigated audio signal can be considered as approximatively periodical. The analysis of periodic functions is a well-developed field which, through the development of Fourier analysis and its Fourier transform, has led to the development of a whole family of analysis tools for periodical signals currently used in most voice analysis-based technologies such as voice to text algorithms.

4.3.2.3.1 Fourier transform

The Fourier transform is the foundation of most, if not all frequency-based analysis tools available in signal processing, which expresses a time dependent function of real values as a continuous sum of simple, periodic functions (namely the sine and cosine function) with different phases and frequencies. For a continuous, real-valued function f, its Fourier transform $\hat{f}$ is defined as

$$\forall \xi \in \mathbb{R}, \qquad \hat{f}(\xi) = \int_{-\infty}^{+\infty} f(x) e^{-2\pi i x \xi} dx$$

Although mathematically accurate, this expression of the Fourier transform presents one major computational challenge, in the sense that its applications are limited to the study of continuous signals. As digital signals are by essence discrete functions, an adaptation of this tool is required, and can be found in the discrete Fourier transform, which is defined for a signal s of length T as follows:

$$\forall k \in [\![0, N-1]\!], \qquad \hat{s}(k) = \sum_{t=1}^{T} s(t) \cdot e^{\frac{-i2\pi kt}{T}}$$

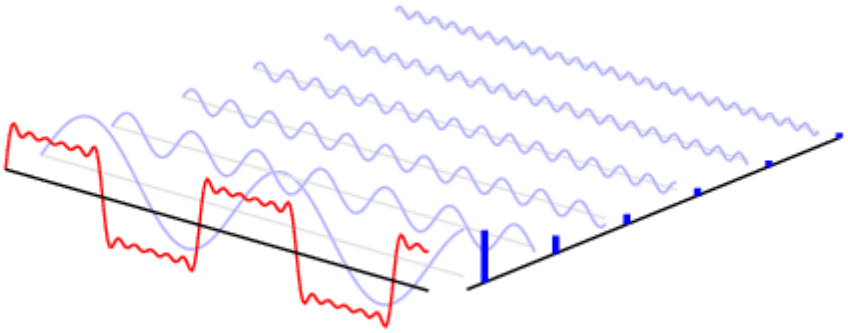


Fig. 4.4 Example of Fourier transform (in blue) of a time dependent function (in red). Because the time dependent function is periodical, its Fourier decomposition is discrete.

This exact expression being typically quite expansive to compute, most signal processing applications compute an approximation of it through a powerful algorithm called Fast Fourier Transform (FFT) (70).

By definition, the Fourier transform of a given function is complex-valued, which is not typically used in statistical analysis or signal processing. As a consequence, it is common practice to transform it into a real valued based function any analysis, typically by computing its amplitude:

$$\forall x \in \mathbb{C}, |x| = \sqrt{Im(x)^2 + Re(x)^2}$$

This final signal obtained by computing the amplitude of the discrete Fourier Transform is called the power spectrum.

4.3.2.3.2 Mel frequency cepstral coefficients

Although quite powerful in describing efficiently most audio signals observed in the world, it presents some problems when tackling psycho-acoustic based problems. Indeed, the Fourier transform does not discriminate between different frequencies. The Human ear, in the other hand, is typically more sensitive to frequencies below 1000Hz and its response to audio stimuli is highly non-linear (71). As a consequence, voice analysis-based applications do not directly use the power spectrum, but rather apply additional transformation to it, in order to derive representations of the signal that are tailored for psychoacoustic tasks such as voice recognition. The most widely used transformation in voice analysis is called Mel-filtering and leads to the extraction of what are called the Mel frequency cepstral coefficients, which are used, for instance, as the main vocal features in the Google Voice application of speech recognition. These powerful vocal features can be derived from an audio signal s of length T through a 5-step process:

- Compute the discrete Fourier transform $\hat{s}$ of s, typically with the Fast Fourier Transform algorithm for computational efficiency
- Multiply the signal's power spectrum with all filters of the Mel-filter bank. The Mel-filter bank are triangular shaped filters as can be seen in figure 4.5 that are designed to map the Fourier transform into a more psycho-acoustically relevant representation. This transformation will yield as many real valued signals as there are filters in the Mel-filter bank, each of same length as $\hat{s}$

- Take the discrete cosine transform of the resulting signals' log. The discrete cosine transform is a signal processing tool closely related to the discrete Fourier Transform, which outputs K real numbers from a signal s of length T, with K typically chosen between 12 and 16 (12 in the present application) and is formally defined as follows:

$$\forall k \in [\![1, K]\!], \qquad C^{(mel)}(k) = \sqrt{\frac{2}{T}} \cdot \sum_{t=0}^{T-1} s(t) \cdot \cos\left(\frac{\pi}{T}\left(t + \frac{1}{2}\right) \cdot k\right)$$

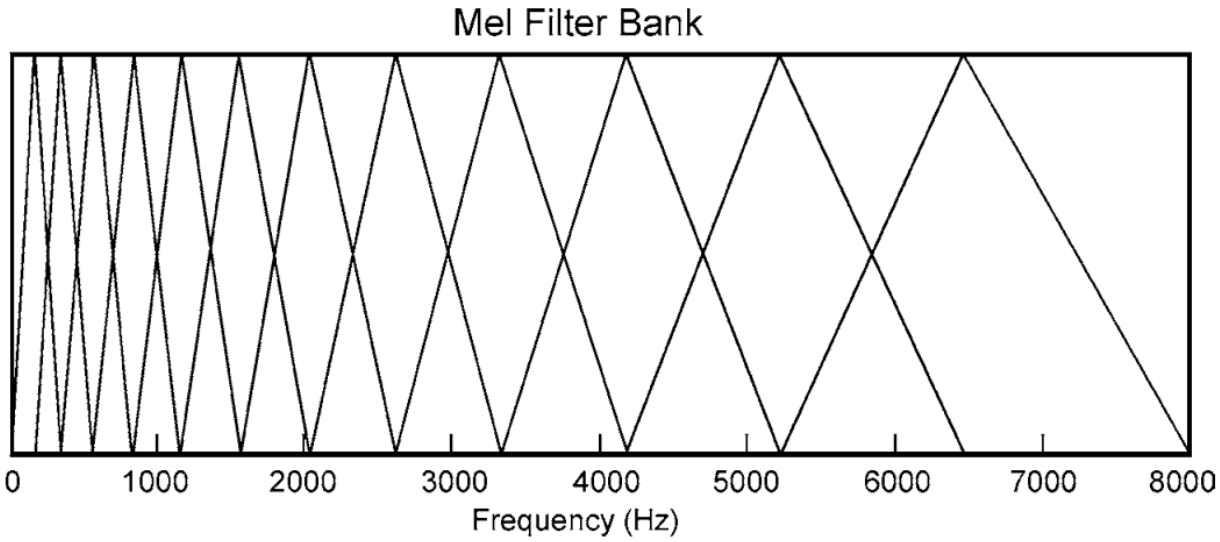


Fig. 4.5 Example of Mel-Filter Bank. These 12 triangular shapes are multiplied with the signal's Fourier transform to obtain the Mel-frequency cepstral coefficients (72)

Similar to linear predictive coefficients, it is common practice to adjust predictive models on the Mel-frequency cepstral coefficients' first and second order temporal derivatives (also called speed and acceleration), whose computation for a discrete signal s of length T is straightforward:

$$\forall t \in [\![1, T-1]\!], \qquad \Delta s(t) = \frac{s(t+1) - s(t)}{\delta t}$$

$$with\ \delta t\ the\ true\ time\ difference\ between\ \ s(t+1) and\ s(t)$$

Following this preprocessing process of data cleaning and windowing-based feature extraction, every 30 seconds slice of audio has been converted into a sequence of 54 variables, observed sequentially 1200 times:

- Fundamental frequency, 1 variable
- Zero crossing rate, 1 variable
- Energy, 1 variable
- Loudness, 1 variable
- Linear predictive coefficients, 7 variables
- Linear predictive coefficients first order derivatives, 7 variables
- Mel frequency cepstral coefficients, 12 variables
- Mel frequency cepstral coefficients first and second order derivatives, 24 variables

All of these features were computed on the investigated dataset using the Opensmile library, which is a high performance, multicore enabled library specialized in the extraction of voice audio features for emotion analysis. All features were then normalized following the input normalization procedure described in section 2.3.4.1.

Although these features are the most commonly used in voice analysis-based technologies, the field of signal processing is extremely rich and whole other families of feature extraction procedure for voice signals are available, such as spline regression or wavelet-based decompositions (73). They were however discarded from this experiment due to their prohibitive computational cost. Indeed, pre-built libraries specialized in the extraction of such

features take up to 16 seconds to process 1 second of audio on a high-performance computer, which presents two problems:

- In order to perform real time analysis on sequential data, the processing time of n seconds of audio needs to be at most equal to n, anything bigger leading to diverging latencies
- With the computational resources available for the experiment, the total extraction time of these features, assuming uninterrupted computation, on the entire dataset, would amount to more than 3 months, which was not compatible with the time available for the completion of this thesis

4.4 Analysis

After preprocessing, the data analysis problem has become more tractable, and neural network can be fit to the dataset. In order to restrain the problem's dimensionality, the methodology derived for the data acquisition was designed to partition the complete modelling process into two separate sub models:

- A first predictive model predicting the local evaluation (from only 30 seconds of audio) from the features directly extracted from the corresponding audio recordings
- A second predictive model, predicting the contextual evaluation (from everything happening so far in the recording) from the local evaluation

The first model, although more complex with regard to the phenomenon it investigates, is actually a simple case of regression analysis on longitudinal data, as has been described previously in section 2.2.6 for instance. Indeed, only the input is of longitudinal nature, the output being a single variable.

The second problem, however, is not as straightforward on a mathematical point of view. It requires indeed to predict a sequence of output variables from a sequence of input variables. This type of modelling problem is known as a sequence-to-sequence model and can be performed with both recurrent and convolutional neural networks (74).

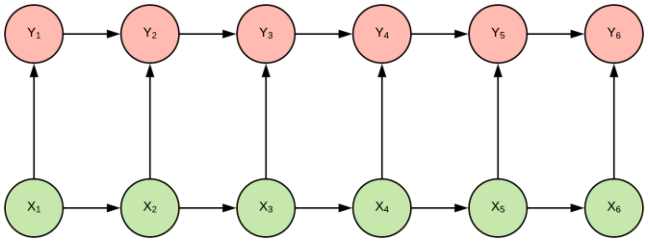


Fig. 4.6 Example of sequence to sequence modelling problem representing the second predictive model. This specific problem has an advantage over the traditional sequence-to-sequence modelling task. Indeed, both the input and output sequences have the same length, which is not necessarily the case, as can be observed in voice to text or machine translation algorithms

Finally, a specificity common to both modelling problems is the ordinal nature of their output variables. Ordinal variables are usually considered as standing between qualitative and quantitative variables and are typically defined as categorical variables that present an order relationship (in the case of this study and the CGI-S evaluation, 3 is typically less concerning than 7). This additional information is quite powerful and needs to be incorporated into the modelling process. It is however not straightforward, and there is no widely approved method of ordinal regression as of this day, especially in the machine learning community. In addition to traditional methods, two approaches specialized in the modelling of ordinal variables with deep neural networks were selected for this study, the first one being the actual state of the art in deep ordinal regression models, and the second of the author's making, that is shown to outperforms all other methods on a simple simulated dataset.

4.4.1 Ordinal regression models

4.4.1.1 Deep ordinal regression with unimodal probability distribution

Regression with unimodal distribution is a recently introduced method of ordinal analysis specifically designed for deep neural network related applications (61). Its rationale is derived from the observation that, when modelling ordinal quantities, misclassification of the target variable are not all equal. Outputting a state of 2 when the ground truth is 1 is clearly better than outputting a state of 7 for the same ground truth. However, there is no way to include this prior knowledge within the adjustment of a standard logistic regression. One approach to overcome this difficulty is to enforce unimodality of the model's output probability distribution. As such, during the optimization of a cross entropy objective, penalties due to misclassification will implicitly account for the distance between ground truth and estimation, without the use of a least square cost function.

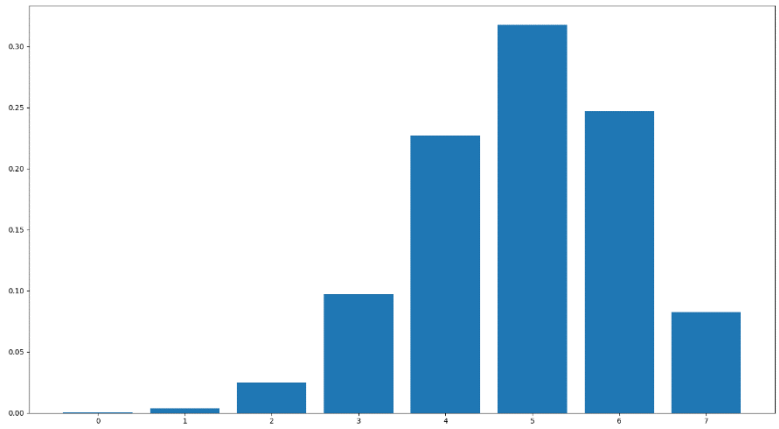


Fig. 4.7 Example of a unimodal probability distribution, the Bernoulli distribution. Probabilities decrease as they get farther away from the mode, which is highly desirable in the context of ordinal regression

The authors of this ordinal regression method decided to enforce unimodality by, instead of having a neural network directly build the output probabilities, have it output parameters for

probability distributions well known for their unimodality, as, for instance, the Poisson and Binomial distributions. As the original paper introducing this ordinal regression method reported from experiments that the Bernoulli based approach yielded significantly better performances than other approaches, the other methods based on different probability distributions were discarded for the present experiment.

Deep unimodal ordinal regression analysis of an ordinal variable y with s states from n explanatory variables $(x_i)_{i\in[\![1,n]\!]}$ can be implemented as follows:

- Design a neural network taking as inputs the explanatory variables and output a scalar value $p \in [0,1]$ typically ensured by the application of a sigmoid function
- Use the previously defined p value as the parameter for a binomial distribution of s trials defining the probabilities of each of the ordinal variable's states:

$$P(k;K-1,p) = \binom{K-1}{k} p^k (1-p)^{k-1-k}$$

- Adjust the whole model through cross entropy optimization with backpropagation and gradient descent

4.4.1.2 Deep ordinal regression with binary search and recurrent neural networks

Most ordinal regression methods, such as the one aforementioned or the ordered logit model tend to require either the fine tuning of some extra hyperparameter or to satisfy some hypothesis such as the proportional odds assumption (75). Reduced to its essence, ordinal regression can be considered as a search problem within an ordered set of values. Indeed, the model is trying to predict the correct output value, according to the explanatory variables, and the information available to quantify its performance is based on comparisons (is the model's output bigger or lower than the ground truth?).

A binary search is a powerful recursive algorithm that, from a sorted array, determines the position of one of its given element, by computing a logarithmic amount of comparison between the investigated value and the table's elements:

- Select the median element of the table, and compare it to the investigated value
- If the median element is bigger than the investigated value, repeat the algorithm applied to the table's lower half
- If the median element is lower than the investigated value, repeat the algorithm applied to the table's bigger half
- If the median element is equal to the investigated value, stop the algorithm and return the median element's position

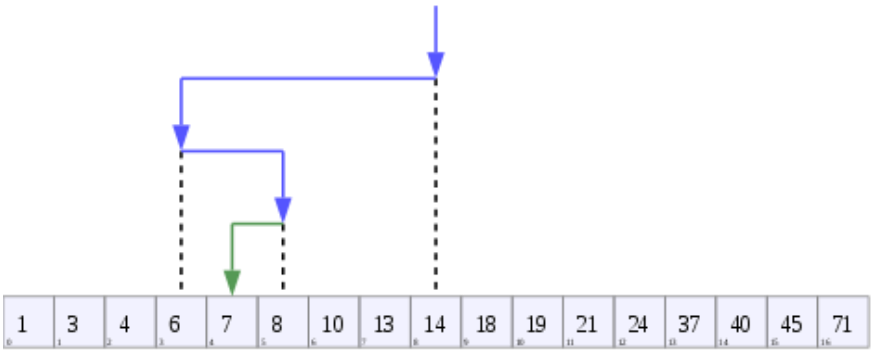


Fig.4.8 Example of a binary search algorithm (source: Wikipedia). The algorithm needs only 3 comparisons to find the position of an element in a table of 16 elements

As a powerful tool to manipulate ordered sets with nothing more than comparisons between values, the binary search algorithm constitutes an interesting basis for the design of a method for ordinal regression.

Instead of directly trying to model the investigated ordinal variable, the proposed approach focuses on learning a binary search process to identify its correct state, given the input

variables. Essentially, the investigated ordinal output variable of n states is converted into $log_2(n)$ binary variables, encoding every one of the variable's state position on a binary search tree, as can be seen on figure 4.9, for an ordinal variable with 8 different states.

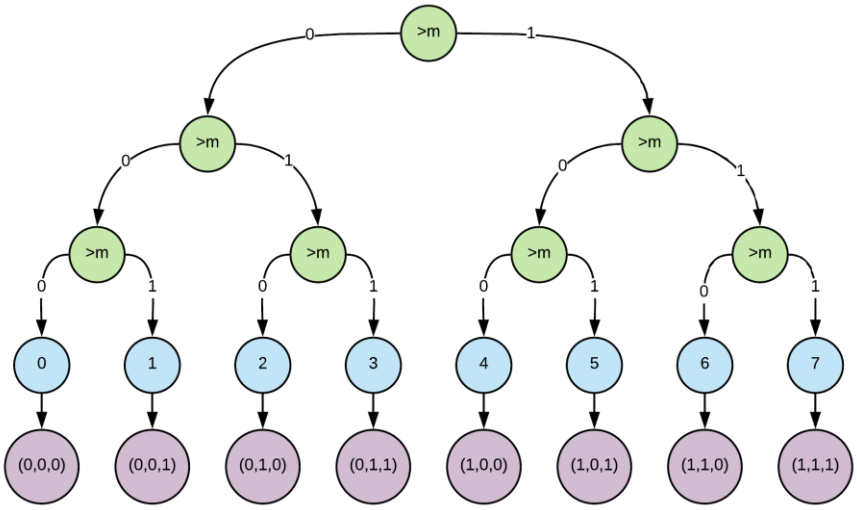


Fig. 4.9 Binary search tree. Each state of the ordinal variable is associated with a vector of binary variables representing its location on the tree, and the result of the equivalent binary search process

In order to fully understand the relationship between the regression analysis of an investigated ordinal variable y and the modelling of its corresponding aforedefined set of binary variables:

- The first binary variable can be interpreted as the logical condition 'the current output state is located in the top half of the variable's states' and estimates the probability $P(y \geq 4)$
- The second binary variable can be interpreted as the logical condition 'the current output state is located in the top half of the variable's states subset defined by the previous condition' and estimates the probability $P((y \geq 2|V < 4) \cup (y \geq 6|y \geq 4))$

- The third binary variable can be interpreted similarly to the second one, in a recursive manner

The probabilistic interpretation of the proposed binary variables clearly shows that this approach to ordinal modelling makes only one assumption about the output variable, that its states are ordered, which is the definition of an ordinal variable. However, managing to properly model this encoding is not straightforward. Although the first binary variable can easily be estimated through standard logistic regression, the following variables are all built on conditional probabilities, and as such depends on all the previous binary variables, which prevents them from being directly modelled through logistic regression. However, recurrent neural networks were previously introduced as tools to model decision processes where past decisions impact the current ones, and they make as such an ideal tool to model the subsequent binary variables. As a consequence, the modelling of an ordinal variable y with s states by n explanatory variables $(x_i)_{i\in[\![1,n]\!]}$ can be visualized in fig. 4.10 and described as follows:

- Create a quantitative variable $\hat{y}$ as the output of a neural network taking as inputs the explanatory variables and defined according to the modelling problem's need
- Inject $\hat{y}$ into a recurrent neural network, typically a LSTM network, whose number of units has been empirically derived as $\lceil log_2(s) \rceil$
- Fit the outputs of the recurrent neural networks with traditional logistic regression backpropagated through the whole network (both the LSTM and the previous neural network)

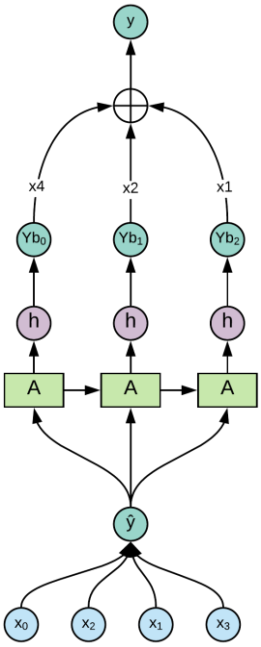


Fig. 4.10 Example of ordinal regression with a recurrent architecture for a variable with 8 states. A linear model (which can be replaced by a neural network) is fitted to model a quantitative variable through the cost function defined with a recurrent neural network.

Through experimental observations, the architecture as it is at present has shown some undesirable behaviours, especially when applied to noisy data, resulting from the fact that errors on every decisions are not equivalent. Indeed, wrongly predicting the first binary variable will systematically result in worst performance than wrongly predicting the last one while correctly predicting all the previous variables. As an example, considering the case of 8 states ordinal variable (indexed from 0 to 7):

- A model predicting probabilities (0.55, 0.9, 0.9) will output an ordinal state of 7
- A model predicting probabilities (0.45, 0.9, 0.9) will output an ordinal state of 3

Despite really similar binary output, the model gives sensibly different estimates of the investigated ordinal variable. One approach to counteract this unwanted property, is to

prevent the model from forcing binary variables to their target values if one of the preceding one was incorrectly predicted, by implementing a binary mask built from the agreement of target binary variables and their model estimates and applying it to the cost function. For an ordinal variable with s states and a neural network parameterized in θ, the modified cost function applied to one example with target output in binary form $(b_i)_{i\in[\![0,log_2(s)-1]\!]}$ can be defined the following manner:

$$Original\ cost\ function: L(\theta) = \sum_{i=0}^{log_2(s)-1} L_i(\theta)$$

With L_i the cross entropy associated with binary variable b_i $\forall i \in [\![0, n-1]\!]$

$$Masked\ cost\ function: L(\theta) = \sum_{i=0}^{log_2(s)-1} L_i(\theta) * \prod_{j=0}^{i-1} \delta_{b_j,\hat{b}_j}$$

With:

- L_i the cross entropy associated with binary variable b_i $\forall i \in [\![0, n-1]\!]$
- $\delta_{i,j}$ the Kronecker delta defined as: $\forall i,j \in \mathbb{N} = \begin{cases} 0\ if\ i \neq j \\ 1\ if\ i = j \end{cases}$
- b_j the j^{th} binary variable's true state
- $\hat{b}_j$ the j^{th} binary variable's model estimation

The ordinal regression method with modified cost function applied to a simulated dataset described below showed significantly better performances.

4.4.1.3 *Confrontation of ordinal regression methods on a simulated dataset*

In order to evaluate the performances of the proposed ordinal regression method based on binary search and recurrent neural networks, a simulated dataset was designed for ordinal regression:

- The ordinal variable's number of state was fixed to 64
- The training sampling size was fixed to 3200, with 50 examples per ordinal variable state
- 64 real numbers were uniformly sampled from $[0, 40]$
- 64 gaussian distributions were created with mean the 64 previous real numbers as mean, and random standard deviation sampled uniformly from $[0, 5]$
- The input variables are sampled from the created gaussian distributions
- The output variable states are defined by the ordering of the previously defined gaussian distributions mean values

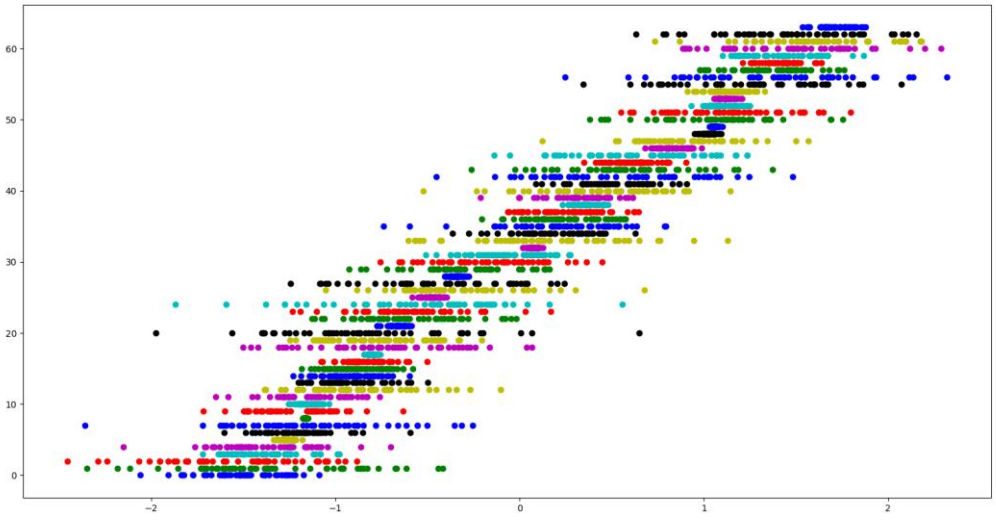


Fig. 4.11 example of dataset sampled from the aforedefined methodology

Although quite noisy, the dataset was voluntarily designed to be relatively simple, following the rationale that more complicated problems should be reducible to a similar dataset through the use of a deep neural network, under the assumption that the ordinal variable is an imperfect observation of a quantitative latent variable.

A thousand versions of the dataset are then fit with 4 different approaches:

- Input normalization and least square regression
- Input normalization and softmax regression
- Input normalization, and unimodal regression (as described in 4.4.1.1)
- Input normalization and ordered logit model (75)
- Input normalization, and binary search based ordinal regression without mask (as described in 4.4.1.2)
- Input normalization, linear model and binary search based ordinal regression with mask

All models were simultaneously adjusted through Adam optimization on the entire training dataset for 8000 iterations. The Adam procedure's learning rate was fixed at 0.01.

Model performance was evaluated through evaluation of both the prediction accuracy and the quadratic Cohen kappa coefficient between predictions and ground truth on a separate test dataset sampled from the same distribution, whose average values with 95% confidence intervals across all experiments are reported in Table 4.3.

Model	Quadratic Cohen Kappa	Test accuracy
Linear regression	0.961	0.087 ± 0.010

Softmax regression	0.961	0.168 ± 0.013
Unimodal regression	0.518	0.027 ± 0.006
Ordered logit	0.963	0.111 ± 0.0108
Unmasked recurrent binary search	0.951	$\mathbf{0.202 \pm 0.014}$
Masked recurrent binary search	0.957	$\mathbf{0.212 \pm 0.014}$

Table 4.3. Performance results of all proposed ordinal regression methods. The test accuracies' confidence intervals were derived from the Wilson score interval

All methods but the Unimodal regression exhibit sensibly similar Kappa values. In fact, although softmax and ordered logits regression models slightly outperform the proposed approach with regard to this statistic, the results were here presented with a sufficient roundup to see the differences between the measurements. It is however easy to notice that a roundup of 2 decimals leads to the exact same performance in term of Kappa metric for all methods beside Unimodal regression and unmasked recurrent binary search.

The test accuracies, whose poor values are easily explained by the amount of noise present in the modelling problem, as can be seen in figure 4.11, clearly shows that the proposed recurrent binary search ordinal regression method, clearly outperforms other existing approaches. Although the masked version of the method slightly outperforms its raw, unmasked counterpart, the gain in performance is not statistically significant.

In conclusion, a new method for the regression of ordinal data has been introduced and shown to be on par with most traditionally used techniques for the analysis of such problems in term of Kappa metric, and to significantly outperform them in term of classification accuracy. In addition, as the recurrent neural network is only used in the definition of the cost function, and not the actual model, this neural network-based technique can be used with

traditional linear modelling, which can be of importance especially in the scientific community for the interpretability of the derived model coefficients.

4.4.2 Local evaluation model

The first modelling subproblem, the local evaluation subproblem, consists of estimating the local CGI evaluations, evaluated on the information contained by 30 seconds slices of audio only, with the features extracted from said slices of audio. It constitutes a crucial element of the whole model but is as the same time the most challenging. Indeed, after feature extraction, the investigated dataset available for the adjustment of the local evaluation model is the following:

- 4365 ordinal output variables (CGI evaluation of the psychiatrist on the 30 seconds length audio samples)
- Their associated input variables consisting of 64 sequences of 1200 temporal observations, with an additional 4-states categorical variable indicating the judge's identity
- An additional disjoint set of 14228 slices of 30 seconds audio, without their corresponding output variables ($2x10^7$ slices when allowing full overlapping)

As such, this problem constitutes a regression analysis problem on a sequential dataset, as was defined in section 2.2.6, and can be addressed with either convolutional or recurrent networks in a rather straightforward manner.

Even though recurrent and convolutional neural architectures allow, through parameter sharing, for the modelling of highly dimensional datasets, the relatively small available amount of data considering the problem's dimensionality requires the use of some additional techniques:

- Semi supervised learning with ladder network (as described in section 2.5)
- Data augmentation with temporal and pitch translation

4.4.2.1 Data augmentation

Data augmentation regroups a family of techniques used to artificially augment the size of a dataset in order to reduce overfitting, typically by exploiting invariant properties of the investigated data (76). Considering image analysis as an example, most pictures of real world object can be transformed by vertical axial symmetry, small rotation, and small degradations, such as blurring or addition of small random noise, without actually changing the image's content, from the point of view of a human being. The pixel values of the resulting images, however, can be drastically impacted by such transformations, and can be considered by the neural network as entirely new observations during training.

As it makes use of invariant properties within the input variables of the investigated modelling problem, data augmentation needs to be pursued on a case by case basis. However, the analysis of audio signals allows for two relatively straightforward type of augmentation, temporal translation and pitch modification. Indeed, it is highly unlikely for two overlapping slices of an audio recording [t, t+30] and [t+2, t+32] to have been evaluated differently by a psychiatrist. As a consequence, slight temporal translation can be applied to the recording slices, without changing their associated ordinal evaluation. The question of setting the maximum amount of translation allowed, however, is not straightforward. Indeed, the variability of evaluations between consecutive slices strongly impacts the data's invariance to temporal shifts. Considering a degenerated dataset were the output variable never changes during an entire recording, any translation of any slice would still constitute a correctly evaluated new sample. In addition, the investigated variable being ordinal, the final model

should present some robustness to small changes in the output variable. Fig. 4.12 shows a histogram plot of the distance between every consecutive evaluations in the dataset, and clearly shows that the evaluation is relatively stable during one given recording (95% of consecutive differences are 1 or less). In addition, given the target variable's ordinal nature the derived model should show some robustness to small perturbations in the output. As a consequence, the maximum allowed temporal shift was set to 15 seconds, thus allowing for complete overlapping between slices but maximizing the amount of new observations derivable from the augmentation process.

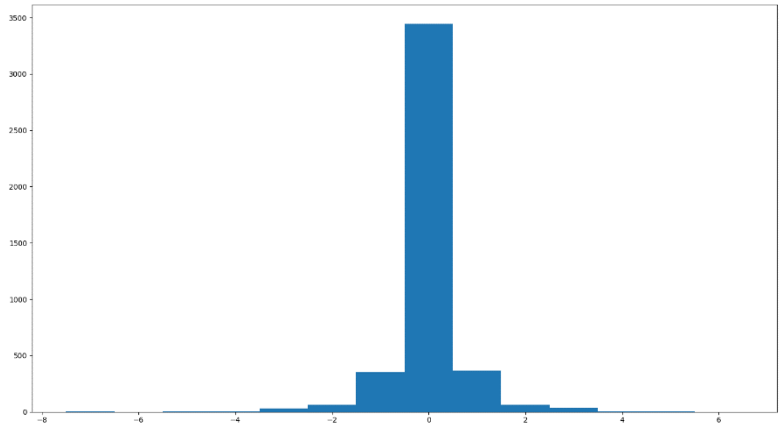


Fig.4.12 Histogram plot of distance between two consecutive local evaluations, demonstrating the stability of measurement across time

4.4.2.2 Model architecture

As previously mentioned, the wide availability of unevaluated recordings makes this modelling problem a choice candidate for semi supervised learning. To this date, the ladder network architecture and its variants are the most powerful family of semi supervised

learning algorithm on traditional benchmark datasets (45). As a consequence, it is only natural to use it in this application.

The current modelling problem is of sequential nature. As a consequence, multilayer perceptron architectures are not fitted for this analysis, at least for neural network's lowest levels. Two architectures are thus available for use, recurrent and convolutional networks. Convolutional neural networks were selected for this application, for three main reasons:

- Convolutional neural networks have been recently shown to consistently outperform recurrent neural networks on sequence classification tasks (77), especially when modelling long to very long sequences of data (typically longer than 200/500 temporal observations)
- Convolutional neural networks are extensively faster to train on GPU hardware, making the hyperparameter search process much more tractable. In addition, the final application requiring real time processing, this convenient property allows for the fitting of bigger neural networks while allowing for efficient real-time computation
- Due to their retroactive mechanism, the adaptation of recurrent neural networks in a ladder architecture is not as straightforward as it is for convolutional neural networks (which are essentially multilayer perceptrons with limited connections). Several methods have been presented over the years, but no consensus has been reached on which to definitely adopt (78)

Following traditional convolutional architectures, the ladder's encoder part will be comprised of a first stage of convolutional layers, organized in residual blocks to allow for deeper architectures, followed by a small, multilayer perceptron, that will take as inputs both the convolutional network's outputs and the categorical variable encoding the judge's identity,

written as a one-hot variable. The final ordinal regression step will be performed following 4 different approaches, namely:

- Least square regression
- Softmax regression (multinomial classification)
- Ordinal regression with unimodal probability distribution (as seen in 4.4.1.1)
- The approach introduced in this thesis, ordinal regression through binary search and recurrent neural networks (as seen in 4.4.1.2

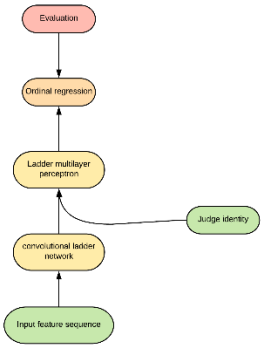


Fig. 4.13 Model architecture for local evaluation prediction

4.4.2.3 Hyperparameter search and model evaluation

The selected architecture presents a number of hyperparameters that need fine-tuning to extract the best possible model:

- Number of convolutional and feedforward layers
- Number of neurons per layer
- Filter size for each convolution neurons
- Stride parameter for convolutional layers
- Noise standard deviation for corrupted encoder
- Denoising cost for each encoder layer

- Momentum parameter for batch-normalization moving average updates
- Type of ordinal regression
- Eventual hyperparameters of ordinal regression methods
- Step rate for the Adam optimization algorithm
- Step rate exponential decay parameters
- Number of descent iterations across the whole dataset
- Size of supervised and unsupervised batch of subjects to optimize on at each given iteration

The exploration of this hyperparameter space will be done through random search guided by a validation performance measured on a held-out dataset. In order to evaluate the model's accuracy, the evaluated dataset has been divided into 3 parts:

- A training dataset of approximately 34 hours on which models are fit
- A validation dataset of approximately 5 hours on which hyperparameters are chosen
- A test dataset of approximately 5 hours on which the final model's performance are evaluated

4.4.2.4 Results

Around 70 models were adjusted following the previously defined methodology using the Tensorflow python library and a computer with a Nvidia GTX 1070 Graphics Processing Unit. One model took between 4 and 12 hours to fully train, which explains the small number of models fit compared to the dimensionality of the hyperparameter space.

The final model was derived by adjusting a model with the set of hyperparameters that maximized the validation performance, on both the training and validation set. Its accuracy was then evaluated on the test set through the computation of a quadratic Cohen Kappa

coefficient between the judge's decisions and the computer's. After convergence, the final Kappa value was evaluated at -0.05, indicating no better agreement than random chance.

4.4.3 Contextual evaluation model

The second modelling problem's objective is to approximate the judges' global evaluation from their corresponding local evaluations. From a practical perspective, this problem is much simpler than the local modelling problem. Indeed, the input's dimensionality is considerably lower, and no preprocessing is required, the evaluations being in a directly tractable format. In addition, the relationship between the local and global evaluation should be much simpler. It however introduces some complexities that were not present in the previous modelling problem:

- The modelling problem is a sequence-to-sequence modelling problem, which is not as straightforward to deal with than a standard classification problem
- The sample size is significantly reduced, from more than 4000 observed evaluation to 90 sequences
- The use of semi supervised settings is not applicable to this situation, as the input variables are as expansive to obtain than the output variables

4.4.3.1 Model architecture

Over the years, sequence to sequence modelling has become an important part of deep learning, as important real-life applications such as speech recognition or machine translation are based on this framework.

Recently, a powerful adaptation of convolutional neural nets has been introduced for the implementation of sequence-to-sequence models, the temporal convolutional networks (79). Temporal convolutional networks, which outperform all recurrent neural network

approaches in sequence-to-sequence modelling on traditional benchmark datasets, are essentially convolutional neural networks with two major modifications. First, in order to provide the user with sequential outputs of same length as the model's inputs, down sampling methods traditionally used in convolutional neural networks such as striding are discarded, and zero padding is used. Finally, the traditional convolution operation at the basis of the convolution is modified into what is called a dilated causal convolution, which differs twofold from its traditional definition:

- In order to respect causality, filters used to define the output at timestep t should not have access to inputs for times $t' > t$.
- A dilation factor d is introduced to the convolution operation. This dilation factor allows the convolution to access non-adjacent part of the input grid structure, thus expanding the length of the corresponding neuron's receptive field without using more parameters. Formally, for $A \in \mathbb{R}^n, B \in \mathbb{R}^m$ with $(n, m) \in \mathbb{N}^2$ such that $n > m$, the dilated convolution of A and B noted $A * B(d)$ is a vector defined as:

$$\forall i, j \in [\![1, n-m+1]\!], \ \ (A * B(d))_i = \sum_{k=0}^{m-1} (A)_{i+k.d} (B)_k$$

Fig. 4.14 shows an example of a temporal convolutional network. By applying successive convolutions with exponentially increasing dilation factors, these convolutional networks are able to model long sequences of data while requiring a surprisingly low number of parameters (the number of parameters evolves logarithmically with the sequences' lengths). In addition, temporal convolution still benefits from the powerful parallel implementation of convolution layers which makes them significantly faster than traditional recurrent neural networks.

Due to its high performance both in term of predictive power and computational efficiency, this neural architecture was chosen for the global modelling problem.

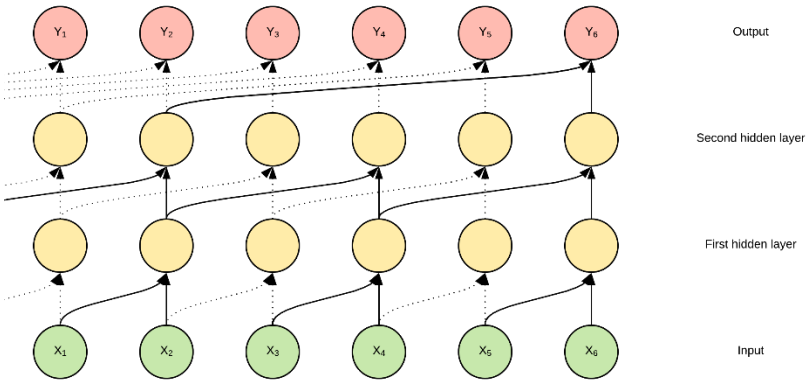


Fig. 4.14 temporal convolution network with 2 hidden layers. The dilation factors of the neural layers are in growing depth order 1, 2 and 4. The output at a given timestep has access to the input's every timestep before it.

4.4.3.2 Hyperparameter search and model evaluation

Following the implementation guidelines derived from the academic literature on temporal convolutional networks, the architecture for the global modelling problem was chosen with the following hyperparameters fixed:

- 3 residual blocks each composed of two temporal convolutional layers
- Convolution filter size of 2
- Dilation factor following a geometric progression of reason 2 and first term 1
- Batch-normalization and drop out regularization applied to every convolutional layer

Although these experimentally derived hyperparameters allow for a simpler search space, a few hyperparameters remain to tune manually:

- The Adam optimization procedure's learning rate
- The drop-out probability parameter for each layer
- The momentum parameter for batch-normalization moving average updates
- Type of ordinal regression
- Eventual hyperparameters of ordinal regression methods

The same three group repartition of the dataset was used for this problem's hyperparameter tuning process.

4.4.3.3 Results

10 models were ran using Tensorflow on the same computer as for the previous experiment. The number of model ran for this problem is significantly lower than for the local modelling problem, because satisfying models were rapidly found, and as a consequence more time resource was dedicated to the local problem. Following the same methodology as used in the previous experiment, the model presenting the best predictive power showed a quadratic Kappa of 0.8, which is considered as just in-between strong and perfect agreement. However, this result was obtained by using the judge's local predictions as input variables, and not their approximation with the local model. Due to their poor predictive quality, no model was adjusted using the evaluations predicted by the local model as input variables.

As a consequence, this experiment shows that under the assumption that the local modelling is tractable, the global CGI-S evaluation can be predicted from suicide hotline recordings by following the proposed methodology.

5 Conclusion and future work

Deep artificial neural networks have already had a significant impact on society and should continue to do so in the years to come. Through this thesis, they have been described as powerful expansions of traditional statistical methods that allow for the analysis of complex, non-linear interactions on hierarchical datasets, such as longitudinal datasets, leading to opportunities that were believed to be unreasonable not so long ago. In addition, the two proposed practical applications investigated, although not completely successful, clearly demonstrate the wide potential of these methods in the biomedical sciences.

Indeed, cluster analysis of longitudinal data is an important topic for epidemiologists because it provides insight in data not achievable with classical methods among which random effect generalized linear models for repeated measures. Several approaches exist at the moment, among which mixture modelling and historical segmentation algorithms like K-Means. This thesis shows that deep learning technics could be a promising alternative, being able to grasp non-spherical clusters (where K-Means procedure is not efficient) with complex temporal evolution or non-Gaussian mixtures (where standard mixture modelling shows strong limitations).

Coherence between the proposed and traditional approaches is ensured for simple cases, which is likely to comfort results already published even if it is strongly suggested that researchers look for robust segmentations of trajectories with a systematic use of different clustering approaches.

The proposed network architecture leaves several questions that might inform future works. First, the investigated method separates the embedding extraction and the clustering procedure into two distinct steps, led by two distinct optimizing processes. Although leading to satisfying results, such an implementation stands out of the usual deep learning

practitioner's philosophy. Indeed, it is usually preferred, when designing a neural network for a given task, to gather every part of the procedure in a single optimization process. Indeed, the low dimensional embedding extracted from the data in the proposed approach isn't necessarily optimized for a clustering task to be performed subsequently. The design of architectures optimizing jointly the low dimensional embedding and the extraction of a data partition is an active area of research, with some promising methods already proposed (55).

Finally, these experiments clearly show that technics originally coming from computer science have a great potential in the field of epidemiology and biostatistics. These new approaches are most often quite demanding in terms of sample size requirements, but they are currently the best available solution for handling non-linear and complex relationships, potentially leading to breakthroughs unthinkable until now.

Although less successful, the proposed application of deep neural network to psychiatry was not entirely void of interest. The derived model's performance to predict the local CGI-S evaluations from audio recordings is clearly not sufficient for practical applications. However, it has been shown that, conditionally to the implementation of a good local model, the global CGI-S evaluation can be extracted with good performances. It is then only natural to focus future work on finding better models for the prediction of the local evaluations from the 30 seconds slices of audio. A number of reasons can explain its lack of performance:

- The information used to predict the target variable is insufficient. This is highly likely, as only information about the caller's voice was used in this experiment. Including the recording's linguistic content in the analysis might lead to significantly better predictions

- The sample size is not sufficient enough to describe the variability of a complex phenomenon such as suicidal risk.
- Time and computational resources might have been insufficient to allow for a thorough enough exploration of the hyperparameter space, and other (bigger) types of neural architectures might lead to better predictive performance
- The feature extracted for analysis might not capture the information the judge used to make their decisions
- Deep artificial neural networks do not have the predictive power to capture the complex relationship between a person's speech and suicidal intent
- The judges were unable to assess the 30 seconds slices of audio independently, and their decisions are correlated to previous slices which are not observed by the learning algorithm

It is worth noticing that most of these possible explanations can be addressed simply by devoting more resources to the problem, such as time, human or computational power. As a consequence, it is reasonable to expect improvement in predictive performance in the near future with future work on this dataset.

The most promising lead for improving local evaluation predictions is the inclusion of the recording's linguistic content in the analysis. Indeed, it is highly likely to contain a substantial part, if not most, of the information considered by the judges in their decision process. As aforementioned, no speech-to-text algorithm is available for the automated transcription of French spoken with a Créole accent, nor for Créole dialects. Consequently, a manual transcription of the entire evaluated dataset will be required. This process is extremely time consuming, but its benefits will be many:

- The conversation's content can be analyzed with NLP techniques, including deep neural networks, to perform content-based predictions
- The availability of both the content and the audio information allow for powerful multimodal analysis
- The learning algorithm can be given a much richer representation of the investigated data, for instance by implementing a specific voice-to-text algorithm. As most speech recognition algorithms are only based on MFCC features, it would not require more preprocessing, and advanced speech to text technologies are able to explicitly map the language with its place in time in the audio recording as can be seen in figure 5.1. This would allow for a powerful, hierarchical representation of the audio signal which could then be used to better fuse the information contained in the speaker's voice and verbatim. It would in addition provide an opportunity to use transfer learning to improve the model's efficiency

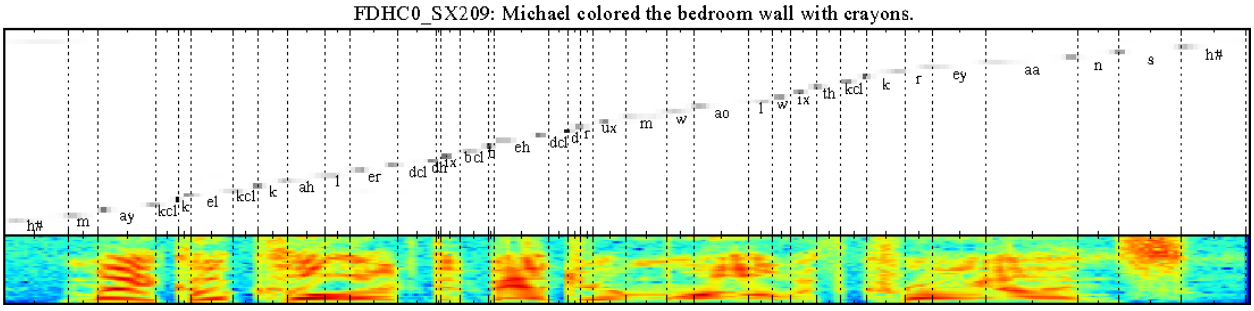


Fig. 5.1 example of representation obtained with an attention-based speech-to-text neural model (80). The phonetic decomposition of the speaker's language is precisely located on the audio signal

Although language modelling constitutes the most promising lead to obtain better predictions, several other approaches are worth exploring, such as gathering additional observations, using bigger networks or improving the hyperparameter tuning. However,

these leads require some investment in term of human or financial resources, as they require either additional work from the judges or more specialized hardware, which is considerably expensive at the present moment.

In addition, this project led to the definition of a new method for ordinal regression. Although its benefits on the investigated, real life dataset are difficult to assess, it yielded clearly superior results in the simple simulated dataset compared to the state of the art. As a consequence, the behavior of this new method for ordinal analysis should be investigated further, whether from a practical or theoretical perspective.

As a final conclusion, I would like to sum up my thesis' most significant contributions to the Biomedical Sciences.

First, I designed a new method of longitudinal data clustering compared with existing state of the art methods in an epidemiological setting This innovative method yields performancesthat are similar to the state of the art on simple datasets, and outperforms them in a variety of situations such as when analyzing trajectories with non-spherical clusters, which is quite often the case in epidemiology. In addition, this method can also be used as a powerful visualization tool, which is an important area of research in time series analysis.

Second, I created a new method of ordinal regression and compared it with existing state of the art methods. This method was also shown to outperform its state of the art counterparts. Although its underlying theory was not investigated, its fundamentally different take on ordinal modelling problems might make it an interesting lead to improve the scientific community's understanding of these models.

Finally, after a commited work of communication on Artificial Intelligence to the psychiatric research community, I managed to convince several researcher to work with me on gathering a one of a kind dataset. Although no significant results were obtained from this dataset during my time as a MSc. student, a lot remains to be done with this dataset and I firmly believe in its potential for future work.

6 Bibliography

1. Najafabadi MM, Villanustre F, Khoshgoftaar TM, Seliya N, Wald R, Muharemagic E. Deep learning applications and challenges in big data analytics. J Big Data. 2015 Dec 1;2(1):1.

2. LeCun Y, Bengio Y, Hinton G. Deep learning. Nature. 2015 May;521(7553):436–44.

3. Esteva A, Kuprel B, Novoa RA, Ko J, Swetter SM, Blau HM, et al. Dermatologist-level classification of skin cancer with deep neural networks. Nature. 2017 Feb;542(7639):115–8.

4. Lakhani P, Sundaram B. Deep Learning at Chest Radiography: Automated Classification of Pulmonary Tuberculosis by Using Convolutional Neural Networks. Radiology. 2017 Apr 24;284(2):574–82.

5. Shwartz-Ziv R, Tishby N. Opening the Black Box of Deep Neural Networks via Information. ArXiv170300810 Cs [Internet]. 2017 Mar 2 [cited 2018 Jul 12]; Available from: http://arxiv.org/abs/1703.00810

6. Goodfellow I, Bengio Y, Courville A. Deep Learning. MIT Press; 2016.

7. Some Studies in Machine Learning Using the Game of Checkers - IBM Journals & Magazine [Internet]. [cited 2018 Jul 12]. Available from: https://ieeexplore.ieee.org/document/5392560/

8. Mitchell TM. Machine Learning. 1st ed. New York, NY, USA: McGraw-Hill, Inc.; 1997.

9. Mohri M, Rostamizadeh A, Talwalkar A. Foundations of Machine Learning. The MIT Press; 2012.

10. Boser BE, Guyon IM, Vapnik VN. A Training Algorithm for Optimal Margin Classifiers. In: Proceedings of the Fifth Annual Workshop on Computational Learning Theory [Internet]. New York, NY, USA: ACM; 1992 [cited 2018 Jan 17]. p. 144–152. (COLT '92). Available from: http://doi.acm.org/10.1145/130385.130401

11. The Elements of Statistical Learning - Data Mining, Inference, | Trevor Hastie | Springer [Internet]. [cited 2018 Jan 17]. Available from: http://www.springer.com/gb/book/9780387848570

12. Comparing different clustering algorithms on toy datasets — scikit-learn 0.19.1 documentation [Internet]. [cited 2018 Jul 12]. Available from: http://scikit-learn.org/stable/auto_examples/cluster/plot_cluster_comparison.html

13. Press TM. Semi-Supervised Learning [Internet]. The MIT Press. [cited 2018 Jul 12]. Available from: https://mitpress.mit.edu/books/semi-supervised-learning

14. Doukkali F. Simple explanation of Semi-Supervised Learning and Pseudo Labeling [Internet]. Towards Data Science. 2017 [cited 2018 Jul 12]. Available from:

https://towardsdatascience.com/simple-explanation-of-semi-supervised-learning-and-pseudo-labeling-c2218e8c769b

15. Statistical models theory and practice 2nd edition | Statistical theory and methods [Internet]. Cambridge University Press. [cited 2018 Jul 12]. Available from: http://www.cambridge.org/gb/academic/subjects/statistics-probability/statistical-theory-and-methods/statistical-models-theory-and-practice-2nd-edition

16. Cox DR. The Regression Analysis of Binary Sequences. J R Stat Soc Ser B Methodol. 1958;20(2):215–42.

17. Hinton GE. Rectified Linear Units Improve Restricted Boltzmann Machines Vinod Nair.

18. Rectifier Nonlinearities Improve Neural Network Acoustic Models - Semantic Scholar [Internet]. [cited 2018 Apr 26]. Available from: /paper/Rectifier-Nonlinearities-Improve-Neural-Network-Maas-Hannun/367f2c63a6f6a10b3b64b8729d601e69337ee3cc

19. Rumelhart DE, Hinton GE, Williams RJ. Learning representations by back-propagating errors. Nature. 1986 Oct 9;323(6088):533–6.

20. Mozer M. A Focused Backpropagation Algorithm for Temporal Pattern Recognition. Complex Syst. 1995 Jan 1;3.

21. Bengio Y, Simard P, Frasconi P. Learning long-term dependencies with gradient descent is difficult. IEEE Trans Neural Netw. 1994 Mar;5(2):157–66.

22. Cho K, van Merrienboer B, Gulcehre C, Bahdanau D, Bougares F, Schwenk H, et al. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. ArXiv14061078 Cs Stat [Internet]. 2014 Jun 3 [cited 2018 Mar 2]; Available from: http://arxiv.org/abs/1406.1078

23. Long Short-Term Memory | Neural Computation | MIT Press Journals [Internet]. [cited 2017 Sep 7]. Available from: http://www.mitpressjournals.org/doi/abs/10.1162/neco.1997.9.8.1735

24. Zhang W, Itoh K, Tanida J, Ichioka Y. Parallel distributed processing model with local space-invariant interconnections and its optical architecture. Appl Opt. 1990 Nov 10;29(32):4790–7.

25. Lecun Y, Bengio Y. Convolutional networks for images, speech, and time-series. Handb Brain Theory Neural Netw [Internet]. 1995 [cited 2018 Jul 12]; Available from: https://nyuscholars.nyu.edu/en/publications/convolutional-networks-for-images-speech-and-time-series

26. Oord A van den, Dieleman S, Zen H, Simonyan K, Vinyals O, Graves A, et al. WaveNet: A Generative Model for Raw Audio. ArXiv160903499 Cs [Internet]. 2016 Sep 12 [cited 2018 Jul 12]; Available from: http://arxiv.org/abs/1609.03499

27. Dumoulin V, Visin F. A guide to convolution arithmetic for deep learning. ArXiv160307285 Cs Stat [Internet]. 2016 Mar 23 [cited 2018 Jul 12]; Available from: http://arxiv.org/abs/1603.07285

28. Goodfellow IJ, Warde-Farley D, Mirza M, Courville A, Bengio Y. Maxout Networks. ArXiv13024389 Cs Stat [Internet]. 2013 Feb 18 [cited 2017 Nov 14]; Available from: http://arxiv.org/abs/1302.4389

29. Zeiler MD, Fergus R. Visualizing and Understanding Convolutional Networks. ArXiv13112901 Cs [Internet]. 2013 Nov 12 [cited 2018 Jul 12]; Available from: http://arxiv.org/abs/1311.2901

30. Hochreiter S, Bengio Y, Frasconi P, Schmidhuber J. Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies. 2001.

31. Ripley BD. Pattern Recognition and Neural Networks by Brian D. Ripley [Internet]. Cambridge Core. 1996 [cited 2018 Jul 13]. Available from: /core/books/pattern-recognition-and-neural-networks/4E038249C9BAA06C8F4EE6F044D09C5C

32. Kohavi R. A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection. In: Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2 [Internet]. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 1995 [cited 2018 Jul 13]. p. 1137–1143. (IJCAI'95). Available from: http://dl.acm.org/citation.cfm?id=1643031.1643047

33. Geman S, Bienenstock E, Doursat R. Neural Networks and the Bias/Variance Dilemma. Neural Comput. 1992 Jan;4(1):1–58.

34. Ng AY. Feature Selection, L1 vs. L2 Regularization, and Rotational Invariance. In: Proceedings of the Twenty-first International Conference on Machine Learning [Internet]. New York, NY, USA: ACM; 2004 [cited 2018 Jul 13]. p. 78–. (ICML '04). Available from: http://doi.acm.org/10.1145/1015330.1015435

35. Srivastava N, Hinton G, Krizhevsky A, Sutskever I, Salakhutdinov R. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. J Mach Learn Res. 2014;15:1929–58.

36. Hoerl AE, Kennard RW. Ridge Regression: Biased Estimation for Nonorthogonal Problems. Technometrics. 1970 Feb 1;12(1):55–67.

37. Regression Shrinkage and Selection via the Lasso on JSTOR [Internet]. [cited 2018 Jul 13]. Available from: https://www.jstor.org/stable/2346178?seq=1#page_scan_tab_contents

38. Glorot X, Bengio Y. Understanding the difficulty of training deep feedforward neural networks. In: PMLR [Internet]. 2010 [cited 2018 Jan 31]. p. 249–56. Available from: http://proceedings.mlr.press/v9/glorot10a.html

39. Bengio Y, Lamblin P, Popovici D, Larochelle H. Greedy Layer-Wise Training of Deep Networks. In: Schölkopf B, Platt JC, Hoffman T, editors. Advances in Neural Information

Processing Systems 19 [Internet]. MIT Press; 2007 [cited 2018 Feb 5]. p. 153–160. Available from: http://papers.nips.cc/paper/3048-greedy-layer-wise-training-of-deep-networks.pdf

40. LeCun Y, Bottou L, Orr GB, Müller K-R. Efficient BackProp. In: Neural Networks: Tricks of the Trade [Internet]. Springer, Berlin, Heidelberg; 1998 [cited 2018 Jul 13]. p. 9–50. (Lecture Notes in Computer Science). Available from: https://link.springer.com/chapter/10.1007/3-540-49430-8_2

41. Kingma DP, Ba J. Adam: A Method for Stochastic Optimization. ArXiv14126980 Cs [Internet]. 2014 Dec 22 [cited 2017 Sep 8]; Available from: http://arxiv.org/abs/1412.6980

42. Ioffe S, Szegedy C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. ArXiv150203167 Cs [Internet]. 2015 Feb 10 [cited 2017 Nov 17]; Available from: http://arxiv.org/abs/1502.03167

43. Liou C-Y, Cheng W-C, Liou J-W, Liou D-R. Autoencoder for Words. Neurocomput. 2014 Sep;139:84–96.

44. Vincent P, Larochelle H, Bengio Y, Manzagol P-A. Extracting and Composing Robust Features with Denoising Autoencoders. In: Proceedings of the 25th International Conference on Machine Learning [Internet]. New York, NY, USA: ACM; 2008 [cited 2018 Jan 19]. p. 1096–1103. (ICML '08). Available from: http://doi.acm.org/10.1145/1390156.1390294

45. Shinoda S, Worrall DE, Brostow GJ. Virtual Adversarial Ladder Networks For Semi-supervised Learning. ArXiv171107476 Cs Stat [Internet]. 2017 Nov 20 [cited 2018 Jan 19]; Available from: http://arxiv.org/abs/1711.07476

46. Rasmus A, Valpola H, Honkala M, Berglund M, Raiko T. Semi-Supervised Learning with Ladder Networks. ArXiv150702672 Cs Stat [Internet]. 2015 Jul 9 [cited 2018 Jan 19]; Available from: http://arxiv.org/abs/1507.02672

47. Pezeshki M, Fan L, Brakel P, Courville A, Bengio Y. Deconstructing the Ladder Network Architecture. ArXiv151106430 Cs [Internet]. 2015 Nov 19 [cited 2018 Jan 25]; Available from: http://arxiv.org/abs/1511.06430

48. Genolini C, Falissard B. KmL: a package to cluster longitudinal data. Comput Methods Programs Biomed. 2011 Dec;104(3):e112-121.

49. Jones BL, Nagin DS. Advances in Group-Based Trajectory Modeling and an SAS Procedure for Estimating Them. Sociol Methods Res. 2007 May 1;35(4):542–71.

50. Rolfe JT, LeCun Y. Discriminative Recurrent Sparse Auto-Encoders. ArXiv13013775 Cs [Internet]. 2013 Jan 16 [cited 2017 Sep 8]; Available from: http://arxiv.org/abs/1301.3775

51. Body-size throughout life and risk of depression in postmenopausal women: Findings from the E3N cohort - Perquier - 2014 - Obesity - Wiley Online Library [Internet]. [cited 2017 Sep 8]. Available from: http://onlinelibrary.wiley.com/doi/10.1002/oby.20799/full

52. Caliński T, Harabasz J. A dendrite method for cluster analysis. Commun Stat. 1974 Jan 1;3(1):1–27.

53. Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. ArXiv160304467 Cs [Internet]. 2016 Mar 14 [cited 2017 Sep 8]; Available from: http://arxiv.org/abs/1603.04467

54. Falissard B. A spherical representation of a correlation matrix. J Classif. 1996 Sep 1;13(2):267–80.

55. Dilokthanakul N, Mediano PAM, Garnelo M, Lee MCH, Salimbeni H, Arulkumaran K, et al. Deep Unsupervised Clustering with Gaussian Mixture Variational Autoencoders. ArXiv161102648 Cs Stat [Internet]. 2016 Nov 8 [cited 2017 Sep 21]; Available from: http://arxiv.org/abs/1611.02648

56. Alghowinem S, Goecke R, Wagner M, Epps J, Hyett M, Parker G, et al. Multimodal Depression Detection:Fusion Analysis of Paralinguistic, Head Pose and Eye Gaze Behaviors. IEEE Trans Affect Comput. 2017;1–1.

57. Williamson JR, Quatieri TF, Helfer BS, Horwitz R, Yu B, Mehta DD. Vocal Biomarkers of Depression Based on Motor Incoordination. In: Proceedings of the 3rd ACM International Workshop on Audio/Visual Emotion Challenge [Internet]. New York, NY, USA: ACM; 2013 [cited 2018 Jul 13]. p. 41–48. (AVEC '13). Available from: http://doi.acm.org/10.1145/2512530.2512531

58. Busner J, Targum SD. The Clinical Global Impressions Scale. Psychiatry Edgmont. 2007 Jul;4(7):28–37.

59. McHugh ML. Interrater reliability: the kappa statistic. Biochem Medica. 2012 Oct 15;22(3):276–82.

60. Landis JR, Koch GG. The Measurement of Observer Agreement for Categorical Data. Biometrics. 1977;33(1):159–74.

61. Beckham C, Pal C. Unimodal probability distributions for deep ordinal classification. ArXiv170505278 Stat [Internet]. 2017 May 15 [cited 2018 Jan 18]; Available from: http://arxiv.org/abs/1705.05278

62. Citations Screenshots and Permissions | Audacity ® [Internet]. [cited 2018 Jul 13]. Available from: https://www.audacityteam.org/about/citations-screenshots-and-permissions/

63. K B. English: Window function and its Fourier transform: Rectangular window [Internet]. 2013 [cited 2018 Jul 13]. Available from:

https://commons.wikimedia.org/wiki/File:Window_function_and_frequency_response_-_Rectangular.svg

64. Niemitalo O. English: Hamming window function and its Fourier transform. This image was generated with parameter α = 0.53836. Another common value is 0.54. The Hamming window is a member of the cosine-sum family, in which α corresponds to parameter a0. [Internet]. 2013 [cited 2018 Jul 13]. Available from: https://commons.wikimedia.org/wiki/File:Window_function_and_frequency_response_-_Hamming_(alpha_%3D_0.53836).svg

65. On the use of autocorrelation analysis for pitch detection - IEEE Journals & Magazine [Internet]. [cited 2018 Jul 9]. Available from: https://ieeexplore.ieee.org/abstract/document/1162905/

66. Shinohara S, Omiya Y, Nakamura M, Higuchi M, Hagiwara N, Takano T, et al. Major depression index derived from the relationship between hurst exponent and zero crossing rate in voice. In 2017.

67. Eyben F, Weninger F, Gross F, Schuller B. Recent Developments in openSMILE, the Munich Open-source Multimedia Feature Extractor. In: Proceedings of the 21st ACM International Conference on Multimedia [Internet]. New York, NY, USA: ACM; 2013 [cited 2017 Sep 7]. p. 835–838. (MM '13). Available from: http://doi.acm.org/10.1145/2502081.2502224

68. Eyben F, Wöllmer M, Schuller B. Opensmile: The Munich Versatile and Fast Open-source Audio Feature Extractor. In: Proceedings of the 18th ACM International Conference on Multimedia [Internet]. New York, NY, USA: ACM; 2010 [cited 2018 Jul 13]. p. 1459–1462. (MM '10). Available from: http://doi.acm.org/10.1145/1873951.1874246

69. Deng L, O'Shaughnessy D. Speech Processing: A Dynamic and Optimization-Oriented Approach. CRC Press; 2003. 656 p.

70. Fast Fourier Transform - Algorithms and Applications | K R Rao | Springer [Internet]. [cited 2018 Jul 13]. Available from: https://www.springer.com/gb/book/9781402066283

71. Real-time Speech and Music Classification by Large Audio Feature Space Extraction | Florian Eyben | Springer [Internet]. [cited 2018 Jul 13]. Available from: https://www.springer.com/gb/book/9783319272986

72. Is this a correct interpretation of the DCT step in MFCC calculation? [Internet]. Signal Processing Stack Exchange. [cited 2018 Jul 13]. Available from: https://dsp.stackexchange.com/questions/15938/is-this-a-correct-interpretation-of-the-dct-step-in-mfcc-calculation

73. Tsanas A. Accurate telemonitoring of Parkinson's disease symptom severity using nonlinear speech signal processing and statistical machine learning. :261.

74. Sutskever I, Vinyals O, Le QV. Sequence to Sequence Learning with Neural Networks. :9.

75. McCullagh P. Regression Models for Ordinal Data. J R Stat Soc Ser B Methodol. 1980;42(2):109–42.

76. Dyk DA van, Meng X-L. The Art of Data Augmentation. J Comput Graph Stat. 2001 Mar 1;10(1):1–50.

77. Bai S, Kolter JZ, Koltun V. An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling. ArXiv180301271 Cs [Internet]. 2018 Mar 3 [cited 2018 Apr 27]; Available from: http://arxiv.org/abs/1803.01271

78. Prémont-Schwarz I, Ilin A, Hao T, Rasmus A, Boney R, Valpola H. Recurrent Ladder Networks. In: Guyon I, Luxburg UV, Bengio S, Wallach H, Fergus R, Vishwanathan S, et al., editors. Advances in Neural Information Processing Systems 30 [Internet]. Curran Associates, Inc.; 2017 [cited 2018 Jul 13]. p. 6009–6019. Available from: http://papers.nips.cc/paper/7182-recurrent-ladder-networks.pdf

79. Lea C, Vidal R, Reiter A, Hager GD. Temporal Convolutional Networks: A Unified Approach to Action Segmentation. ArXiv160808242 Cs [Internet]. 2016 Aug 29 [cited 2018 May 2]; Available from: http://arxiv.org/abs/1608.08242

80. Attention-Based Models for Speech Recognition - Semantic Scholar [Internet]. [cited 2018 Jul 12]. Available from: /paper/Attention-Based-Models-for-Speech-Recognition-Chorowski-Bahdanau/26c8d040bef85ad6dde55a8f71af936fb38356ad

7 Appendix

7.1 Longitudinal clustering experiment code

7.1.1 Model

```
import tensorflow as tf
import math
```

```
def inference(data, rec_size, hidden_size1, hidden_size2, bottle_size):

    with tf.name_scope("recurrent_encoder"):
        lstm_cell_1 = tf.contrib.rnn.LSTMCell(rec_size, forget_bias=1.0, state_is_tuple=True,
activation=tf.nn.tanh)

        output, state = tf.nn.dynamic_rnn(lstm_cell_1, data, dtype=tf.float32)

        output = state[1]

        print(output)


        w_0 = tf.Variable(
            tf.truncated_normal([rec_size, bottle_size],
                                stddev=1.0 / math.sqrt(float(50))), name='weights')
        b_0 = tf.Variable(tf.zeros([bottle_size]),
                          name='biases')
        print(output)
        print(w_0)
        bottle = tf.matmul(output, w_0) + b_0

        out = tf.nn.relu(bottle)

    with tf.name_scope("decoder"):
        w_1 = tf.Variable(
            tf.truncated_normal([bottle_size, hidden_size1],
                                stddev=1.0 / math.sqrt(float(50))), name='weights')
        b_1 = tf.Variable(tf.zeros([hidden_size1]),
                          name='biases')

        hidden1 = tf.matmul(out, w_1) + b_1

        w_2 = tf.Variable(
            tf.truncated_normal([hidden_size1, hidden_size2],
                                stddev=1.0 / math.sqrt(float(50))), name='weights')
        b_2 = tf.Variable(tf.zeros([hidden_size2]),
                          name='biases')

        hidden2 = tf.nn.relu(tf.matmul(hidden1, w_2) + b_2)
        print(hidden2)

        w_3 = tf.Variable(
            tf.truncated_normal([hidden_size2, 10],
                                stddev=1.0 / math.sqrt(float(50))), name='weights')
        b_3 = tf.Variable(tf.zeros([10]),
                          name='biases')

        decoder = tf.matmul(hidden2, w_3) + b_3
        print(decoder)

    return decoder, bottle


def loss(decoder, data):
    print(decoder)
    data = tf.reduce_mean(data, axis=2)
    print(data)
    loss_value = tf.losses.mean_squared_error(decoder, data, weights=1)

    return loss_value


def training(loss_value):
    optimizer = tf.train.RMSPropOptimizer(0.001, decay=0.9, momentum=0.7, epsilon=1e-10)
    global_step = tf.Variable(0, name='global_step', trainable=False)
    train_op = optimizer.minimize(loss_value, global_step=global_step)
    return train_op
```

7.1.2 Training

```
import tensorflow as tf
import numpy as np
```

```
import math
import random
import time
import pickle

import Model as model

batch size = 2000
rec size = 10
bottle size = 2
hidden_size1 = 5
hidden_size2 = 10

max steps = 20000

data = []
lab = []

simu_treatment = np.random.uniform(-2, 2, 5000)
simu treatment = [[math.sin((x + 1 - 1) * math.pi / 2) * 5 + 10 + np.random.normal(0, 1, 1),
                   math.sin((x + 2 - 1) * math.pi / 2) * 5 + 10 + np.random.normal(0, 1, 1),
                   math.sin((x + 3 - 1) * math.pi / 2) * 5 + 10 + np.random.normal(0, 1, 1),
                   math.sin((x + 4 - 1) * math.pi / 2) * 5 + 10 + np.random.normal(0, 1, 1),
                   math.sin((x + 5 - 1) * math.pi / 2) * 5 + 10 + np.random.normal(0, 1, 1),
                   math.sin((x + 6 - 1) * math.pi / 2) * 5 + 10 + np.random.normal(0, 1, 1),
                   math.sin((x + 7 - 1) * math.pi / 2) * 5 + 10 + np.random.normal(0, 1, 1),
                   math.sin((x + 8 - 1) * math.pi / 2) * 5 + 10 + np.random.normal(0, 1, 1),
                   math.sin((x + 9 - 1) * math.pi / 2) * 5 + 10 + np.random.normal(0, 1, 1),
                   math.sin((x + 10 - 1) * math.pi / 2) * 5 + 10 + np.random.normal(0, 1, 1)]
for x in simu_treatment]
simu treatment = np.reshape(np.array(simu treatment), [5000,10])

simu_control = np.add(np.random.normal(0, 1, [5000, 10]), 10)

training data = np.concatenate((simu treatment[:3000], simu control[:3000]), axis=0)
training labels = np.concatenate((np.zeros([3000]), np.add(np.zeros([3000]), 1)), axis=0)

test data = np.concatenate((simu treatment[3000:], simu control[3000:]), axis=0)
test_labels = np.concatenate((np.zeros([2000]), np.add(np.zeros([2000]), 1)), axis=0)

# for RNN models data = np.reshape(data, [6000, 10, 1])

with open("dataset", "wb") as f:
    pickle.dump(data, f)

def get_batch_training():
    batch = []

    for _ in range(batch_size):
        rand = random.randrange(0, 6000)
        batch.append(data[rand])

    return batch


def fill_feed_dict(input_pl, type_dict):
    if type_dict == 'train':
        data in = get batch training()

    else:
        data_in = data

    fed_dict = {
        input pl: data in,
    }

    return fed_dict

with tf.Graph().as_default():

    input_placeholders = tf.placeholder(tf.float32, shape=[None, 10, 1])

    decoder, embed = model.inference(input_placeholders,
                                     rec_size,
                                     hidden size1,
                                     hidden size2,
                                     bottle_size)
```

```
    loss = model.loss(decoder, input placeholders)

    train op = model.training(loss)

    sess = tf.Session()

    sess.run(tf.global variables initializer())

    # Start the training loop.
    for step in range(max_steps):
        start_time = time.time()

        # Fill a feed dictionary with the actual set of images and labels
        # for this particular training step.
        feed_dict = fill_feed_dict(input_placeholders, 'train')

        # Run one step of the model.  The return values are the activations
        # from the `train_op` (which is discarded) and the `loss` Op.  To
        # inspect the values of your Ops or variables, you may include them
        # in the list passed to sess.run() and the value tensors will be
        # returned in the tuple from the call.
        _, loss_value = sess.run([train_op, loss], feed_dict=feed_dict)

        duration = time.time() - start time

        # Write the summaries and print an overview fairly often.
        if step % 100 == 0:
            # Print status to stdout.
            print('Step %d: total_loss = %.2f (%.3f sec) ' % (step, loss_value, duration))

        if step % 1000 == 0:
            feed_dict = fill_feed_dict(input_placeholders, 'everyone')
            bottle_value = sess.run([embed], feed_dict=feed_dict)

            with open("embedding_" + str(step), "wb") as f:
                pickle.dump(bottle_value, f)
```

7.2 Ordinal regression experiment code

```
import tensorflow as tf
import numpy as np
from sklearn.metrics import cohen_kappa_score
import time
from ordinal_regression import full_unimodal
import mord


num_classes = 64
group_sample_size = 50

def generate_data(size, num_per_group):

    means = np.random.uniform(0, 40, size=size)
    var = np.random.uniform(0, 5, size=size)

    means = np.sort(means)

    input_data = []
    input_test = []
    labels = []
    for i in range(size):
        input data.append(np.random.normal(means[i], var[i], size=[num per group, 1]))
        input test.append(np.random.normal(means[i], var[i], size=[num per group, 1]))
        labels.append(np.ones(shape=num_per_group) * i)

    input_data = np.concatenate(input_data, axis=0)
    input test = np.concatenate(input test, axis=0)
    labels = np.concatenate(labels)

    mean = np.mean(input_data)
```

```
    var = np.var(input_data)

    input data = (input data - mean) / np.sqrt(var)
    input test = (input test - mean) / np.sqrt(var)

    return input_data, labels.astype(np.int32), input_test, labels.astype(np.int32)


def get_bin_search_labels(labels, search_size):

    val_parse = labels
    binary_search_labels = []

    for i in reversed(range(search_size)):
        val parse = (val parse % np.power(2, i + 1))
        binary_search_labels.append(val_parse // np.power(2, i))

    return tf.stack(binary_search_labels, axis=-1)


inputs = tf.placeholder(tf.float32, shape=[None, 1])
labels = tf.placeholder(tf.int64, shape=[None])
binary_labels = get_bin_search_labels(labels, 6)

initialize = tf.variance_scaling_initializer(scale=1.0, distribution='normal')

with tf.name_scope('binary_search'):

    out = tf.expand dims(tf.tile(inputs, [1, 6]), axis=-1)

    lstm_cell = tf.nn.rnn_cell.LSTMCell(forget_bias=1.0, state_is_tuple=True,
activation=tf.nn.tanh, num units=6)

    outputs, state = tf.nn.dynamic_rnn(lstm_cell, out, dtype=tf.float32)

    logits_1 = tf.layers.dense(outputs, units=2, activation=None,
kernel initializer=initialize)
    loss 1 = tf.nn.sparse softmax cross entropy with logits(labels=binary labels,
logits=logits_1)

    mask 1 = tf.equal(tf.argmax(logits 1, axis=-1), binary labels)

    mask = tf.cumprod(tf.cast(mask_1, dtype=tf.float32), axis=-1, exclusive=True)

    loss_1 = mask * loss_1

    loss 1 = tf.reduce mean(loss 1)
    predictions 1 = tf.reduce sum(tf.argmax(logits 1, axis=-1) * [32, 16, 8, 4, 2, 1], axis=-
1)

    precision_1 = tf.reduce_mean(tf.cast(tf.equal(predictions_1, labels), tf.float32))

with tf.name_scope('logistic_regression'):

    logits_2 = tf.layers.dense(inputs=inputs, units=num_classes, activation=None,
kernel_initializer=initialize)

    loss 2 = tf.reduce mean(tf.nn.sparse softmax cross entropy with logits(labels=labels,
logits=logits 2))

    predictions_2 = tf.argmax(logits_2, axis=-1)

    precision_2 = tf.reduce_mean(tf.cast(tf.equal(predictions_2, labels), tf.float32))

with tf.name_scope('linear_regression'):

    predictions_3 = tf.layers.dense(inputs=inputs, units=1, activation=None,
kernel_initializer=initialize)

    loss 3 = tf.losses.mean squared error(labels=labels,
predictions=tf.squeeze(predictions_3))

    predictions_3 = tf.cast(tf.round(predictions_3), tf.int64)

    precision 3 = tf.reduce mean(tf.cast(tf.equal(predictions 3, labels), tf.float32))

with tf.name_scope('unimodal_regression'):
```

```
    hidden_4 = tf.layers.dense(inputs=inputs, units=10, activation=tf.nn.leaky_relu,
kernel_initializer=initialize)

    pre_logits_4 = tf.layers.dense(inputs=hidden_4, units=1, activation=tf.nn.sigmoid,
kernel_initializer=initialize)

    loss_4, out_4 = full_unimodal(pre_logits_4, labels=labels, h_param={'temp': 1.,
'num_classes': num_classes})

    loss_4 = tf.reduce_mean(loss_4)

    predictions_4 = tf.argmax(out_4, axis=-1)

    precision_4 = tf.reduce_mean(tf.cast(tf.equal(predictions_4, labels), tf.float32))

with tf.name_scope('unmasked_binary_search'):

    hidden_binary_5 = tf.layers.dense(inputs=inputs, units=1, activation=None,
kernel_initializer=initialize)
    out_5 = tf.expand_dims(tf.tile(hidden_binary_5, [1, 6]), axis=-1)

    lstm_cell_5 = tf.nn.rnn_cell.LSTMCell(forget_bias=1.0, state_is_tuple=True,
activation=tf.nn.tanh, num_units=6, name='second_lstm')

    outputs_5, state_5 = tf.nn.dynamic_rnn(lstm_cell_5, out_5, dtype=tf.float32)

    logits_5 = tf.layers.dense(outputs_5, units=2, activation=None,
kernel_initializer=initialize)
    loss_5 = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=binary_labels,
logits=logits_5)
    loss_5 = tf.reduce_mean(loss_5)

    predictions_5 = tf.reduce_sum(tf.argmax(logits_5, axis=-1) * [32, 16, 8, 4, 2, 1], axis=-
1)

    precision_5 = tf.reduce_mean(tf.cast(tf.equal(predictions_5, labels), tf.float32))


def get_binary_indices(num_classes, index):
    return (np.arange(num_classes)//index) % 2


loss = loss_1 + loss_2 + loss_3 + loss_4 + loss_5 #+ loss_6 + loss_7
predictions = [predictions_1, predictions_2, predictions_3, predictions_4, predictions_5]#,
predictions_6, predictions_7]
accuracy = [precision_1, precision_2, precision_3, precision_4, precision_5]#, precision_6,
precision_7]

train_step = tf.train.AdamOptimizer(0.01).minimize(loss)

sess = tf.Session()
init = tf.global_variables_initializer()

total_accuracy = []
total_kappa = []

for i in range(0, 1000):
    sess.run(init)

    x, y, x_te, y_te = generate_data(size=num_classes, num_per_group=group_sample_size)

    classifier = mord.LogisticAT()
    classifier.fit(x, y)
    mord_pred = classifier.predict(x_te)

    for i in range(10000):

        start_time = time.time()
         , loss_value = sess.run([train_step, loss], feed_dict={inputs: x, labels: y})

        if i%1000 == 0:
            print(loss_value)
            print('Step ' + str(i) + ': loss = ' + str(loss_value))
            predictions_values = sess.run(predictions, feed_dict={inputs: x_te, labels: y_te})
            validation_accuracy = []
            std_dev = []
```

```
            kappa = []
            for i in range(len(predictions_values)):
                predictions values[i] = np.squeeze(predictions values[i])
                validation accuracy.append(np.mean(1 * predictions values[i] == y te))
                std_dev.append(np.std(1 * predictions_values[i] == y_te))
                kappa.append(cohen_kappa_score(predictions_values[i], y_te,
weights='quadratic'))

            print('accuracy ' + str(validation_accuracy))
            print('std_dev ' + str(std_dev))
            print('kappa    ' + str(kappa))
            print('mord accuracy ' + str(np.mean(1 * (mord_pred == y_te))))
            print('mord std_dev' + str(np.std(1 * (mord_pred == y_te))))
            print('mord kappa ' + str(cohen_kappa_score(mord_pred, y_te,
weights='quadratic')))
            total_kappa.append(kappa)
            total_accuracy.append(kappa)
            np.save('predictions_values', predictions_values)

    predictions values = sess.run(predictions, feed dict={inputs: x te, labels: y te})

    validation_accuracy = []
    kappa = []

    for i in range(len(predictions_values)):
        validation accuracy.append(np.mean(1 * predictions values[i] == y te))
        kappa.append(cohen_kappa_score(predictions_values[i], y_te, weights='quadratic'))

    print('accuracy ' + str(validation_accuracy))
    print('kappa    ' + str(kappa))
    total kappa.append(kappa)
    total accuracy.append(kappa)

np.save('experiment_kappa_values', total_kappa)
np.save('experiment_accuracy_values', total_accuracy)
np.save('predictions_values', predictions_values)
```

7.3 Voice analysis and suicide intent detection experiment code

7.3.1 Data preprocessing code

```
import pickle
import numpy as np
import csv
import data.audio.audio_process as audio


def get_labels_from_csv(filename):
    """reads the csv file containing evaluations and converts it into 4 arrays containing
every evaluation"""

    caller = []
    counselor = []
    counselor_context = []
    caller_context = []
    data parse = []
    i = 0
    with open(filename, newline='') as csv_file:
        read = csv.reader(csv_file, delimiter=',')
        for row in read:
            if (i + 1) % 5 != 0:
                data_parse.append([int(x) for x in row if x != ''])
            i += 1

    for i in range(int(len(data_parse) / 4)):
        caller.append(data parse[4 * i])
        caller context.append(data parse[4 * i + 1])
        counselor.append(data parse[4 * i + 2])
        counselor_context.append(data_parse[4 * i + 3])
```

```
    return caller, counselor, caller_context, counselor_context


def create_sliced_dataset(data):
    """shapes the dataset into an array where every slice is accessible
    input : truncated dataset of shape (num of calls, truncated length of second, num
features)
    outputs : reorganized dataset of shape (num of calls, number of slices per call, length of
slice, num features"""
    sliced = []
    for dat in data:
        sliced.append(np.array(np.split(dat, len(dat)//1200)))

    print("--- slicing done---")
    return np.array(sliced)


def compute_moments(data):
    """flatten dataset and compute variables moments per slices
    input : truncated dataset of shape (num of calls, numb of slices, slice length, num
features)
    outputs : mean and std dev of shape (2, 64)"""
    flat = data[0]
    for dat in data[1:]:
        flat = np.concatenate((flat, dat), axis=0)
    print(flat.shape)
    print("--- flattening done --- ")

    mean = np.mean(flat, axis=(0, 1))
    var = np.var(flat, axis=(0, 1))

    print("--- moment computation done ---")
    return mean, np.sqrt(var)


def extract_audio_features(feature_type):

    if feature_type == 'opensmile':
        feature_procedure = audio.extract_features_open_smile
    if feature_type == 'pyaudio':
        feature procedure = audio.extract features pyaudioanalysis

    labels = get_labels_from_csv('labels_data.csv')

    for i in range(1, 60):
        print('beginning extraction for recording' + str(i))
        feature = feature_procedure('audio/recordings/' + str(i) + '_caller_perfect.wav',
                                                              len(labels[0][i - 1]))
        np.save(feature_type + '_features/caller_' + str(i) + '_features_opensmile', feature)
        #np.save('pyaudio_features/counselor_' + str(i) + '_features.csv', feature[0])
```

7.3.2 Data processing for local model code

```
import numpy as np
import csv
import time


missing = np.arange(59, 72)

present = np.delete(np.arange(0, 90), missing)

true_validation_ind = [3, 14, 11, 30, 44, 50, 48, 73, 74] #[4, 15, 10, 31, 45, 51, 49, 74, 75]
true_test_ind = [5, 18, 47, 36, 32, 55, 85, 87, 89]

validation ind = [3, 14, 11, 30, 44, 50, 48, 73-13, 74-13]
test_ind = [4, 15, 10, 31, 45, 51, 49, 74 - 13, 75 - 13]  # [5, 18, 47, 36, 32, 55, 85-13, 87-
13, 89-13]

training_ind = np.arange(0, 90-13)

non training = np.concatenate((validation ind, test ind))
```

```
training_ind = np.delete(training_ind, non_training, axis=0)


def get_feature_slices(slices, data, training):

    if training:
        is_end_or_beginning = np.logical_not(np.logical_or(np.isin(slices,
data['end_rec_indices']), np.isin(slices, data['end_rec_indices'] + 1)))

        offset_value = np.random.uniform(0, 1, size=len(slices)) * is end or beginning *
(0/30) * data['slice_length']

        feature_slices = np.array([np.arange(begin * data['slice_length'] + off,
                                             (begin + 1) * data['slice_length'] + off)
                                   for begin, off in zip(slices,
offset_value.astype(np.int32))])

    else:
        feature_slices = np.array([np.arange(begin * data['slice_length'], (begin + 1) *
data['slice_length'])
                                   for begin in slices])

    return feature_slices


def next_semisup_batch(data_sup, data_unsup, step, epoch, training, batch_size,
unsup_batch_size):

    next sup, new step, new epoch = next sup batch(data sup, step, epoch, training,
batch size)

    next unsup = next unsup batch(data unsup, unsup batch size)

    return next_sup, next_unsup, new_step, new_epoch


def next_sup_batch(data, step, epoch, training, batch_size):

    if step == 0 and training:
        np.random.shuffle(data['indices'])

    length = len(data['indices'])

    new_step = step + batch_size if step + batch_size < length else length

    feature_slices = get_feature_slices(data['indices'][step:new_step], data, training)


    next_batch = {'features': data['features'][feature_slices],
                  'labels': np.maximum(data['labels'][data['indices'][step:new_step]] - 1, 0),
                  'psych': data['psych'][data['indices'][step:new_step]],
                  'bin': 1 * (data['labels'][data['indices'][step:new_step]] > 0)}

    if new_step % length == 0:
        epoch += 1

    return next_batch, new_step % length, epoch


def next_unsup_batch(unsup, batch_size):

    indices = np.random.randint(0, len(unsup['features']), size=batch_size)
    slices = (np.random.uniform(0, 1, size=batch_size) * (unsup['lengths'][indices] -
unsup['slice_length'])).astype(np.int32)

    batch = []

    for i in range(batch_size):
        batch.append(unsup['features'][indices[i]][slices[i]:slices[i] +
unsup['slice_length']])

    return batch


def get_labels_from_csv(filename):
    """reads the csv file containing evaluations and converts it into 4 arrays containing
every evaluation"""
```

```
    caller = []
    counselor = []
    counselor context = []
    caller_context = []
    data_parse = []
    i = 0
    with open(filename, newline='') as csv_file:
        read = csv.reader(csv_file, delimiter=',')
        for row in read:
            if (i + 1) % 5 != 0:
                data_parse.append(np.array([int(x) for x in row if x != '']))
            i += 1

    for i in range(int(len(data_parse) / 4)):
        caller.append(data_parse[4 * i])
        caller_context.append(data_parse[4 * i + 1])
        counselor.append(data_parse[4 * i + 2])
        counselor_context.append(data_parse[4 * i + 3])

    return np.array(caller)[present], np.array(counselor)[present], \
           np.array(caller_context)[present], np.array(counselor_context)[present]


def get_labels(call_or_counsel='caller'):

    ca, co, caco, coco = get_labels_from_csv('labels_data.csv')

    if call_or_counsel == 'caller':
        labels = ca

    else:
        labels = co

    return np.array(labels)


def create_psych_info_from_labels(labels):

    psych = []

    for lab in labels[:25]:
        psych.append(np.array([[0, 0, 0, 1]] * len(lab)))

    for lab in labels[25:47]:
        psych.append(np.array([[0, 0, 1, 0]] * len(lab)))

    for lab in labels[47:59]:
        psych.append(np.array([[0, 1, 0, 0]] * len(lab)))

    for lab in labels[59:]:
        psych.append(np.array([[0, 1, 0, 0]] * len(lab)))

    return np.array(psych)


def input_normalize(data, moments):
    normalized = (np.concatenate(data, axis=0) - moments[0][:32]) / moments[1][:32]
    return normalized


def load_dataset(py_audio=True, call_or_counsel='caller', factor_value=7):

    print('   ---- Beginning dataset loading ----   ')
    slice length = 1200
    if py_audio:
        features = []
        features_low = []
        features_high = []
        moments_file = 'data/input_moments_pyaudio.npy'
        for i in present:
            features.append(np.transpose(
                np.load('data/pyaudio_features/caller_' + str(i + 1) + '_features.npy',
                        mmap_mode='r')))
            features low.append(np.transpose(
                np.load('data/opensmile_features/caller_' + str(i + 1) + '_opensmile_low.npy',
                        mmap_mode='r')))
```

```
            features_high.append(np.transpose(
                np.load('data/opensmile_features/caller_' + str(i + 1) +
'_opensmile_high.npy',
                        mmap_mode='r')))
    else:
        features = []
        features_low = []
        features_high = []
        moments_file = 'data/input_moments_opensmile.npy'
        for i in present:
            features.append(np.transpose(
                np.load('data/opensmile_features/caller_' + str(i + 1) + '_opensmile.npy',
                        mmap_mode='r')[:32, :]))
            features_low.append(np.transpose(
                np.load('data/opensmile_features/caller_' + str(i + 1) + '_opensmile_low.npy',
                        mmap_mode='r')[:32, :]))
            features_high.append(np.transpose(
                np.load('data/opensmile_features/caller_' + str(i + 1) +
'_opensmile_high.npy',
                        mmap_mode='r')[:32, :]))

    #features = np.concatenate((features, features_low, features_high))
    features = np.array(features)
    #augmented_training_ind = np.concatenate((training_ind, training_ind + len(present),
training_ind + 2 * len(present)))

    labels = get_labels(call_or_counsel)
    #training_labels = np.concatenate((labels[training_ind], labels[training_ind],
labels[training_ind]))

    psych = create_psych_info_from_labels(labels)

    cum_len = np.cumsum([len(x) for x in psych[training_ind]])
    #cum_len = np.concatenate((cum_len, cum_len + cum_len[-1], cum_len + 2 * cum_len[-1]))

    #flat_psych = np.concatenate(np.concatenate((psych[training_ind], psych[training_ind],
psych[training_ind])))
    flat_psych = np.concatenate((psych[training_ind]))

    moments = np.load(moments_file)

    training_data = {'features': input_normalize(features[training_ind], moments),
                     'labels': np.minimum(np.concatenate(labels), factor_value),
                     'psych': flat_psych,
                     'indices': np.arange(0, len(flat_psych)),
                     'slice_length': slice_length,
                     'end_rec_indices': np.concatenate(([0], cum_len-1))}

    flat_psych = np.concatenate(psych[validation_ind])
    cum_len = np.cumsum([len(x) for x in psych[validation_ind]])

    validation_data = {'features': input_normalize(features[validation_ind], moments),
                       'labels': np.minimum(np.concatenate(labels[validation_ind]),
factor_value),
                       'psych': flat_psych,
                       'indices': np.arange(0, len(flat_psych)),
                       'slice_length': slice_length,
                       'end_rec_indices': np.concatenate(([0], cum_len - 1))}

    flat_psych = np.concatenate(psych[test_ind])
    cum_len = np.cumsum([len(x) for x in psych[test_ind]])

    test_data = {'features': input_normalize(features[test_ind], moments),
                 'labels': np.minimum(np.concatenate(labels[test_ind]), factor_value),
                 'psych': flat_psych,
                 'indices': np.arange(0, len(flat_psych)),
                 'slice_length': slice_length,
                 'end_rec_indices': np.concatenate(([0], cum_len - 1))}

    print('   ---- Dataset loaded ----   ')

    return training_data, validation_data, test_data


def load_fake_dataset(call_or_counsel='caller', factor_value=7):

    labels = get_labels(call_or_counsel)
```

```
    data = []

    slice_length = 1200

    for seq in labels:
        seq_parse = [np.random.normal(0, 0.2, size=[1200, 34]) +
np.concatenate((np.ones(shape=[1200,1]), np.zeros(shape=[1200, 33])), axis=-1) * y for y in
seq]
        data.append(np.concatenate(seq_parse, axis=0) / 7)

    features = np.array(data)

    psych = create_psych_info_from_labels(labels)

    cum_len = np.cumsum([len(x) for x in psych[training_ind]])

    flat_psych = np.concatenate(psych[training_ind])

    training_data = {'features': np.concatenate(features[training_ind], axis=0),
                     'labels': np.minimum(np.concatenate(labels[training_ind]), factor_value),
                     'psych': flat_psych,
                     'indices': np.arange(0, len(flat_psych)),
                     'slice_length': slice_length,
                     'end_rec_indices': np.concatenate(([0], cum_len - 1))}

    flat_psych = np.concatenate(psych[validation_ind])
    cum_len = np.cumsum([len(x) for x in psych[validation_ind]])

    validation_data = {'features': np.concatenate(features[validation_ind], axis=0),
                       'labels': np.minimum(np.concatenate(labels[validation_ind]),
factor value),
                       'psych': flat_psych,
                       'indices': np.arange(0, len(flat_psych)),
                       'slice_length': slice_length,
                       'end_rec_indices': np.concatenate(([0], cum_len - 1))}

    flat psych = np.concatenate(psych[test ind])
    cum_len = np.cumsum([len(x) for x in psych[test_ind]])

    test_data = {'features': np.concatenate(features[test_ind], axis=0),
                 'labels': np.minimum(np.concatenate(labels[test_ind]), factor_value),
                 'psych': flat_psych,
                 'indices': np.arange(0, len(flat_psych)),
                 'slice_length': slice_length,
                 'end_rec_indices': np.concatenate(([0], cum_len - 1))}

    print('   ---- Dataset loaded ----   ')

    return training_data, validation_data, test_data


def load_unsup_dataset(py_audio=True):
    feat = []
    lengths = []
    slice_length = 1200

    if py_audio:
        for i in range(0, 309):
            feat.append(np.load('data/pyaudio_features_unsup/caller_unsup_' + str(i) + '.npy',
mmap_mode='r'))
            lengths.append(len(feat[-1]))

    else:
        for i in range(1, 310):
            feat.append(np.load('data/opensmile_features_unsup/caller_unsup_' + str(i) +
'_opensmile.npy', mmap_mode='r')[:, :32])
            lengths.append(len(feat[-1]))

    return {'features': np.array(feat), 'lengths': np.array(lengths), 'slice_length':
slice_length}


def get_binary_search_labels(labels, h_param):

    search_size = np.ceil(np.log2(h_param['num_classes'])).astype(np.int32)
    val_parse = labels
```

```
    binary_search_labels = []

    for i in reversed(range(search_size)):
        val parse = (val parse % np.power(2, i + 1))
        binary_search_labels.append(val_parse // np.power(2, i))

    return np.stack(binary_search_labels, axis=1)
```

7.3.3 Ladder model code

```
import tensorflow as tf
import numpy as np
import models.losses as losses
import time
from sklearn.metrics import cohen_kappa_score, mean_squared_error


def combinator(inputs, side, name, reuse):

    stacked = tf.stack([inputs, side, tf.multiply(inputs, side)], axis=-1)

    hidden = tf.layers.dense(stacked, 4,
                             activation=tf.nn.leaky relu,
                             kernel initializer=tf.random normal initializer(mean=0,
stddev=0.025),
                             name=name + '_hidden',
                             reuse=reuse)

    out = tf.layers.dense(hidden, 1,
                          activation=None,
                          kernel_initializer=tf.random_normal_initializer(mean=0,
stddev=0.025),
                          name=name + '_out',
                          reuse=reuse)

    return tf.squeeze(out, axis=-1)


def get_loss_initial(noisy_logits, psych, labels, h_param, reuse=None):

    ordinal, binary = tf.split(noisy logits, num or size splits=[tf.shape(noisy logits)[-1] -
2, 2], axis=-1)

    with tf.name_scope('binary_loss'):

        binary loss =
tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(labels=labels['binary'],

logits=binary))

    with tf.name_scope('ordinal_loss'):

        ordinal_split = tf.split(ordinal, num_or_size_splits=4, axis=-1)
        loss_tab = []

        if h_param['loss_type'] == 'logistic':

            for i in range(4):

loss_tab.append(tf.nn.sparse_softmax_cross_entropy_with_logits(labels=labels['ordinal'],

logits=ordinal split[i])
                                * tf.cast(labels['binary'], dtype=tf.float32))
            out = ordinal_split

        else:
            out = []

            for i in range(4):
                reuse_val = (i == 0 and reuse) or (i > 0 or reuse)
```

```
                loss_parse, out_parse = losses.full_unimodal(ordinal_split[i], labels,
h param, reuse val)

                loss tab.append(loss parse)
                out.append(out_parse)

        ordinal_loss = tf.reduce_mean(tf.stack(loss_tab, axis=-1) * psych)

    return ordinal_loss + binary_loss, [out, binary]


def get_loss(noisy_logits, psych, labels, h_param, initializer, reuse, is_training):

    if h_param['loss_type'] == 'logistic':
        logits = tf.layers.dense(noisy_logits, h_param['num_classes'] * 4,
kernel_initializer=initializer,
                                 name='logits', reuse=reuse)
        logits = tf.reshape(logits, shape=[-1, 4, h_param['num_classes']])

    else:
        logits = tf.layers.dense(noisy logits, 4, kernel initializer=initializer,
name='logits', reuse=reuse)
        logits = tf.expand_dims(logits, axis=-1)

    with tf.name_scope('ordinal_loss'):

        if h_param['loss_type'] == 'logistic':

            ordinal loss =
tf.nn.sparse_softmax_cross_entropy_with_logits(labels=labels['ordinal'], logits=logits)

            ordinal loss = tf.reduce mean(tf.reduce sum(ordinal loss * psych, axis=-1)
                                          * tf.cast(labels['binary'], dtype=tf.float32),
axis=-1)

            out = logits

        elif h_param['loss_type'] == 'unimodal':

            ordinal_loss, out = losses.full_unimodal(logits=logits, labels=labels,
h param=h param, reuse=reuse)

            ordinal loss = tf.reduce mean(tf.reduce sum(ordinal loss * psych, axis=-1)
                                          * tf.cast(labels['binary'], dtype=tf.float32),
axis=-1)

        elif h_param['loss_type'] == 'binary_search':

            ordinal_loss, out = losses.binary_search(logits=logits, labels=labels,
initializer=initializer,
                                                     h_param=h_param,
                                                     reuse=reuse)

            ordinal loss = tf.reduce mean(tf.reduce sum(ordinal loss * psych, axis=-1)
                                          * tf.cast(labels['binary'], dtype=tf.float32),
axis=-1)

        else:

            ordinal loss = tf.losses.mean squared error(predictions=tf.squeeze(logits),
labels=labels['ordinal'],
                                                        reduction=tf.losses.Reduction.NONE)


            ordinal loss = tf.reduce sum(tf.reduce sum(ordinal loss * tf.minimum(psych + 0.1,
1), axis=-1)
                                         * labels['loss_weights']
                                         * tf.cast(labels['binary'], dtype=tf.float32))
            print(ordinal_loss)

            out = logits
            print(out)

    return ordinal_loss, out #'+ binary_loss'


def get_loss_v2(noisy_logits, psych, labels, h_param, initializer, reuse, is_training):
```

```
    ordinal = tf.concat([noisy logits, psych], axis=-1)

    hidden_ordinal = tf.layers.dense(ordinal, units=h_param['linear'],
                                     activation=tf.nn.leaky_relu,
                                     kernel_initializer=initializer,
                                     name='hidden_ordinal',
                                     reuse=reuse)

    hidden ordinal = tf.layers.batch normalization(hidden ordinal,
momentum=h_param['momentum'], training=is_training,
                                                   name='final_batch_norm', reuse=reuse)

    if h_param['loss_type'] == 'logistic':
        logits = tf.layers.dense(hidden_ordinal, h_param['num_classes'],
kernel_initializer=initializer,
                                 name='logits', reuse=reuse)
    else:
        logits = tf.layers.dense(hidden_ordinal, 1, kernel_initializer=initializer,
name='logits', reuse=reuse)

    with tf.name_scope('ordinal_loss'):

        if h_param['loss_type'] == 'logistic':

            ordinal_loss = tf.losses.sparse_softmax_cross_entropy(labels=labels['ordinal'],
logits=logits,

weights=labels['loss_weights'])

            ordinal_loss = tf.reduce_mean(ordinal_loss * tf.cast(labels['binary'],
dtype=tf.float32), axis=-1)

            out = logits

        elif h_param['loss_type'] == 'unimodal':

            ordinal loss, out = losses.full unimodal(logits=logits, labels=labels,
h_param=h_param, reuse=reuse)

            ordinal_loss = tf.reduce_mean(ordinal_loss * tf.cast(labels['binary'],
dtype=tf.float32), axis=-1)

        elif h_param['loss_type'] == 'binary_search':

            ordinal_loss, out = losses.binary_search_v2(logits=logits, labels=labels,
initializer=initializer,
                                                        h param=h param,
                                                        reuse=reuse)

            ordinal_loss = tf.reduce_mean(ordinal_loss * tf.cast(labels['binary'],
dtype=tf.float32), axis=-1)

        else:

            ordinal_loss = tf.losses.mean_squared_error(predictions=tf.squeeze(logits),
labels=labels['ordinal'],
                                                        weights=labels['loss_weights'])
            out = logits

    return ordinal_loss, out


def get_deconv_parameters(inputs, desired_outputs, kernel):

    out_shape = desired_outputs.get_shape().as_list()[1]
    in_shape = inputs.get_shape().as_list()[1]
    strides = out_shape // in_shape
    mod = out_shape % in_shape
    if mod == 0:
        kernel_size = np.minimum(strides, kernel)
    else:
        kernel_size = mod + strides

    return strides, kernel_size
```

```
def denoise_feed_layer(inputs, side, size, initializer, is_training, momentum, name, reuse):

    hidden = tf.layers.dense(inputs, size,
                             activation=None,
                             kernel_initializer=initializer,
                             name=name + '_hidden',
                             reuse=reuse)

    batch norm = tf.layers.batch normalization(hidden, axis=-1,
                                               training=is training,
                                               momentum=momentum,
                                               center=False,
                                               scale=False,
                                               name=name + '_batch_norm',
                                               reuse=reuse)

    out = combinator(batch_norm, side, name=name + 'combinator', reuse=reuse)

    return out


def denoise_conv(inputs, side, filters, initializer, is_training, momentum, reuse, name,
kernel):

    strides, kernel size = get deconv parameters(inputs, side, kernel)

    deconv = tf.layers.conv2d_transpose(tf.expand_dims(inputs, axis=1), filters,
                                        kernel_size=[1, kernel_size],
                                        strides=[1, strides],
                                        kernel initializer=initializer,
                                        padding='valid',
                                        name=name + '_deconv',
                                        reuse=reuse)

    batch norm = tf.layers.batch normalization(deconv, axis=-1,
                                               training=is training,
                                               momentum=momentum,
                                               center=False,
                                               scale=False,
                                               name=name + '_batch_norm',
                                               reuse=reuse)

    out = combinator(tf.squeeze(batch_norm, axis=1), side, name=name + '_combinator',
reuse=reuse)

    return out


def tcn_block_v1(inputs, filters, activation, initializer, is_training, momentum, name, reuse,
noise_std):

    with tf.name_scope("tcn_block"):
        with tf.name_scope("conv_1"):
            conv = tf.layers.conv1d(inputs, filters=filters[0],
                                    kernel_size=3,
                                    strides=2,
                                    padding='SAME',
                                    activation=None,
                                    kernel initializer=initializer,
                                    name=name + '_block_1',
                                    reuse=reuse)

            batch_norm = tf.layers.batch_normalization(conv, axis=-1,
                                                       training=is training,
                                                       momentum=momentum,
                                                       center=False,
                                                       scale=False,
                                                       name=name + '_batch_norm_1',
                                                       reuse=reuse)

            batch_norm += tf.random_normal(shape=tf.shape(conv)) * noise_std

            with tf.variable_scope(name + '_rescale_1', reuse=reuse):
                gamma = tf.get_variable(name + "_gamma_1",
initializer=tf.ones(shape=[filters[0]]))
                beta = tf.get_variable(name + "_beta_1",
initializer=tf.zeros(shape=[filters[0]]))
```

```
            out 1 = activation(tf.multiply(gamma, batch norm) + beta)

        with tf.name_scope("conv_2"):
            conv = tf.layers.conv1d(out_1, filters=filters[1],
                                    kernel_size=3,
                                    strides=2,
                                    padding='SAME',
                                    activation=None,
                                    kernel initializer=initializer,
                                    name=name + '_block_2',
                                    reuse=reuse)

    with tf.name_scope("shortcut"):
        shortcut = tf.layers.conv1d(inputs, filters=filters[1],
                                    kernel_size=1,
                                    strides=4,
                                    padding='SAME',
                                    activation=None,
                                    kernel initializer=initializer,
                                    name=name + '_shortcut',
                                    reuse=reuse)

    batch norm 2 = tf.layers.batch normalization(conv + shortcut, axis=-1,
                                                 training=is training,
                                                 momentum=momentum,
                                                 center=False,
                                                 scale=False,
                                                 name=name + '_batch_norm_2',
                                                 reuse=reuse)

    batch norm 2 += tf.random normal(shape=tf.shape(conv)) * noise std

    with tf.variable_scope(name + '_rescale_2', reuse=reuse):
        gamma = tf.get_variable(name + "_gamma_2", initializer=tf.ones(shape=[filters[1]]))
        beta = tf.get_variable(name + "_beta_2", initializer=tf.zeros(shape=[filters[1]]))

    out = activation(tf.multiply(gamma, batch norm 2) + beta)

    return out, [batch_norm, batch_norm_2]


def tcn_block(inputs, filters, stride, activation, initializer, is_training, momentum, name,
reuse, noise_std):

    with tf.name_scope("tcn_block"):
        with tf.name_scope("conv_1"):
            conv = tf.layers.conv1d(inputs, filters=filters[0],
                                    kernel_size=[stride+1],
                                    strides=[stride],
                                    padding='SAME',
                                    activation=None,
                                    kernel initializer=initializer,
                                    name=name + '_block_1',
                                    reuse=reuse)

            batch_norm = tf.layers.batch_normalization(conv, axis=-1,
                                                       training=is training,
                                                       momentum=momentum,
                                                       center=False,
                                                       scale=False,
                                                       name=name + '_batch_norm_1',
                                                       reuse=reuse)

            batch norm += tf.random normal(shape=tf.shape(conv)) * noise std

            with tf.variable_scope(name + '_rescale_1', reuse=reuse):
                gamma = tf.get_variable(name + "_gamma_1",
initializer=tf.ones(shape=[filters[0]]))
                beta = tf.get_variable(name + "_beta_1",
initializer=tf.zeros(shape=[filters[0]]))

            out_1 = activation(tf.multiply(gamma, batch_norm) + beta)

        with tf.name_scope("conv_2"):
            conv = tf.layers.conv1d(out 1, filters=filters[1],
                                    kernel_size=[stride+1],
```

```
                                    strides=[stride],
                                    padding='SAME',
                                    activation=None,
                                    kernel initializer=initializer,
                                    name=name + '_block_2',
                                    reuse=reuse)

    with tf.name_scope("shortcut"):
        shortcut = tf.layers.conv1d(inputs, filters=filters[1],
                                    kernel size=1,
                                    strides=[stride*stride],
                                    padding='SAME',
                                    activation=None,
                                    kernel initializer=initializer,
                                    name=name + '_shortcut',
                                    reuse=reuse)

    batch_norm_2 = tf.layers.batch_normalization(conv, axis=-1,
                                                 training=is_training,
                                                 momentum=momentum,
                                                 center=False,
                                                 scale=False,
                                                 name=name + '_batch_norm_2',
                                                 reuse=reuse)

    batch norm 2 += tf.random normal(shape=tf.shape(conv)) * noise std

    with tf.variable_scope(name + '_rescale_2', reuse=reuse):
        gamma = tf.get_variable(name + "_gamma_2", initializer=tf.ones(shape=[filters[1]]))
        beta = tf.get_variable(name + "_beta_2", initializer=tf.zeros(shape=[filters[1]]))

    out = activation(tf.multiply(gamma, batch norm 2) + beta)

    out = tf.layers.batch_normalization(out + shortcut, axis=-1,
                                        training=is training,
                                        momentum=momentum,
                                        center=False,
                                        scale=False,
                                        name=name + '_batch_norm_3',
                                        reuse=reuse)

    return out, [batch_norm, batch_norm_2]


def denoise_tcn(inputs, sides, filters, initializer, is_training, momentum, reuse, name,
kernel=3):

    out 1 = denoise conv(inputs,
                         side=sides[1],
                         filters=filters[1],
                         initializer=initializer,
                         is training=is training,
                         momentum=momentum,
                         reuse=reuse,
                         name=name+'_block_1',
                         kernel=kernel)

    out 2 = denoise conv(out 1,
                         side=sides[0],
                         filters=filters[0],
                         initializer=initializer,
                         is_training=is_training,
                         momentum=momentum,
                         reuse=reuse,
                         name=name + '_block_2',
                         kernel=kernel)

    return [out_2, out_1]


def tcn_encoder_semi_sup(inputs, is_training, h_param, initializer, reuse, noise_std):

    norm_out = []
    conv_out = inputs + tf.random_normal(shape=tf.shape(inputs)) * noise_std

    for i in range(len(h_param['filters'])):
```

```
        with tf.name_scope('tcn_block_' + str(i)):

            conv out, batch norm = tcn block(conv out,
                                             filters=h_param['filters'][i],
                                             stride=h_param['stride'],
                                             activation=tf.nn.leaky_relu,
                                             initializer=initializer,
                                             is training=is training,
                                             momentum=h_param['momentum'],
                                             name="tcn_" + str(i),
                                             reuse=reuse,
                                             noise_std=noise_std)

            norm out.append(batch norm)

    feed = tf.layers.flatten(conv_out)

    return feed, norm_out


def tcn_encoder_semi_sup_v2(inputs, is_training, h_param, initializer, reuse, noise_std):

    norm_out = []
    conv out = inputs + tf.random normal(shape=tf.shape(inputs)) * noise std

    for i in range(len(h_param['filters'])):

        with tf.name_scope('tcn_block_' + str(i)):

            conv out, batch norm = tcn block(conv out,
                                             filters=h_param['filters'][i],
                                             stride=h_param['stride'],
                                             activation=tf.nn.leaky_relu,
                                             initializer=initializer,
                                             is training=is training,
                                             momentum=h_param['momentum'],
                                             name="tcn_" + str(i),
                                             reuse=reuse,
                                             noise_std=noise_std)

            norm out.append(batch norm)

    feed = tf.layers.flatten(conv out)

    return feed, norm_out


def tcn_decoder_semi_sup(logits, noisy, clean, is_training, h_param, initializer, reuse):
    loss = []
    layer_count = -1
    denoise = [tf.expand_dims(logits, axis=-2), 0]

    for i in reversed(range(0, len(h_param['filters']))):
        with tf.name_scope("denoising_layer_" + str(len(h_param['filters']) - i)):

            denoise = denoise_tcn(denoise[0], noisy[layer_count],
                                  filters=h_param['filters'][i],
                                  initializer=initializer,
                                  is training=is training,
                                  momentum=h_param['momentum'],
                                  name='denoise_conv_' + str(len(h_param['filters']) - i),
                                  reuse=reuse)

            loss.append(tf.multiply(tf.losses.mean squared error(clean[layer count][0],
denoise[0]),
                                    h_param['denoise_conv'][i][0]))
            loss.append(tf.multiply(tf.losses.mean_squared_error(clean[layer_count][1],
denoise[1]),
                                    h_param['denoise_conv'][i][1]))
            layer count -= 1

    return loss


def model_semisup(inputs, unsup_inputs, psych, labels, is_training, h_param):

    initialize = tf.variance_scaling_initializer(scale=1.0, distribution='normal')
```

```
    if h_param['unsup_batch_size'] > 0:
        everyone = tf.concat((inputs, unsup inputs), axis=0)
    else:
        everyone = inputs

    clean_logits, clean_norm_out = tcn_encoder_semi_sup(everyone,
                                                        is training=is training,
                                                        h param=h param,
                                                        initializer=initialize,
                                                        reuse=None,
                                                        noise_std=0.0)

    noisy logits, noisy norm out = tcn encoder semi sup(everyone,
                                                        is training=is training,
                                                        h_param=h_param,
                                                        initializer=initialize,
                                                        reuse=True,
                                                        noise_std=h_param['noise_std'])

    unsup losses = tcn decoder semi sup(noisy logits,
                                        noisy=noisy_norm_out,
                                        clean=clean_norm_out,
                                        is training=is training,
                                        h param=h param,
                                        initializer=initialize,
                                        reuse=None)

    if h_param['unsup_batch_size'] > 0:
        supervised clean,    = tf.split(clean logits, num or size splits=[tf.shape(inputs)[0],
tf.shape(unsup inputs)[0]],
                                        axis=0)
        supervised_noisy, _ = tf.split(noisy_logits, num_or_size_splits=[tf.shape(inputs)[0],
tf.shape(unsup_inputs)[0]],
                                        axis=0)
    else:
        supervised clean = clean logits
        supervised noisy = noisy logits

    sup_loss, out_noisy = get_loss(supervised_noisy, psych, labels, h_param, reuse=None,
initializer=initialize, is training=is training)
    _, out_clean = get_loss(supervised_clean, psych, labels, h_param, reuse=True,
initializer=initialize, is training=is training)

    return sup_loss, tf.reduce_sum(unsup_losses), out_clean


def model_semisup_v2(inputs, unsup_inputs, psych, labels, is_training, h_param):

    initialize = tf.variance_scaling_initializer(scale=1.0, distribution='normal')

    if h_param['unsup_batch_size'] > 0:
        everyone = tf.concat((inputs, unsup inputs), axis=0)
    else:
        everyone = inputs

    clean_logits, clean_norm_out = tcn_encoder_semi_sup_v2(everyone,
                                                           is training=is training,
                                                           h param=h param,
                                                           initializer=initialize,
                                                           reuse=None,
                                                           noise_std=0.0)

    noisy logits, noisy norm out = tcn encoder semi sup v2(everyone,
                                                           is training=is training,
                                                           h_param=h_param,
                                                           initializer=initialize,
                                                           reuse=True,
                                                           noise_std=h_param['noise_std'])

    unsup_losses = tcn_decoder_semi_sup(noisy_logits,
                                        noisy=noisy_norm_out,
                                        clean=clean_norm_out,
                                        is_training=is_training,
                                        h param=h param,
                                        initializer=initialize,
                                        reuse=None)
```

```
    if h_param['unsup_batch_size'] > 0:
        supervised clean,   = tf.split(clean logits, num or size splits=[tf.shape(inputs)[0],
tf.shape(unsup inputs)[0]],
                                       axis=0)
        supervised_noisy, _ = tf.split(noisy_logits, num_or_size_splits=[tf.shape(inputs)[0],
tf.shape(unsup_inputs)[0]],
                                       axis=0)
    else:
        supervised clean = clean logits
        supervised_noisy = noisy_logits

    sup_loss, out_noisy = get_loss_v2(supervised_noisy, psych, labels, h_param, reuse=None,
initializer=initialize, is training=is training)
    _, out_clean = get_loss_v2(supervised_clean, psych, labels, h_param, reuse=True,
initializer=initialize, is_training=is_training)

    return sup_loss, tf.reduce_sum(unsup_losses), out_clean



def get_binary_search_labels(labels, h_param):

    search_size = np.ceil(np.log2(h_param['num_classes'])).astype(np.int32)
    val parse = labels
    binary search labels = []

    for i in reversed(range(search_size)):
        val parse = (val parse % np.power(2, i + 1))
        binary search labels.append(val parse // np.power(2, i))

    return np.stack(binary_search_labels, axis=1)
```

7.3.4 Temporal convolutional network for global modelling problem code

```
import tensorflow as tf
import numpy as np
import models.losses_context as losses


def get_loss(logits, labels, lengths, h_param, reuse=None):

    if h_param['loss_type'] == 'logistic':

        loss, out = losses.logistic(logits, labels, lengths)

    else:

        loss, out = losses.unimodal(logits, labels, h param, reuse)

    return loss, out


def tcn_block(inputs, filters, dilation_power, activation, initializer, drop_prob,
is training, momentum, name):

    with tf.name_scope("tcn_block"):
        with tf.name_scope("first_tcn"):
            conv = tf.layers.conv1d(inputs, filters=filters[0],
                                    kernel_size=3,
                                    dilation rate=[np.power(2, dilation power)],
                                    padding='SAME',
                                    activation=activation,
                                    kernel_initializer=initializer,
                                    name=name+'_first_block')

            batch norm = tf.layers.batch normalization(conv, axis=-1,
                                                       training=is training,
                                                       momentum=momentum,
                                                       center=True,
                                                       scale=True)
```

```
        with tf.name_scope("second_tcn"):
            conv = tf.layers.conv1d(batch norm, filters=filters[1],
                                    kernel size=3,
                                    dilation rate=[np.power(2, dilation power+1)],
                                    padding='SAME',
                                    activation=activation,
                                    kernel_initializer=initializer,
                                    name=name + '_second_block')

            batch norm = tf.layers.batch normalization(conv, axis=-1,
                                                       training=is_training,
                                                       momentum=momentum,
                                                       center=True,
                                                       scale=True)

    with tf.name_scope("shortcut"):
        shortcut = tf.layers.conv1d(inputs, filters=filters[1],
                                    kernel_size=1,
                                    padding='SAME',
                                    activation=None,
                                    kernel initializer=initializer,
                                    name=name + '_shortcut')

    batch norm 2 = tf.layers.batch normalization(batch norm + shortcut, axis=-1,
                                                 training=is training,
                                                 momentum=momentum,
                                                 center=False,
                                                 scale=False)

    dropped = tf.layers.dropout(batch norm 2, rate=drop prob, training=is training)

    return dropped


def model(inputs, psych, lengths, labels, is_training, h_param):

    conv = tf.concat([inputs, psych], axis=-1)

    initialize = tf.variance_scaling_initializer(scale=1.0, distribution='normal')

    for i in range(len(h_param['filters'])):

        conv = tcn_block(conv, h_param['filters'][i],
                         dilation_power=2 * i,
                         activation=tf.nn.leaky_relu,
                         initializer=initialize,
                         drop_prob=h_param['drop_conv'][i],
                         is training=is training,
                         momentum=h_param['momentum'],
                         name='conv_' + str(i))
        print(conv)

    if h_param['loss_type'] == 'logistic':
        logits = tf.layers.dense(conv, h_param['num_classes']+1,
kernel_initializer=initialize,
                                 name='linear')
    else:
        logits = tf.split(tf.layers.dense(conv, 3, kernel initializer=initialize,
                                          name='linear'),
                          [1, 2], axis=-1)  # output of 3, 1 for ordinal and 2 for binary

    loss, out = get_loss(logits, labels, lengths, h_param)

    loss = tf.multiply(loss, lengths)

    if h_param['loss_type'] == 'logistic':
        predictions = tf.argmax(out, axis=-1, output_type=tf.int32)
    else:
        predictions = (tf.argmax(out[0], axis=-1, output_type=tf.int32) + 1) * \
                      tf.argmax(out[1], axis=-1, output type=tf.int32)

    loss = tf.reduce_mean(loss)

    return loss, predictions
```

7.3.5 Losses code

```
import tensorflow as tf
import numpy as np
from scipy.special import comb
from sklearn.metrics import cohen_kappa_score


def logistic(logits, labels):

    labs = labels['binary'] + labels['ordinal']

    with tf.name_scope("log_loss"):
        log_loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(labels=labs,
logits=logits))

    return log_loss, logits


def unimodal(logits, labels, h_param, reuse=None):

    with tf.variable_scope('temp', reuse=reuse):
        temp = tf.get_variable("temperature", dtype=tf.float32,
initializer=tf.constant(h_param['temp']),
                               trainable=False)

    with tf.variable_scope('num_classes', reuse=reuse):
        num_classes = tf.get_variable('num_classes', dtype=tf.float32,

initializer=tf.constant(np.array([h_param['num_classes']]).astype(np.float32)),
                                      trainable=False)
    with tf.name_scope("ordinal_loss"):
        ord_prob = tf.sigmoid(logits[0])
        copy = tf.tile(ord_prob, (1, h_param['num_classes']))
        classes = tf.range(h_param['num_classes'], delta=1, dtype=tf.float32)
        binom = comb(h_param['num_classes'] - 1, np.arange(0, h_param['num_classes']))

        with tf.variable_scope('binomial', reuse=reuse):
            coefs = tf.get_variable("binomial_coefficients", dtype=tf.float32,
                                    initializer=tf.constant(binom.astype(np.float32)),
trainable=False)

        ordinal_logits = tf.divide(tf.log(coefs) + classes * tf.log(copy) +
                                   (num_classes - 1 - classes) * tf.log(1 - copy), temp)
        ord_loss = tf.reduce_mean(

tf.multiply(tf.nn.sparse_softmax_cross_entropy_with_logits(labels=labels['ordinal'],
logits=ordinal_logits),
                          tf.cast(labels['binary'], tf.float32)))  # multiply by bin to put
every non evaluated slice at 0

    with tf.name_scope("binary_loss"):
        binary_loss =
tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(labels=labels['binary'],
logits=logits[1]))

    return binary_loss+ord_loss, [ordinal_logits, logits[1]]


def full_loss(ord_labs, bin_labs, ord_out, bin_out, num_classes):

    temp = tf.get_variable("temperature", dtype=tf.float32, initializer=tf.constant(1.0),
trainable=True)

    with tf.name_scope("ordinal_loss"):
        ord_prob = tf.sigmoid(ord_out)
        copy = tf.tile(ord_prob, (1, num_classes))
        classes = tf.range(num_classes, delta=1, dtype=tf.float32)
        binom = comb(num_classes - 1, np.arange(0, num_classes))
        coefs = tf.get_variable("binomial_coefficients", dtype=tf.float32,
initializer=tf.constant(binom.astype(np.float32)), trainable=False)
        logits = tf.divide(
```

```
            tf.log(coefs) + classes * tf.log(copy) + tf.multiply(num_classes - 1 - classes,
tf.log(1 - copy)), temp)
        ord loss = tf.reduce mean(
            tf.multiply(tf.nn.sparse softmax cross entropy with logits(labels=ord labs,
logits=logits),
                        tf.cast(bin_labs, tf.float32)))  # multiply by bin_labs to put every
non evaluated slice at 0

    with tf.name_scope("binary_loss"):
        binary loss =
tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(labels=bin_labs,
logits=bin_out))

    return ord_loss + binary_loss, logits


def full_unimodal(logits, labels, h_param, reuse=None):

    with tf.variable_scope('temp', reuse=reuse):
        temp = tf.get_variable("temperature", dtype=tf.float32,
initializer=tf.constant(h_param['temp']),
                               trainable=False)

    with tf.variable_scope('num_classes', reuse=reuse):
        num_classes = tf.get_variable('num_classes', dtype=tf.float32,

initializer=tf.constant(np.array([h_param['num_classes']]).astype(np.float32)),
                                      trainable=False)
    with tf.name_scope("ordinal_loss"):
        ord prob = tf.sigmoid(logits)
        copy = tf.tile(ord_prob, (1, 1, h_param['num_classes']))
        classes = tf.range(h_param['num_classes'], delta=1, dtype=tf.float32)
        binom = comb(h_param['num_classes'] - 1, np.arange(0, h_param['num_classes']))

        with tf.variable_scope('binomial', reuse=reuse):
            coefs = tf.get_variable("binomial_coefficients", dtype=tf.float32,
                                    initializer=tf.constant(binom.astype(np.float32)),
trainable=False)

        ordinal_logits = tf.divide(tf.log(coefs) + classes * tf.log(copy) +
                                   (num classes - 1 - classes) * tf.log(1 - copy), temp)

        ord_loss = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=labels['ordinal'],
logits=ordinal_logits)

    return ord_loss, ordinal_logits


def get_bin_search_labels(labels, search_size):

    val_parse = labels
    binary search labels = []

    for i in reversed(range(search_size)):
        val_parse = (val_parse % np.power(2, i + 1))
        binary_search_labels.append(val_parse // np.power(2, i))

    return tf.stack(binary_search_labels, axis=-1)


def binary_search(logits, labels, initializer, h_param, reuse):

    search_size = np.ceil(np.log2(h_param['num_classes'])).astype(np.int32)

    bin_search_labels = get_bin_search_labels(labels['ordinal'], search_size)

    seq_logits = tf.tile(logits, [1, 1, search_size])
    seq_logits = tf.reshape(seq_logits, shape=[-1, tf.shape(seq_logits)[-1], 1])

    lstm cell = tf.nn.rnn cell.LSTMCell(forget bias=1.0,
                                        state_is_tuple=True,
                                        activation=tf.nn.tanh,
                                        num_units=search_size,
                                        name='binary_search_lstm',
                                        reuse=reuse)

    outputs, state = tf.nn.dynamic_rnn(lstm_cell, seq_logits, dtype=tf.float32)
```

```
    out = tf.layers.dense(outputs, units=2, activation=None, kernel_initializer=initializer,
name='binary_search_linear', reuse=reuse)

    out = tf.reshape(out, shape=[-1, 4, search_size, 2])

    loss = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=bin_search_labels,
logits=out)

    mask = tf.equal(tf.cast(tf.argmax(out, axis=-1), dtype=tf.int32),
tf.cast(bin_search_labels, dtype=tf.int32))

    mask = tf.cumprod(tf.cast(mask, dtype=tf.float32), axis=-1, exclusive=True)

    print(mask)

    return tf.reduce_sum(loss * mask * labels['bin_search_loss_weights'], axis=-1), out
#tf.reduce_sum(loss * mask, axis=-1), out


def binary_search_v2(logits, labels, initializer, h_param, reuse):

    search_size = np.ceil(np.log2(h_param['num_classes'])).astype(np.int32)

    bin_search_labels = get_bin_search_labels(labels['ordinal'], search_size)

    seq_logits = tf.tile(logits, [1, search_size])

    seq logits = tf.expand dims(seq logits, axis=-1)

    lstm cell = tf.nn.rnn cell.LSTMCell(forget bias=1.0,
                                        state_is_tuple=True,
                                        activation=tf.nn.tanh,
                                        num_units=search_size,
                                        name='binary_search_lstm',
                                        reuse=reuse)

    outputs, state = tf.nn.dynamic rnn(lstm cell, seq logits, dtype=tf.float32)

    out = tf.layers.dense(outputs, units=2, activation=None, kernel_initializer=initializer,
                          name='binary_search_linear', reuse=reuse)

    loss = tf.nn.sparse softmax cross entropy with logits(labels=bin search labels,
logits=out)

    mask = tf.equal(tf.cast(tf.argmax(out, axis=-1), dtype=tf.int32),
tf.cast(bin search labels, dtype=tf.int32))

    mask = tf.cumprod(tf.cast(mask, dtype=tf.float32), axis=-1, exclusive=True)

    print(mask)

    return tf.reduce_sum(loss * mask * labels['bin_search_loss_weights'], axis=-1), out


def final_binary_search(logits, labels, initializer, h_param, reuse):

    print('coucou')

    search_size = np.ceil(np.log2(h_param['num_classes'])).astype(np.int32)

    seq_logits = tf.tile(logits, [1, search_size])

    seq logits = tf.expand dims(seq logits, axis=-1)

    lstm_cell = tf.nn.rnn_cell.LSTMCell(forget_bias=1.0,
                                        state_is_tuple=True,
                                        activation=tf.nn.tanh,
                                        num_units=search_size,
                                        name='binary_search_lstm',
                                        reuse=reuse)

    outputs, state = tf.nn.dynamic_rnn(lstm_cell, seq_logits, dtype=tf.float32)

    out = tf.layers.dense(outputs, units=2, activation=None, kernel_initializer=initializer,
                          name='binary_search_linear', reuse=reuse)
```

```
    bin_arange = np.expand_dims(np.arange(np.power(2, search_size), dtype=np.uint8), axis=-1)

    bin arange = np.unpackbits(bin arange, axis=1)[:, -
search size:].flatten().astype(np.int32)

    logits = []

    for i in range(np.power(2, search_size)):
        probs parse = []
        for j in range(search_size):
            probs_parse.append(out[:, j, bin_arange[i * search_size +j]])

        logits.append(tf.reduce_sum(tf.stack(probs_parse, axis=1), axis=-1))

    logits = tf.stack(logits, axis=1)

    loss = tf.losses.sparse_softmax_cross_entropy(labels=labels['ordinal'], logits=logits,
weights=labels['loss_weights'])

    return loss, logits


def binary_search_vtest(logits, labels, initializer, h_param, reuse):

    search_size = np.ceil(np.log2(h_param['num_classes'])).astype(np.int32)

    bin_search_labels = get_bin_search_labels(labels['ordinal'], search_size)

    seq logits = tf.tile(logits, [1, search size])
    seq logits = tf.reshape(seq logits, shape=[-1, tf.shape(seq logits)[-1], 1])

    lstm cell = tf.nn.rnn cell.LSTMCell(forget bias=1.0,
                                        state_is_tuple=True,
                                        activation=tf.nn.tanh,
                                        num units=search size,
                                        name='binary_search_lstm',
                                        reuse=reuse)

    outputs, state = tf.nn.dynamic_rnn(lstm_cell, seq_logits, dtype=tf.float32)

    out = tf.layers.dense(outputs, units=2, activation=None, kernel_initializer=initializer)

    loss = tf.nn.sparse softmax cross entropy with logits(labels=bin search labels,
logits=out)

    return loss, out
```

7.3.6 Training for local modelling problem code

```
import numpy as np
import tensorflow as tf
import data.data_process as dat
from pathlib import Path
from models import tcn_ladder_semisup as tcn_ladder
from sklearn.metrics import cohen_kappa_score
import math

i = 1

while Path('results/h_param_exp_' + str(i) + '.npy').is_file():
    i += 1

exp no = i
print(exp_no)


#class distribution array to derive loss weights
distribution = np.array([855, 1579, 858, 362, 128, 18, 33, 0])


def get_binary_weights(samples):
```

```
    binary_0 = np.array([np.sum(samples[:len(samples) // 2]), np.sum(samples[len(samples) //
2:])])

    binary_1 = np.array([np.sum(samples[i * len(samples) // 4: (i + 1) * len(samples) // 4])
for i in range(4)])

    binary_2 = np.array([np.sum(samples[i * len(samples) // 8: (i + 1) * len(samples) // 8])
for i in range(8)])

    binary_2 = (np.reshape(binary_2, [4, 2]) / np.expand_dims(binary_1, -1)).flatten()

    binary_1 = (np.reshape(binary_1, [2, 2]) / np.expand_dims(binary_0, axis=-1)).flatten()

    binary_0 = np.divide(binary_0, np.sum(binary_0))

    binary_weights = []

    for i in range(8):
        binary_weights.append([binary_0[i // 4], binary_1[i // 2], binary_2[i]])

    return np.array(binary_weights)


binary_weights = get_binary_weights(distribution)

# dictionary containing the model's hyper parameters
h_param = {'filters': [[300, 300], [300, 300], [300, 300]],
           'linear': 200,
           'momentum': 0.99,
           'temp': 1.,
           'loss_type': 'binary_search',  # 'logistic', 'unimodal', 'binary_search', 'lin_reg'
           'num_classes': 7,
           'denoise_conv': [[0.0, 0.0], [0.0]*2, [.0]*2],
           'noise_std': .0,
           'sup_batch_size': 20,
           'unsup_batch_size': 5,
           'num_epoch': 150,
           'starter_rate': 0.0005,
           'decay_after': 4,
           'call_or_counsel': 'caller',
           'pyaudio': False,
           'chosen_model': 'version_2'}  # 'version_1 or version_2

h_param['stride'] = 3 #np.floor(math.log(1200, len(h_param['filters'])*2)).astype(np.int32)
print(h_param['stride'])
np.save('results/h_param_exp_' + str(exp_no), h_param)

# additional hyper parameters
batch_size = h_param['sup_batch_size']
unsup_batch_size = h_param['unsup_batch_size']
num_epochs = h_param['num_epoch']
starter_rate = h_param['starter_rate']
decay_after = h_param['decay_after']
call_or_counsel = h_param['call_or_counsel']
pyaudio = h_param['pyaudio']

distribution[h_param['num_classes'] - 1] += np.sum(distribution[h_param['num_classes']:])
distribution = 1 / distribution[:h_param['num_classes']]
distribution /= np.mean(distribution)

if h_param['chosen_model'] == 'version_1':
    chosen_model = tcn_ladder.model_semisup

else:
    chosen model = tcn ladder.model semisup v2

np.save('results/h_param_exp_' + str(exp_no), h_param)


def eval_on_dataset(data, unsup_batch, session, desired_value, input_place,
unsup_inputs_place, psych_place, train_place, labs_place, bin_place, h_param):

    step_eval = 0
    epoch_eval = 0
    batch eval, step eval, epoch eval = dat.next sup batch(data, step eval, epoch eval,
                                                           False, h_param['sup_batch_size'])
```

```
    values = session.run(desired_value, feed_dict={input_place: batch_eval['features'],
                                                   psych_place: batch_eval['psych'],
                                                   train_place: False,
                                                   labs_place: batch_eval['labels'],
                                                   bin_place: batch_eval['bin'],
                                                   unsup_inputs_place: unsup_batch[:2]})

    while epoch_eval < 1:
        batch_eval, step_eval, epoch_eval = dat.next_sup_batch(data, step_eval, epoch_eval,
                                                               False,
h_param['sup_batch_size'])

        values_parse = session.run(desired_value, feed_dict={input_place:
batch_eval['features'],
                                                             psych_place: batch_eval['psych'],
                                                             train_place: False,
                                                             labs_place: batch_eval['labels'],
                                                             bin_place: batch_eval['bin'],
                                                             unsup_inputs_place:
unsup_batch[:2]})

        for i in range(len(values)):
            values[i] = np.concatenate([values[i], values_parse[i]], axis=0)

    return values


def get_predictions_from_logits(ordinal_logits, psych, h_param):

    predictions = np.argmax(ordinal_logits, axis=-1)

    if h_param['loss_type'] == 'binary_search':

        predictions = np.sum(predictions
                             * np.flip(np.power(2,
np.arange(np.ceil(np.log2(h_param['num_classes']))))), axis=0),
                             axis=-1)

    if h_param['loss_type'] == 'lin_reg':
        predictions = np.abs(np.squeeze(np.around(ordinal_logits))).astype(np.int32)

    predictions = predictions[np.arange(len(psych)), np.argmax(psych, axis=-1)]

    return predictions


def get_predictions_from_logits_v2(ordinal_logits, h_param):

    predictions = np.argmax(ordinal_logits, axis=-1)

    if h_param['loss_type'] == 'binary_search':
        predictions = np.sum(predictions
                             * np.flip(np.power(2,
np.arange(np.ceil(np.log2(h_param['num_classes']))))), axis=0),
                             axis=-1).astype(np.int32)

    if h_param['loss_type'] == 'lin_reg':
        predictions = np.abs(np.squeeze(np.around(ordinal_logits))).astype(np.int32)

    return predictions


def get_ordinal_accuracy(ordinal_predictions, ordinal_labels, binary_labels):

    indices = np.nonzero(binary_labels)
    ordinal_accuracy = np.mean(1 * (ordinal_predictions[indices] == ordinal_labels[indices]))
    kappa = cohen_kappa_score(ordinal_predictions[indices], ordinal_labels[indices],
weights='quadratic')

    return ordinal_accuracy, kappa

# define placeholders
if pyaudio:
    inputs = tf.placeholder(tf.float32, shape=[None, 1200, 34])
    unsup_inputs = tf.placeholder(tf.float32, shape=[None, 1200, 34])
else:
    inputs = tf.placeholder(tf.float32, shape=[None, 1200, 32])
```

```
    unsup_inputs = tf.placeholder(tf.float32, shape=[None, 1200, 32])

psych = tf.placeholder(tf.float32, shape=[None, 4])
is training = tf.placeholder(tf.bool)

labs = tf.placeholder(tf.int32, shape=[None])
binary_labs = tf.placeholder(tf.int32, shape=[None])
loss_weights = tf.placeholder(tf.float32, shape=[None])
bin_search_loss_weights = tf.placeholder(tf.float32, shape=[None, 3])

if h_param['chosen_model'] == 'version_1':
    labels = {'ordinal': tf.tile(tf.expand_dims(labs, axis=-1), (1, 4)),
              'binary': binary_labs,
              'loss_weights': loss_weights,
              'bin_search_loss_weights': bin_search_loss_weights}
else:
    labels = {'ordinal': labs,
              'binary': binary_labs,
              'loss_weights': loss_weights,
              'bin_search_loss_weights': bin_search_loss_weights}

sup_loss, unsup_loss, logits = chosen_model(inputs, unsup_inputs, psych, labels, is_training,
h_param)

loss = sup loss + unsup loss

# define the optimization step, with update_ops to update the batch_norms moving averages
update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)

learning_rate = tf.Variable(starter_rate, trainable=False)
with tf.control_dependencies(update_ops):
    train step = tf.train.AdamOptimizer(learning rate).minimize(loss)

# start tensorflow session
sess = tf.Session()
init = tf.global variables initializer()
sess.run(init)

step = 0
epoch = 0

# load the dataset
training, validation, test = dat.load dataset(py audio=pyaudio,
                                              call_or_counsel=call_or_counsel,
                                              factor_value=h_param['num_classes'])


unsup dataset = dat.load unsup dataset(py audio=pyaudio)

num_iter = int(len(training['labels']) / batch_size) * num_epochs

print('   ----    Training    ----   ')
acc ma = 0.0
bin acc ma = 0.0
sup_loss_ma = 0.0
unsup_loss_ma = 0.0
kappa_ma = 0.0
results_dict = {'loss_sup': [],
                'loss_unsup': [],
                'val_acc_ord': [],
                'val_acc_bin': [],
                'train_acc_ord': [],
                'train_acc_bin': [],
                'val_kappa': [],
                'train_kappa': []}

for i in range(num_iter):

    # load next batch
    batch, batch unsup, step, epoch = dat.next semisup batch(training, unsup dataset, step,
epoch, True, batch_size,
                                                             unsup_batch_size)

    # do train op and get loss value
    results = sess.run([train step, sup loss, unsup loss, logits],
                       feed_dict={inputs: batch['features'],
                                  unsup_inputs: batch_unsup,
```

```
                                    psych: batch['psych'],
                                    is_training: True,
                                    labs: batch['labels'],
                                    binary_labs: batch['bin'],
                                    loss_weights: distribution[batch['labels']],
                                    bin_search_loss_weights: 1/binary_weights[batch['labels']]})

    if h_param['chosen_model'] == 'version_1':

        ordinal metrics = get ordinal accuracy(get predictions from logits(results[-1],
batch['psych'], h_param),
                                               batch['labels'],
                                               batch['bin'])

        acc ma = 0.8 * acc ma + 0.2 * ordinal metrics[0]
        kappa_ma = 0.8 * kappa_ma + 0.2 * ordinal_metrics[1]
        sup_loss_ma = 0.8 * sup_loss_ma + 0.2 * results[1]
        unsup_loss_ma = 0.8 * unsup_loss_ma + 0.2 * results[2]

        if i > 0 and i % 10 == 0:
            if (epoch) >= decay_after:

                if starter_rate == h_param['starter_rate']:
                    starter rate /= 10
                    print(starter rate)

                # decay learning rate
                # learning_rate = starter_learning_rate * ((num_epochs - epoch_n) /
(num_epochs - decay_after))
                ratio = 1.0 * (num_epochs - (epoch + 1))  # epoch_n + 1 because learning rate
is set for next epoch
                ratio = max(0, ratio / (num epochs - decay after))
                print(ratio * starter_rate)
                sess.run(learning_rate.assign(starter_rate * ratio))

            results = eval on dataset(data=validation,
                                      unsup batch=batch unsup,
                                      session=sess,
                                      desired_value=[logits],
                                      input_place=inputs,
                                      unsup inputs place=unsup inputs,
                                      psych place=psych,
                                      train place=is training,
                                      labs_place=labs,
                                      bin_place=binary_labs,
                                      h_param=h_param)

            ordinal predictions = get predictions from logits(results[0],
np.tile(validation['psych'], [2, 1]), h_param)
            ordinal_metrics = get_ordinal_accuracy(ordinal_predictions,
                                                   np.tile(validation['labels'] - 1, 2),
                                                   np.tile(1 * (validation['labels'] > 0), 2))
            mean predictions = np.stack(np.split(ordinal predictions, 2, axis=0), axis=-1)
            mean predictions = np.around(np.mean(mean predictions, axis=-1)).astype(np.int32)
            print(mean_predictions)
            print(validation['labels'])
            mean_metrics = get_ordinal_accuracy(mean_predictions,
                                                validation['labels'] - 1,
                                                1 * (validation['labels'] > 0))

            ord_acc_val = ordinal_metrics[0]
            kappa_val = ordinal_metrics[1]
            non_zero = np.nonzero(validation['labels'])

            print("At %3.0f of %5.0f (epoch %2.0f)\nloss supervised % 6.2f   unsupervised
%6.2f" %
                  (i, num_iter, epoch, sup_loss_ma, unsup_loss_ma))
            print("val acc ordinal % 6.2f (%2.2f)" %
                  (ord_acc_val, acc_ma))
            print("val kappa cohen % 6.2f (%2.2f)\n" %
                  (kappa_val, kappa_ma))
            print(mean_metrics)

            results_dict['train_acc_ord'].append(acc_ma)
            results_dict['train_acc_bin'].append(bin_acc_ma)
            results_dict['train_kappa'].append(kappa_ma)
            results_dict['loss_sup'].append(sup_loss_ma)
```

```
            results_dict['loss_unsup'].append(unsup_loss_ma)
            results_dict['val_acc_ord'].append(ord_acc_val)
            results_dict['val_kappa'].append(kappa_val)

            np.save('results/experiment_'+str(exp_no)+'_results', results_dict)

    else:

        ordinal metrics = get ordinal accuracy(get predictions from logits v2(results[-1],
h param),
                                               batch['labels'],
                                               batch['bin'])

        acc ma = 0.8 * acc ma + 0.2 * ordinal metrics[0]
        kappa ma = 0.8 * kappa ma + 0.2 * ordinal metrics[1]
        sup_loss_ma = 0.8 * sup_loss_ma + 0.2 * results[1]
        unsup_loss_ma = 0.8 * unsup_loss_ma + 0.2 * results[2]

        if i > 0 and i % 20 == 0:

            if (epoch) >= decay_after:
                # decay learning rate
                # learning_rate = starter_learning_rate * ((num_epochs - epoch_n) /
(num_epochs - decay_after))
                if starter_rate == h_param['starter_rate']:
                    starter rate /= 10
                    print(starter_rate)

                ratio = 1.0 * (num_epochs - (epoch + 1))  # epoch_n + 1 because learning rate
is set for next epoch
                ratio = max(0, ratio / (num epochs - decay after))
                sess.run(learning rate.assign(starter rate * ratio))
                print(ratio * starter_rate)

            results = eval on dataset(data=validation,
                                      unsup batch=batch unsup,
                                      session=sess,
                                      desired value=[logits],
                                      input_place=inputs,
                                      unsup_inputs_place=unsup_inputs,
                                      psych place=psych,
                                      train place=is training,
                                      labs place=labs,
                                      bin_place=binary_labs,
                                      h_param=h_param)

            ordinal metrics = get ordinal accuracy(get predictions from logits v2(results[0],
h param),
                                                   np.maximum(validation['labels'] - 1, 0),
                                                   1 * (validation['labels'] > 0))


            ord acc val = ordinal metrics[0]
            kappa val = ordinal metrics[1]
            non_zero = np.nonzero(validation['labels'])
            print(get_predictions_from_logits_v2(results[0], h_param)[non_zero])
            print(validation['labels'][non_zero] - 1)

            print("At %3.0f of %5.0f (epoch %2.0f)\nloss supervised % 6.2f   unsupervised
%6.2f" %
                  (i, num_iter, epoch, sup_loss_ma, unsup_loss_ma))
            print("val acc ordinal % 6.2f (%2.2f)" %
                  (ord_acc_val, acc_ma))
            print("val kappa cohen % 6.2f (%2.2f)\n" %
                  (kappa val, kappa ma))

            results_dict['train_acc_ord'].append(acc_ma)
            results_dict['train_kappa'].append(kappa_ma)
            results_dict['loss_sup'].append(sup_loss_ma)
            results_dict['loss_unsup'].append(unsup_loss_ma)
            results_dict['val_acc_ord'].append(ord_acc_val)
            results_dict['val_kappa'].append(kappa_val)

            np.save('results/experiment_' + str(exp_no) + '_results', results_dict)

        if i > 1 and i%100 == 0:
            results = eval_on_dataset(data=test,
```

```
                                        unsup_batch=batch_unsup,
                                        session=sess,
                                        desired value=[logits],
                                        input place=inputs,
                                        unsup_inputs_place=unsup_inputs,
                                        psych_place=psych,
                                        train_place=is_training,
                                        labs place=labs,
                                        bin place=binary labs,
                                        h param=h param)

            ordinal_metrics = get_ordinal_accuracy(get_predictions_from_logits_v2(results[0],
h_param),
                                                   np.maximum(test['labels'] - 1, 0),
                                                   1 * (test['labels'] > 0))

            ord_acc_test = ordinal_metrics[0]
            kappa_test = ordinal_metrics[1]

            print("test acc ordinal % 5.2f (%2.2f)" %
                  (ord acc test, acc ma))
            print("test kappa cohen % 5.2f (%2.2f)\n" %
                  (kappa_test, kappa_ma))
```

7.3.7 Training for the global modelling problem code

```
import numpy as np
import tensorflow as tf
import data.data_process_context as dat
import time

from models import tcn_context
from sklearn.metrics import cohen_kappa_score


exp_no = 1

# dictionary containing the model's hyper parameters
h_param = {'rec_size': [40, 40],
           'rec_drop': [0.5],
           'momentum': 0.99616,
           'temp': 0.5,
           'loss_type': 'logistic',
           'num_classes': 7}

h_param = {'filters': [[16, 32], [32, 64], [64, 32]],
           'momentum': 0.99616,
           'temp': 0.5,
           'loss_type': 'logistic',
           'num_classes': 7,
           'drop_feed': 0.2,
           'drop_conv': [0.5, 0.5, 0.5, 0.5, 0.5, 0.5]}

# additional hyper parameters
batch_size = 32
rate = 0.005
call_or_counsel = 'caller'
chosen model = tcn context.model


def eval_on_dataset(data, session, desired_value, input_place, psych_place, train_place,
labs place, bin place, lengths place):

    step = 0
    epoch = 0
    value = []

    while epoch < 1:
        batch, step, epoch = dat.next_batch(data, step, epoch, False, batch_size)
        value_parse = session.run(desired_value, feed_dict={input_place: batch['features'],
                                                            psych_place: batch['psych'],
                                                            train_place: False,
```

```
                                                        labs_place: batch['labels'],
                                                        bin_place: batch['bin'],
                                                        lengths_place: batch['lengths']})
        value.append(value parse)

    return value


# define placeholders
inputs = tf.placeholder(tf.float32, shape=[None, None, h_param['num_classes']+1])
psych = tf.placeholder(tf.float32, shape=[None, None, 4])
lengths = tf.placeholder(tf.float32, shape=[None, None])
labs = tf.placeholder(tf.int32, shape=[None, None])
binary_labs = tf.placeholder(tf.int32, shape=[None, None])
labels = {'ordinal': labs,
          'binary': binary_labs}

is_training = tf.placeholder(tf.bool)


loss, predictions = chosen model(inputs, psych, lengths, labels, is training, h param)

# compute overall accuracy
accuracy = tf.reduce_sum(tf.multiply(tf.cast(tf.equal(predictions, labels['binary'] +
labels['ordinal']), tf.float32),
                                     tf.cast(lengths, tf.float32))) /
tf.cast(tf.reduce_sum(lengths), tf.float32)

# define the optimization step, with update_ops to update the batch_norms moving averages
update ops = tf.get collection(tf.GraphKeys.UPDATE OPS)
with tf.control_dependencies(update_ops):
    train step = tf.train.AdamOptimizer(rate).minimize(loss)

# start tensorflow session
sess = tf.Session()
init = tf.global variables initializer()
sess.run(init)

step = 0
epoch = 0

# load the dataset
training, validation, test = dat.load context dataset(call or counsel=call or counsel,
                                                      factor_value=h_param['num_classes'])

for i in range(5000):

    start time = time.time()

    # load next batch
    batch, step, epoch = dat.next_batch(training, step, epoch, True, batch_size)

    # do train op and get loss value
    _, loss_value = sess.run([train_step, loss], feed_dict={inputs: batch['features'],
                                                        psych: batch['psych'],
                                                        is_training: True,
                                                        labs: batch['labels'],
                                                        binary_labs: batch['bin'],
                                                        lengths: batch['lengths']})

    duration = time.time() - start_time

    # store loss value
    if i % 5 == 0:
        print('Step %d, epoch %d: batch_loss = %.2f (%.3f sec) ' % (i, epoch, loss_value,
duration))

    # store training, validation and test performance
    if i % 50 == 0:
        pred val, acc value = sess.run([predictions, accuracy], feed dict={inputs:
batch['features'],
                                                                           psych:
batch['psych'],
                                                                           is_training: True,
                                                                           labs:
batch['labels'],
                                                                           binary_labs:
```

```
batch['bin'],
                                                                    lengths:
batch['lengths']})

        #training_kappa = cohen_kappa_score(pred_val, batch['labels'] + batch['bin'],
weights='quadratic')

        print('\ntraining accuracy = %.2f' % acc_value)
        #print('training quadratic kappa cohen = %.2f' % training_kappa)

        batch_valid, step, epoch = dat.next_batch(validation, 0, 0, False,
len(validation['labels']))
        pred_val, acc_value = sess.run([predictions, accuracy], feed_dict={inputs:
batch_valid['features'],
                                                                    psych:
batch_valid['psych'],
                                                                    is_training: True,
                                                                    labs:
batch_valid['labels'],
                                                                    binary labs:
batch_valid['bin'],
                                                                    lengths:
batch_valid['lengths']})

        pred_val_var_length = np.concatenate([x[:len(y)] for x, y in zip(pred_val,
validation['labels'])])
        validation_kappa = cohen_kappa_score(pred_val_var_length,
np.concatenate(validation['labels']),
                                             weights='quadratic')

        print('\nvalidation accuracy: accuracy: ordinal = %.2f' % acc_value)
        print('validation quadratic kappa cohen = %.2f' % validation_kappa)
        #print(pred_val[:50])
        #print(validation['labels'][:50])
        #np.save('results/validation_' + str(i), [pred_val, validation['labels']])
```