Improving neural networks by reduction of parameters and noise injection.

Marcin Moczulski
Department of Computer Science
St Catherine's College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy

Supervisor: Prof. Nando de Freitas

June 2019

I dedicate this thesis to my parents and Monika.
Dear parents, you gave me the support which made it possible.
Monika, you did the rest.

Abstract

Deep learning has been revolutionising the way we look at prediction, signal processing, perception, classification, RL, control and many others. Neural networks have become the centre of attention of the scientific community revolving around computational solutions to difficult problems.

While in principle such models can be used in their pristine form to approximate complex functions, it is important to point out rather high requirements during training and evaluation.

This thesis is concerned with directly supervised problems - a setting fundamental to the core deep learning. It is the most common incarnation of the neural revolution.

Improving supervised training of deep networks can have a critical impact on the future of prediction based on labels, which is becoming the tool of choice for solving a rapidly growing number of challenges.

Acknowledgements

First of all, I would like to thank Prof. Nando de Freitas for giving me the opportunity to do my D.Phil with him. I am incredibly grateful. It changed my life, and it was extremely empowering to see someone with such knowledge and expertise believing in me and encouraging me to do research.

It is important to remember that what constitutes a great professor is not only her/his personality but also the type of environment she/he creates. Thanks to Nando I had a pleasure to meet Dr Misha Denil and Dr Ziyu Wang. Having the two of you in the lab was one of the most inspiring episodes of my D.Phil. You are so full of insights and so creative. Just observing you is a humbling learning experience.

I also would like to thank Prof. Yoshua Bengio for all the mentorship and all the help you offered me when I visited Montreal. It was a joy and an honour to collaborate with you.

It is probably the right place to acknowledge the fantastic Montreal research crew known under the name MILA. MILA is a unique mix of students, admins, professors and travellers of the Universe.

Caglar Gulcehre you were probably the closest friend of mine during my stay in Yoshua's lab. I want to thank you for all the help and guidance you gave me.

My special thanks go to everybody at University of Oxford. I think one person that especially deserves highlighting is Julie Sheppard. Julie, you are a fantastic person.

I am thankful to all the friends and colleagues I met during these extraordinary years. With some of you I had been sitting side by side in a lab. Some of you I met at conferences. And some of you I still see on a daily basis. I am fortunate that I had met all of you.

Notation

Regular variables such as x represent scalars. Lowercase variables with bold font face like $\mathbf{v}$ are used to denote vectors. Uppercase variables with bold font face like $\mathbf{W}$ represent matrices. When an element of a vector is in mind the notation is a bold face with an index, for example $\mathbf{x}_i$.

The norm of a vector is written as $\|\mathbf{v}\|$. By default the L_2 norm is assumed for vectors. The default norm for matrices is the Frobenius norm denoted $\|\mathbf{W}\|$. $\mathbf{W}^\top$ symbolises the transpose operation. Similarly, $\mathbf{W}^{-1}$ denotes the inverse. The space of k-dimensional real vectors and the space of m-by-n real matrices are represented by $\mathbb{R}^k$ and $\mathbb{R}^{m \times n}$ respectively. The operation $\mathrm{diag}(\mathbf{a})$ evaluates to a diagonal matrix with entries on the diagonal coming from the vector $\mathbf{a}$, for example $\mathbf{A} = \mathrm{diag}(\mathbf{a})$. Element-wise multiplication is expressed with $\odot$.

Regular face denotes a scalar-valued function e.g. $f(x)$. Bold function name refers to a vector-valued function, for example $\mathbf{g}(x)$. $p(\mathbf{w})$ represents a scalar-valued function that takes a vector $\mathbf{w}$ as an argument. The symbol $\sim$ for example in $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \mathrm{diag}(\boldsymbol{\ell}^2)^{-1})$ stands for sampling and usually is followed by the specification of the distribution. The notation $\mathcal{N}(\mu, \sigma^2)$ is a normal distribution with mean μ and standard deviation σ (both scalars). A vector variant of the normal distribution with mean $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$ is written $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$.

The expected value of a random variable is denoted by $\mathbb{E}$, for example $\mu = \mathbb{E}_x[f(x)]$. The gradient of a function $f(\boldsymbol{\theta})$ w.r.t. $\boldsymbol{\theta}$ is symbolised by $\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta})$ or just by $\nabla f(\boldsymbol{\theta})$ for obvious cases. Partial derivatives are marked with ∂ as in $\frac{\partial f(x,y)}{\partial x}$. The Big-O notation is represented by $\mathcal{O}(n)$. In the context of complex numbers i is the imaginary part, for example $x = 17 + 5i$. The element-wise application of $sin(x)$ or $cos(x)$ to a vector $\mathbf{v}$ is expressed with $sin(\mathbf{v})$ and $cos(\mathbf{v})$ respectively and the result is a vector. The symbol $\approx$ stands for an approximation i.e. $\mathbb{E}_\xi[\phi(x, \xi)] \approx h(x)$. Convolution operation is denoted by $*$ as in $(f * g)(\theta) = \int f(x) g(\theta - x) dx$. 'M' stands for millions when the number of parameters of a model is provided, for example "an MLP with 58.6M parameters". The sign function is meant by $sgn(x)$ and takes value -1 when $x < 0$ and takes value $+1$ when $x \geq 0$. The absolute value is denoted by $|z|$. $\mathbf{1}_{r \leq 1}$ is the indicator function and it evaluates to 1 if the condition is true and to 0 otherwise.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Overview

A recent revival of neural networks has brought state of the art results on many benchmarks and sparked a revolution. However, exceptional performance is delivered at the expense of resources such as computational power, fast and relatively large memory, disk space occupied by huge datasets and finally the size of the models themselves. The most popular way to address these requirements is to use Graphical Processing Units (GPUs), which are an extremely parallel architecture suitable for scientific computing. On top of that, GPUs have an advantage in single-precision operations, which is the usual setting for training and evaluating neural networks.

We observe an increase in interest in neural computation on embedded devices. Companies like Google, Apple and NVIDIA and engineering departments in many universities are excited by the idea of putting these impressive models on GPUs embedded in cars, quadcopters and other drones. A difficulty is to find simplifications and workarounds that would allow running networks in such constrained environments.

Distributed computation is also an important perspective. With ubiquitous smartphones and other mobile platforms, the ambition is to make deep learning available to as many people as possible all over the world. However, the size of trained models is one of the main obstacles. Due to the large number of parameters, which is in tens or hundreds of millions of floating-point numbers, neural models imply a considerable storage cost. The Internet's bandwidth limits the ability to distribute models to personal computing units.

Performing a significant amount of computation on user's device would open the door to a massive world-wide adoption of neural networks. Enormous computing centres would not be necessary, therefore giving smaller organisations or individuals a chance to have a considerable impact - something that is usually reserved for big players. This approach could also have a positive environmental impact.

In the first part of this thesis I describe techniques for reducing the number of parameters of neural networks with the goal of addressing the issues mentioned above.

In the second part I explore the concept of noise injection for improving training of neural models. It has been shown that noise plays a vital role in the optimisation of deep networks.

Stochastic variants of gradient algorithms have a noise component. Typically its primary source is the randomness in the estimation of the gradient. It encourages exploration and it allows to escape extrema which are not local minima like saddle points or are early local minima which do not represent a good solution.

Little is known about how to consciously use noise with modern architectures. I present two novel ideas inspired by this question. The experiments support the hypothesis that described forms of noise injection have a positive effect on training.

1.2 Contributions and outline of the thesis

A classical form of a supervised deep learning model is a sequence of dense matrices interleaved with nonlinearities. Usually the cross-entropy loss function is used at the top of the computational graph.

I challenge this view and experiment with deep networks containing structured matrices. The first type that I consider are diagonal matrices because of their extremely low memory footprint, low storage cost and low computational impact. The second type are real Fourier matrices because of their attractive computational properties due to the FFT algorithm. Moreover, as the matrix-vector product with the Fourier matrix can be done implicitly, they have zero memory and storage requirements.

Although structured matrices limit the memory usage by reducing the number of parameters, unfortunately it is often counter-balanced by the increased number of stored activations effectively offering modest savings. Therefore my investigation focuses on the storage. It is the direction where structured matrices yield the greatest improvements.

Furthermore, I look at a selection of possible ways of injecting noise into a neural network. Noise injection has been investigated in the past, but it remains an open question how to apply it to modern architectures.

I explore two algorithms concerned with adding random signal to a model. The first one is based on the idea of adapting the noise to the problem. The second one is focused on annealing the noise and is related to curriculum learning and continuation methods.

It is essential to be careful about the computational cost when considering the benefits of a novel algorithm for training neural networks. Noise injection can be a desirable candidate due to its marginal computational cost.

Chapter 2 of this thesis gives a quick introduction to neural networks. Its purpose is to provide a foundation and a common vocabulary. Chapter 3 describes my approaches to reduction of parameters. Chapter 4 is devoted to noise injection. Chapter 5 contains conclusions, a compressed outline of the advantages and disadvantages of described methods, and future work.

1.2.1 Summary of Chapter 3

The first problem I consider is the reduction of parameters in the AlexNet model [Krizhevsky et al., 2012] by using structured matrices and wide layers. The method is connected to kernel methods. I use the FastFood [Le et al., 2013] parameterisation to retain a Gaussian-like behaviour of matrices. That amounts to computing fast approximations to an expansion of the Gaussian kernel.

Furthermore, I treat adaptable elements of FastFood as parameters and I use the back-propagation algorithm to compute derivatives of the loss function with respect to them. I arrive at end-to-end trainable networks with efficient structured matrices and other desirable properties. Moreover, the initial state of the network has theoretical justification from random projections.

Experiments show a reduction of around 50% in the number of parameters without loss of accuracy as compared to the original AlexNet model. The new architecture is called DeepFriedConvnet [Yang et al., 2014].

The second challenge I tackle is a more aggressive reduction of parameters with much deeper models. Instead of constructing a few wide layers I propose a deep sequence of narrow modules and a new parameterisation of matrices. Although I still use the Fourier basis and diagonal matrices, they do not take the FastFood form.

Layers are thin and each one is interleaved with ReLUs. The central principle is to perform a deep sequence of element-wise multiplication in the time domain followed by element-wise multiplication in the frequency domain followed by the ReLU nonlinearity. With the proposed architecture I achieve a 6x reduction in the number of parameters while experiencing only $< 1\%$ drop in accuracy. The method is applicable both during training and testing.

1.2.2 Summary of Chapter 4

Although it has been argued that noise injection can have a beneficial impact on training neural networks, it has not been explored in depth for modern neural models. A recent proposition of adding noise directly to the gradient [Neelakantan et al., 2015b] shows that such an operation can produce significant improvements.

The first method I present is concerned with injecting noise in networks that use logistic activation functions when a unit (neuron) becomes approximately saturated. It is desired because such units provide almost no gradient information, which slows down training.

The amount of injected noise is learned together with the rest of the neural network. It is possible to obtain the gradient of the loss function with respect to parameters of the noise generating distribution.

Experiments illustrate that a drop-in application of the method achieves an improvement on many problems without virtually any search over hyperparameters.

The second method is related to curriculum learning and annealing schedules. Sometimes the problem at hand is too difficult to optimise directly. For example, it may be necessary to construct a curriculum which orders training examples from easier ones to harder ones. Learning corresponds to solving easier parts of the task first and then progressing to more challenging ones.

However, building a curriculum requires manual work and expert knowledge. Human intervention is necessary to judge the difficulty of the data. Instead, I investigate an idea related to training recurrent neural networks by diffusion. As an alternative to coming up with a curriculum, one can substitute the original loss function with a sequence of approximations that are easier to optimise.

Chapter 2

What are neural networks?

2.1 Feedforward networks

Feedforward networks usually are trained with Stochastic Gradient Descent (SGD) algorithms. This process can be broken down into a forward and a backward pass. In the forward pass all activations are calculated for a given input. In the backward pass the derivative of the loss with respect to every parameter is computed. Feedforward networks do not have any recurrent connections, which are present when parameters are repeatedly used to compute outputs over the time dimension.

The natural setup for a feedforward network is when the network in one time step fully observes a mini-batch of data examples. In the case of classification each example has a single target label. In the case of regression each example has a single target real value.

The way the loss is calculated depends on the choice of the objective function. For classification a typical choice is the likelihood or the log-likelihood. It corresponds to minimising the cross-entropy. In other words it is desirable to minimise the uncertainty of the choice of the correct target label. For regression the most common objective function is the squared distance between the predicted value and a known target value. The choice of the loss function affects the derivatives calculated in the backward pass.

The parameters are real numbers and their function depends on the type of layers they constitute. Examples of prevalent types of layers are convolutional layers and fully-connected layers.

Training of a feedforward network most of the time is done with stochastic gradient descent algorithm (SGD). Usually forward and backward steps are done in mini-batches.

2.2 Recurrent networks

Recurrent neural networks recently became very popular mainly because of their natural suitability for handling input sequences paired with a single label or handling input sequences paired with a sequence of labels. Alternatively, data can take a form of input sequences paired with sequences of real numbers (similar to regression).

The choice of the objective function is to some extent analogous with feedforward networks. However, it brings a few additional challenges. For example, when mapping from an input sequence to a single target label, some sophistication has to be added to a typical objective function used in feedforward networks.

Parameters in recurrent neural networks are used in a different way than in feedforward models. Every parameter is used more than once in the forward and the backward pass. The output from one time step can serve as an input to the next one resulting in a recurrence.

The most common algorithm for training recurrent networks is a slightly modified SGD. Usually in the forward pass the recurrence is unrolled for a number of time-steps. The backward pass is done back through time .

2.2.1 LSTMs, GRUs

Unfortunately, classical RNNs suffer from gradients vanishing or exploding. Gradients in the backward pass are multiplied by the same matrix during the backpropagation. The directions associated with eigenvalues shrink or grow exponentially quickly. It leads to unstable learning and difficulties with propagating long dependencies.

Moreover, there is no place to explicitly store memories. All observed history is accumulated in a hidden state vector, the contents of which change on every timestep.

A new type of RNN was proposed to alleviate these problems - Long Short-Term Memory (LSTM) [Hochreiter and Schmidhuber, 1997b]. The idea is to store the memory in a special cell. A gating mechanism was introduced to filter information coming in and out of the cell, including information presented on the input. Gradients travel more complicated paths during the backpropagation, and it is unlikely that the signal gets disproportionately reduced or boosted in any direction over time. The place to explicitly store the memories and gates controlling access to it proved successful when learning longer dependencies.

LSTM is an exceptionally successful architecture, but it comes at a cost. The increased number of gates leads to the increased number of parameters and an increased amount of computation. It also generates significantly more activations. Quickly those obstacles became fundamental performance bottlenecks.

After further investigation, the original formulation turned out to be overcomplicated and some elements seemed unnecessary. In 2014 Gated recurrent unit (GRU) was born . It is an architecture with alternative gating compared to LSTM. Although both enjoy a similar performance, GRUs have fewer parameters and are significantly faster.

2.3 Types of layers

While modern neural networks use many types of layers, the two most popular ones are convolutional layers [Fukushima, 1980], [Lecun et al., 1998] and fully-connected layers. While fully-connected layers are a fundamental element of almost all modern architectures, convolutional layers play a significant role in recent advancements in computer vision. Convolutional layers usually contain only 10% of all parameters and take 90% of total computation. In contrast, fully-connected layers have 90% of all parameters and use only 10% of total computation. This crucial fact allows to approach the task of reducing the number of parameters in modern models.

2.4 Graphics processing units (GPUs)

A critical innovation that allowed for the return of neural networks are graphics processing units (GPUs). They are generally simpler and better tailored for the task than CPUs thanks to their massive parallelism. Typical GPU has less RAM than a CPU-based solution. Moreover, the execution of a single thread is slower on GPU than on CPU for most of the time. However, the number of simultaneous threads on a GPU is significantly larger. It is essential to keep the size of the model in mind, as it is necessary to fit it with a mini-batch of data and models activations into the main GPU memory. That can be a difficult task. Especially on embedded devices like GPU installations found in modern cars.

The bottlenecks in GPU computation are of two kinds. The problem can be the speed of computation or the speed of sending data from the main on-device memory (relatively slow) to a tiny shared memory (relatively fast).

Chapter 3

Improving neural networks by reduction of parameters

3.1 Motivation

GPUs offer around 10 times faster execution of most neural models than CPUs. Moreover, an increased interest in application-specific integrated circuits (ASICs) suggests that soon we may expect an even more impressive technology. Nevertheless, there is a substantial amount of research focused on scaling neural networks. The community is always excited about creating bigger and more powerful models today rather than having to wait a decade for improvements to the hardware.

Modern neural networks have dozens of millions of parameters, which is in contrast with the most of other approaches. Although they perform exceptionally well on a wide range of problems, the large number of parameters implies a heavy footprint both during training and testing. We can distinguish three types of footprint: computation, memory and storage.

There is strong evidence that deep neural networks are over-parameterised [Denil et al., 2013b]. The results suggest that in some cases as many as 95% of values of the parameters can be predicted from the remaining 5%. It implies the existence of an underlying structure which could be used to reduce the number of parameters. Such models would be smaller and more efficient.

Deep neural networks contain all of their parameters in affine transformations, which are interleaved with nonlinearities like ReLUs . Therefore those dense matrices will be the primary object of interest.

3.2 Convolutional vs fully-connected layers

Neural networks have two fundamental building blocks with radically different behaviour and properties: convolutional layers and fully-connected layers. Fully-connected layers usually account for 90% of the total number of parameters, while convolutional layers account only for the remaining 10% . Moreover, convolutional layers are responsible for around 90% of the computation and fully-connected layers are responsible only for around 10%.

Convolutional layers are computationally expensive. However, their computation per parameter ratio is impressive because they use parameters sharing. Effectively convolutional layers do not contribute much to the overall number of parameters and the storage requirements.

On the other hand fully-connected layers are very thrifty with computation. The computation per parameter ratio is relatively low. However, their contribution to the overall number of parameters is significant. Therefore they are the dominant cause of high storage cost of neural networks.

Because those two types of layers should be looked at from different perspectives, these observations will be crucial to scaling neural networks by parameters reduction.

3.3 Three perspectives on scaling

I present three major aspects of improving neural networks: computation, memory and storage. All of them are of different nature and all of them are borderline high. Each aspect has to be considered separately and probably it should not be expected that a single approach will address all of them .

Computational requirements are to some extent satisfied by faster hardware like GPUs or ASICs. Scaling happens due to usual advancements in hardware by using even more sophisticated chips and faster clocks. Without a fundamental change in the way we train and test neural networks, for example by using optics, we should not expect a revolution in this dimension. To a large extent it remains a hardware-related problem.

One of the findings of this research is that although the memory consumption is related to the number of parameters, mainly it is a result of storing a large number of activations during the forward pass. The size of mini-batches used with stochastic gradient descent (SGD) has a critical role in how much memory the model needs. Similarly to the problem of computation, probably we need to wait for chips with bigger, fast memory located close to where the computation takes place.

Storage requirements are implied by how the model is constructed which also has an impact on some of the memory requirements and computation. One of the disadvantages of having a large number of parameters are bandwidth problems encountered when, for example, the model is distributed to smartphones. Current throughput of the mobile internet does not allow for easy and convenient download of neural models and it seems that the size of models grows significantly faster than the abilities of remote data transmission. Moreover, it is worth noticing that until recently most of the popular neural architectures were trained on a single GPU on a single machine. It is clear that this trend is changing with multi-GPU training becoming more popular. Furthermore, recent improvements in large mini-batch optimisation suggest that possibly in the future training on multiple GPUs on multiple machines will be a standard setup.

Multi-machine, multi-GPU setting requires frequent exchanges of parameters between workers and parameters server. The number of parameters that have to be exchanged directly impacts the efficiency of such approach.

3.3.1 Scaling focused on computation

This imbalance between fully-connected layers and convolutional layers suggests that the efficiency of these two types of layers should be addressed in different ways. [Denton et al., 2014] and [Jaderberg et al., 2014] describe methods for minimising computational cost of evaluating a network at test time by replacing the convolutional filters with their separable approximations.

These approaches realise speed gains at test time but do not address the issue of training since the approximations are made after the network has been fully trained. Additionally, neither approach addresses the issue of the number of parameters since they both work with approximations of the convolutional layers, which represent only a small portion of the total storage required. Many other works have addressed the computational efficiency of convolutional networks in more specialised settings [Farabet et al., 2010, Li et al., 2014].

3.3.2 Scaling focused on memory

In contrast to the above approaches, [Denil et al., 2013a] demonstrate that there is significant redundancy in the parameterisation of several deep learning models, and exploit this to reduce the number of parameters required. More specifically, their method represents the parameter matrix as a product of two low-rank factors, and the training algorithm fixes one factor (called static parameters) and only updates the other factor (called dynamic parameters). However, the static parameters are not jointly trained with the dynamic parameters.

[Sainath et al., 2013, Xue et al., 2013] are similar to [Denil et al., 2013a] in that they use SVD to decompose the matrix to reduce the memory footprint, but the factorisation is only applied to the trained model as a post-processing step. In contrast, methods presented in this thesis allow training more a more lightweight version of a network from scratch.

3.3.3 Scaling focused on storage

A side effect of neural networks having millions of parameters is a prohibitively big size of the trained model. For example a well-known model of Krizhevsky [Krizhevsky et al., 2012] needs 233 MB of storage. That impacts hardware with limited resources like cell phones and embedded devices. A possibly even more significant obstacle is the time it takes to download such a model over the internet to user's device (e.g. smartphone). It is a big issue in nowadays mobile computing. Being able to do at least a part of the computation on the users side is the key to using neural networks on a massive, unprecedented scale.

Most of the storage is taken by fully-connected layers that contain around 90% of all parameters. This insight will prove crucial to the proposed method of reduction of storage. A critical remark is that reducing the number of parameters in the network does not automatically imply a reduction in overall GPU memory consumption. An increased number of stored activations (e.g. using wide layers) can consume most of the savings.

3.4 My approach

The total memory required to represent a deep convolutional neural network can be substantially reduced without sacrificing predictive performance. My approach replaced the fully-connected layers of the network with a kernel machine. In particular the Fastfood method of [Le et al., 2013] or with factorisations inspired by Fastfood . Previous nonlinear kernel machines have not been able to scale to large datasets such as ImageNet (millions of data points and 1000 classes) since their memory requirements are typically quadratic in the number of data points. By using Fastfood to represent a nonlinear kernel, one is able to retain the full representation power of a kernel machine, while at the same time being much more efficient in computation and memory.

Another innovation is an adaptive variant of Fastfood which allows jointly learning the kernel function along with the rest of the convolution parameters of the network. This novel network architecture, which is called a deep fried convolutional network, is able to achieve the same predictive performance as a standard convolutional network on deploy on ImageNet problem using approximately half the number of parameters. Further reduction is also possible with a marginal loss of performance.

Moreover inspired by Fastfood I developed a new type of structured layer called ACDC. The idea is to use efficient elements from Fastfood and create a structured, thrifty alternative to dense layers of a neural network.

3.5 Global average pooling

There have been other attempts to replace the fully-connected layers. The Network in Network architecture of [Lin et al., 2014] achieves state of the art results on several deep learning benchmarks by replacing the fully-connected layers with global average pooling. A similar approach was used by [Szegedy et al., 2014] to win the ILSVRC 2014 object detection competition [Russakovsky et al., 2014].

Although the global average pooling approach achieves impressive results, it has two significant drawbacks. First, feature transfer is more difficult with this approach. It is very common in practice to take a convolutional network trained on ImageNet and re-train the top layer on a different dataset. Re-using features learned from ImageNet for the new task (potentially with fine-tuning) is difficult with global average pooling. This deficiency is noted by [Szegedy et al., 2014], and motivates them to add an extra linear layer to the top of their network to enable easier adaptation to other label sets. The second drawback of global average pooling is computation. Convolutional layers are much more expensive to evaluate than fully-connected layers, so replacing fully-connected layers with more convolutions can decrease model size but comes at the cost of increased evaluation time. In contrast, my approach decreases both model size and computational complexity.

3.6 Sparsity inducing regularizers

Recently [Collins and Kohli, 2014b] have targeted memory usage of the fully-connected layers of convolutional networks. Their approach is based on applying a sparsity inducing regularizer during optimisation which introduces many zero-weight connections that can be removed at test time. Although they achieve a substantial reduction in the total number of parameters, it is accompanied by a significant drop in top-1 performance (1.7%). The reduction in memory usage is less dramatic since realising memory gains requires maintaining sparse data structures which introduce additional memory overhead. Moreover, their approach reduces the number of parameters only at test time, whereas my approach also applies to training.

3.7 Kernels

This work is in line with recent interest in combining kernels with deep neural networks [Cho and Saul, 2009a, Dai et al., 2014b, Huang et al., 2014, Mairal et al., 2014a]. However, the proposed method presents significant advances over previous attempts allowing for gains in term of both accuracy and storage.

The Doubly Stochastic Gradients method of [Dai et al., 2014a] showed that effective use of randomisation allows kernel methods to scale to extensive datasets approaching the performance of deep learning methods on ImageNet using convolutional features. However, this approach operates on fixed convolution features and it cannot jointly learn the kernel classifier and convolution filters. [Mairal et al., 2014b] showed how to learn a kernel function in an unsupervised manner which can take into account hierarchical convolution features. Unlike in [Mairal et al., 2014b] here the kernel representation is learned in a supervised way.

3.8 Random features for kernels

The main bottleneck in scaling up kernel methods is the storage and computation of the kernel matrix, $\mathbf{K}$, which is usually dense. Storing the matrix requires $\mathcal{O}(m^2)$ space, and computing it takes $\mathcal{O}(m^2 d)$ operations, where m is the number of data points and d is the dimension. A valuable insight noted in [Rahimi and Recht, 2008], is that the infinite kernel expansion can be approximated in an unbiased manner using a randomly drawn basis function. For shift-invariant kernels this relies on a classical result from harmonic analysis,

Theorem 1 (Bochner). A continuous, shift-invariant kernel $k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x} - \mathbf{x}')$ on $\mathbb{R}^d$ is positive definite if and only if k is the Fourier transform of a non-negative measure $\mu(\mathbf{w})$.

This measure, known as the spectral density, implies the existence of a probability measure $p(\mathbf{w}) = \frac{\mu(\mathbf{w})}{\alpha}$ such that

$$k(\mathbf{x}, \mathbf{x}') = \int \alpha e^{-i\mathbf{w}^\top(\mathbf{x}-\mathbf{x}')} p(d\mathbf{w}) \tag{3.1}$$

$$= \alpha \mathbb{E}_{\mathbf{w}}[\cos(\mathbf{w}^\top \mathbf{x}) \cos(\mathbf{w}^\top \mathbf{x}') + \sin(\mathbf{w}^\top \mathbf{x}) \sin(\mathbf{w}^\top \mathbf{x}')] \tag{3.2}$$

where the imaginary part is dropped since both the kernel and distribution are real. Finally, by sampling n vectors i.i.d. from p and collecting them in a matrix $\mathbf{W} = (\mathbf{w}_1, ...\mathbf{w}_n)^\top$, this kernel can then be approximated as the inner-product of random features

$$\phi(\mathbf{x}) = \sqrt{\frac{\alpha}{n}} (\cos(\mathbf{W}\mathbf{x}), \sin(\mathbf{W}\mathbf{x}))^\top \tag{3.3}$$

where the $\cos$ and $\sin$ functions are applied element-wise to the vector $\mathbf{Wx}$. Then approximating a kernel function with random features becomes a matter of deriving the correct distribution for $\mathbf{w}$. Such distributions have been derived for many commonly used kernels such as squared exponential kernel

$$k(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\ell^2}\right) \tag{3.4}$$

where the weight distribution is a Gaussian distribution

$$\mathbf{w}_{SE} \sim \mathcal{N}(\mathbf{0}, \mathrm{diag}(\boldsymbol{\ell}^2)^{-1}) \tag{3.5}$$

In fact, the random feature representation of a kernel function is generally applicable to any positive definite kernel function, and not just limited to shift-invariant kernels. For instance, if $\mathbf{W}$ is a random Gaussian matrix and the activation function is a Rectified Linear unit (ReLU)

$$\phi(\mathbf{x}) = \sqrt{\frac{1}{n}}\max(0, \mathbf{Wx}) \tag{3.6}$$

then the rotationally invariant arc-cosine kernel introduced in [Cho and Saul, 2009b, Pandey and Dukkipati, 2014] is obtained

$$k(\mathbf{x}, \mathbf{x}') = \mathbb{E}_{\mathbf{w}}[\phi(\mathbf{x})^\top \phi(\mathbf{x}')] \tag{3.7}$$

$$= \frac{1}{2\pi}\|\mathbf{x}\|\|\mathbf{y}\|(\sin(\theta) + (\pi - \theta)\cos(\theta)) \tag{3.8}$$

where θ is the angle between $\mathbf{x}$ and $\mathbf{x}'$.

3.9 Deep Fried Convolutional Networks

3.9.1 Introduction

In recent years we have witnessed an explosion of applications of convolutional neural networks with millions and billions of parameters. Reducing this vast number of parameters would improve the efficiency of training in distributed architectures. It would also allow for the deployment of state-of-the-art convolutional neural networks on embedded mobile applications. These train and test time considerations are both of great importance.

A standard convolutional network is composed of two types of layers, each with very different properties. Convolutional layers, which contain a small fraction of the network parameters, represent the most of the computational effort. In contrast, fully-connected layers contain the vast majority of the parameters but are comparatively cheap to evaluate [Krizhevsky, 2014].

This work shows how the number of parameters required to represent a deep convolutional neural network can be substantially reduced without sacrificing predictive performance. The approach benefits from replacing the fully-connected layers of the network with an Adaptive Fastfood transform, which is a generalisation of the Fastfood transform for approximating kernels [Le et al., 2013].

Convolutional neural networks with Adaptive Fastfood transforms, which are referred to as deep fried convnets, are end-to-end trainable and achieve the same predictive performance as standard convolutional networks on ImageNet using approximately half the number of parameters.

3.9.2 The Adaptive Fastfood Transform

Large dense matrices are the main building block of fully-connected neural network layers. In propagating the signal from the l-th layer with d activations $\mathbf{h}_l$ to the $l+1$-th layer with n activations $\mathbf{h}_{l+1}$, it is necessary to compute

$$\mathbf{h}_{l+1} = \mathbf{W}\mathbf{h}_l \tag{3.9}$$

The storage and computational costs of this matrix multiplication step are both $\mathcal{O}(nd)$. In particular the storage cost can be prohibitive for many applications.

The proposed solution is to reparameterise the matrix of parameters $\mathbf{W} \in \mathbb{R}^{n \times d}$ with an Adaptive Fastfood transform, as follows

$$\mathbf{h}_{l+1} = (\mathbf{SHG\Pi HB})\,\mathbf{h}_l = \widehat{\mathbf{W}}\mathbf{h}_l \tag{3.10}$$

In Section 3.9.4, I provide background and intuitions behind this design. For now it suffices to state that the storage requirements of this reparameterisation are $\mathcal{O}(n)$ and the computational cost is $\mathcal{O}(n \log d)$. The experimental section also shows that these theoretical savings are mirrored in practice by significant reductions in the number of parameters without increased prediction errors.

To understand these claims, I need to describe the component modules of the Adaptive Fastfood transform. For simplicity of presentation, let us first assume that $\mathbf{W} \in \mathbb{R}^{d \times d}$. Adaptive Fastfood has three types of modules:

- $\mathbf{S}, \mathbf{G}$ and $\mathbf{B}$ are diagonal matrices of parameters. In the original non-adaptive Fastfood formulation they are random matrices, as described further in Section 3.9.4. The computational and storage costs are trivially $\mathcal{O}(d)$.
- $\mathbf{\Pi} \in \{0,1\}^{d \times d}$ is a random permutation matrix. It can be implemented as a lookup table, so the storage and computational costs are also $\mathcal{O}(d)$.

- $\mathbf{H}$ denotes the Walsh-Hadamard matrix, which is defined recursively as

$$\mathbf{H}_2 := \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \text{ and } \mathbf{H}_{2d} := \begin{bmatrix} \mathbf{H}_d & \mathbf{H}_d \\ \mathbf{H}_d & -\mathbf{H}_d \end{bmatrix}$$

  The Fast Hadamard Transform, a variant of Fast Fourier Transform, enables us to compute $\mathbf{H}_d\mathbf{h}_l$ in $\mathcal{O}(d \log d)$ time.

In summary, the overall storage cost of the Adaptive Fastfood transform is $\mathcal{O}(d)$, while the computational cost is $\mathcal{O}(d \log d)$. These are substantial theoretical improvements over the $\mathcal{O}(d^2)$ costs of ordinary fully-connected layers.

When the number of output units n is larger than the number of inputs d, one can perform n/d Adaptive Fastfood transforms and stack them to attain the desired size. In doing so, the computational and storage costs become $\mathcal{O}(n \log d)$ and $\mathcal{O}(n)$ respectively, as opposed to the more substantial $\mathcal{O}(nd)$ costs for linear modules. The number of outputs can also be refined with pruning.

3.9.3 Learning Fastfood by backpropagation

The parameters of the Adaptive Fastfood transform ($\mathbf{S}, \mathbf{G}$ and $\mathbf{B}$) can be learned by standard error derivative backpropagation. Moreover, the backward pass can also be computed efficiently using the Fast Hadamard Transform.

In particular, let us consider learning the l-th layer of the network, $\mathbf{h}_{l+1} = \mathbf{SHG\Pi HB}\mathbf{h}_l$. For simplicity it is assumed that $\mathbf{W} \in \mathbb{R}^{d \times d}$ and that $\mathbf{h}_l \in^d$. It should be noted that $\frac{\partial E}{\partial \mathbf{h}_{l+1}}$, where E is the objective function, is already available by virtue of backpropagation. Then

$$\frac{\partial E}{\partial \mathbf{S}} = \text{diag} \frac{\partial E}{\partial \mathbf{h}_{l+1}} (\mathbf{HG\Pi HB}\mathbf{h}_l)^\top \tag{3.11}$$

Since $\mathbf{S}$ is a diagonal matrix, it is necessary only to calculate the derivative with respect to the diagonal entries and this step requires only $\mathcal{O}(d)$ operations.

Proceeding in this way, denote the partial products by

$$\begin{aligned} \mathbf{h}_S &= \mathbf{HG\Pi HB}\mathbf{h}_l \\ \mathbf{h}_{H1} &= \mathbf{G\Pi HB}\mathbf{h}_l \\ \mathbf{h}_G &= \mathbf{\Pi HB}\mathbf{h}_l \\ \mathbf{h}_\Pi &= \mathbf{HB}\mathbf{h}_l \\ \mathbf{h}_{H2} &= \mathbf{B}\mathbf{h}_l \end{aligned} \tag{3.12}$$

Then the gradients with respect to different parameters in the Fastfood layer can be computed recursively as follows

$$\begin{aligned}
\frac{\partial E}{\partial \mathbf{h}_S} &= \mathbf{S}^\top \frac{\partial E}{\partial \mathbf{h}_{l+1}} & \frac{\partial E}{\partial \mathbf{h}_{H1}} &= \mathbf{H}^\top \frac{\partial E}{\partial \mathbf{h}_S} \\
\frac{\partial E}{\partial \mathbf{G}} &= \mathrm{diag} \frac{\partial E}{\partial \mathbf{h}_{H1}} \mathbf{h}_G^\top & \frac{\partial E}{\partial \mathbf{h}_G} &= \mathbf{G}^\top \frac{\partial E}{\partial \mathbf{h}_{H1}} \\
\frac{\partial E}{\partial \mathbf{h}_\Pi} &= \mathbf{\Pi}^\top \frac{\partial E}{\partial \mathbf{h}_G} & \frac{\partial E}{\partial \mathbf{h}_{H2}} &= \mathbf{H}^\top \frac{\partial E}{\partial \mathbf{h}_\Pi} \\
\frac{\partial E}{\partial \mathbf{B}} &= \mathrm{diag} \frac{\partial E}{\partial \mathbf{h}_{H2}} \mathbf{h}_l^\top & \frac{\partial E}{\partial \mathbf{h}_l} &= \mathbf{B}^\top \frac{\partial E}{\partial \mathbf{h}_{H2}}
\end{aligned} \tag{3.13}$$

The operations in $\frac{\partial E}{\partial \mathbf{h}_{H1}}$ and $\frac{\partial E}{\partial \mathbf{h}_{H2}}$ are simply applications of the Hadamard transform since $\mathbf{H}^\top = \mathbf{H}$, and consequently can be computed in $\mathcal{O}(d \log d)$ time. The operation in $\frac{\partial E}{\partial \mathbf{h}_\Pi}$ is an application of a permutation (the transpose of a permutation matrix is a permutation matrix) and can be computed in $\mathcal{O}(d)$ time. All other operations are diagonal matrix multiplications.

3.9.4 Intuitions behind Adaptive Fastfood

The proposed Adaptive Fastfood transform may be understood either as a trainable type of structured random projection or as an approximation to the feature space of a learned kernel. Both views not only shed light on Adaptive Fastfood and competing techniques but also open up room to innovate new techniques to reduce computation and memory in neural networks.

3.9.4.1 A view from structured random projections

Adaptive Fastfood is based on the Fastfood transform [Le et al., 2013], in which the diagonal matrices $\mathbf{S}$, $\mathbf{G}$ and $\mathbf{B}$ have random entries. The experiments compare the performance of the existing random and the proposed adaptive versions of Fastfood when used to replace fully-connected layers in convolutional neural networks.

The intriguing idea of constructing neural networks with random weights has been reasonably explored in the neural networks field [Saxe et al., 2011, Jaeger and Haas, 2004]. This idea is related to random projections, which have been deeply studied in theoretical computer science [Mitzenmacher and Upfal, 2005]. In a random projection, the basic operation is of the form

$$\mathbf{y} = \mathbf{W}\mathbf{x} \tag{3.14}$$

where $\mathbf{W}$ is a random matrix, either Gaussian [Indyk and Motwani, 1998] or binary [Achlioptas, 2003]. Importantly, the embeddings generated by these random projections

approximately preserve metric information, as formalised by many variants of the celebrated Johnson-Lindenstrauss Lemma.

The one shortcoming of random projections is that the cost of storing the matrix $\mathbf{W}$ is $\mathcal{O}(nd)$. Using a sparse random matrix $\mathbf{W}$ by itself to reduce this cost is often not a viable option because the variance of the estimates of $\|\mathbf{Wx}\|$ can be very high for some inputs, for example when $\mathbf{x}$ is also sparse. To see this, consider the extreme case of a very sparse input $\mathbf{x}$, then many of the products with $\mathbf{W}$ will be zero and hence not help improve the estimates of metric properties of the embedding space.

One popular option for reducing the storage and computational costs of random projections is to adopt random hash functions to replace the random matrix multiplication. For example, the count-sketch algorithm [Charikar et al., 2004] uses pairwise independent hash functions to carry this job very effectively in many applications [Cormode et al., 2012]. This technique is often referred to as the hashing trick [Weinberger et al., 2009] in the machine learning literature. Hashes have irregular memory access patterns, so it is not clear how to get good performance on GPUs when following this approach, as pointed out in [Chen et al., 2015].

Ailon and Chazelle [Ailon and Chazelle, 2009] introduced an alternative approach that is not only very efficient but also preserves most of the desirable theoretical properties of random projections. Their idea was to replace the random matrix by a transform that mimics the properties of random matrices, but which can be stored efficiently. In particular, they proposed the following PHD transform

$$\mathbf{y} = \mathbf{PHDx} \tag{3.15}$$

where $\mathbf{P}$ is a sparse $n \times d$ random matrix with Gaussian entries, $\mathbf{H}$ is a Hadamard matrix and $\mathbf{D}$ is a diagonal matrix with $\{+1, -1\}$ entries drawn independently with probability $1/2$. The inclusion of the Hadamard transform avoids the problems of using a sparse random matrix by itself, but it is still efficient to compute.

One can think of the original Fastfood transform

$$\mathbf{y} = \mathbf{SHG\Pi HBx} \tag{3.16}$$

as an alternative to this. Fastfood reduces the computation and storage of random projections to $\mathcal{O}(n \log d)$ and $\mathcal{O}(n)$ respectively. In the original formulation $\mathbf{S}, \mathbf{G}$ and $\mathbf{B}$ are diagonal random matrices, which are computed once and then stored.

In contrast, in the proposed Adaptive Fastfood transform, the diagonal matrices are learned by backpropagation. By adapting $\mathbf{B}$ the algorithm effectively implements Automatic Relevance Determination on features. The matrix $\mathbf{G}$ controls the bandwidth of the

kernel and its spectral incoherence. Finally, $\mathbf{S}$ represents different kernel types. For example, for the RBF kernel $\mathbf{S}$ follows a Chi-squared distribution. By adapting $\mathbf{S}$, the layer learns the correct kernel type.

While this section serves as an introduction of Fastfood, it was originally proposed as a fast way of computing random features to approximate kernels. This perspective is expanded in the following section.

3.9.4.2 A view from kernels

There is a duality between inner products of features and kernels. This duality can be used to design neural network modules using kernels and vice-versa.

For computational reasons, it is often desired to determine the features associated with a kernel. Working with features is preferable when the kernel matrix $\mathbf{K}$ is dense and large.

Fortunately, Monte Carlo methods can be used to approximate the expected value in the Bochner's Lemma, and hence approximate the kernel $k(\mathbf{x}, \mathbf{x}')$ with an inner product of stacked cosine and sine features.

$$\begin{aligned} k(\mathbf{x}, \mathbf{x}') &= \int \alpha e^{-i\mathbf{w}^\top(\mathbf{x}-\mathbf{x}')}\, p(\mathbf{w}) d\mathbf{w} \\ &= \alpha \mathbb{E}_{\mathbf{w}}[\cos(\mathbf{w}^\top \mathbf{x})\cos(\mathbf{w}^\top \mathbf{x}') + \sin(\mathbf{w}^\top \mathbf{x})\sin(\mathbf{w}^\top \mathbf{x}')] \end{aligned}$$

Specifically, suppose n vectors $i.i.d.$ are sampled from $p(\mathbf{w})$ and are collected in a matrix $\mathbf{W} = (\mathbf{w}_1, \dots \mathbf{w}_n)^\top$. The kernel can then be approximated as the inner-product of the following random features

$$\boldsymbol{\phi}_{\text{rbf}}(\mathbf{W}\mathbf{x}) = \sqrt{\alpha/n}\cos(\mathbf{W}\mathbf{x}), \sin(\mathbf{W}\mathbf{x})^\top \tag{3.17}$$

That is, $\boldsymbol{\phi}(\mathbf{W}\mathbf{x})$ is the neural network module, consisting of a linear layer $\mathbf{W}\mathbf{x}$ and entry-wise nonlinearities (cosine and sine in the above equation), that corresponds to a particular implicit kernel function.

Approximating a given kernel function with random features requires the specification of a sampling distribution $p(\mathbf{w})$. Such distributions have been derived for many popular kernels. For example, if one wants the implicit kernel to be a squared exponential kernel,

$$k(\mathbf{x}, \mathbf{x}') = \exp -\frac{\mathbf{x} - \mathbf{x}'^2}{2\ell^2} \tag{3.18}$$

it is known that the distribution $p(\mathbf{w})$ must be Gaussian: $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \text{diag}(\boldsymbol{\ell}^2)^{-1})$. In other words, if the rows of $\mathbf{W}$ are drawn from this Gaussian distribution and the equation (3.17)

is used to implement a neural module, then it results in an implicit approximation to a squared exponential kernel.

The Fastfood transform was introduced to replace $\mathbf{Wx}$ in Equation 3.17 with $\mathbf{SHG\Pi HBx}$, thus decreasing the computational and storage costs.

3.9.5 The proposed method

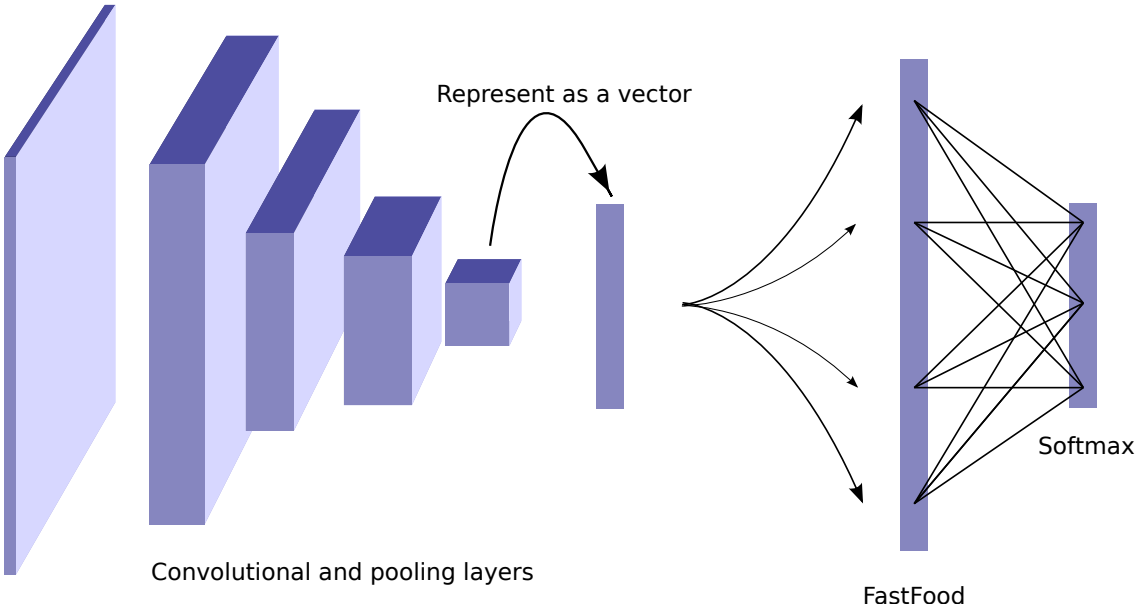


Figure 3.1: The structure of a deep fried convolutional network. The convolution and pooling layers are identical to those in a standard convnet. However, the fully-connected layers are replaced with the Adaptive Fastfood transform.

This work proposes to greatly reduce the number of parameters of the fully-connected layers by replacing them with an Adaptive Fastfood transform followed by a nonlinearity. This new architecture is called a deep fried convolutional network. An illustration of the architecture is shown in Figure 3.1.

In principle, one could also apply the Adaptive Fastfood transform to the softmax classifier. However, reducing the memory cost of this layer is already well studied; for example, [Sainath et al., 2013] show that low-rank matrix factorisation can be applied during training to reduce the size of the softmax layer substantially. Importantly, they also show that training a low-rank factorisation for the internal layers performs poorly, which agrees with the results of [Denil et al., 2013b]. Therefore it is reasonable to focus on reducing the size of the internal layers.

MNIST (joint)	Error	Params
Fastfood 1024 (ND)	0.83%	38,821
Adaptive Fastfood 1024 (ND)	0.86%	38,821
Fastfood 2048 (ND)	0.90%	52,124
Adaptive Fastfood 2048 (ND)	0.92%	52,124
Fastfood 1024	**0.71**%	38,821
Adaptive Fastfood 1024	0.72%	38,821
Fastfood 2048	**0.71**%	52,124
Adaptive Fastfood 2048	0.73%	52,124
Reference Model	**0.87**%	430,500

Table 3.1: MNIST jointly trained layers: comparison between a reference convolutional network with one fully-connected layer (followed by a densely-connected softmax layer) and two deep fried networks on the MNIST dataset. Numbers indicate the number of features used in the Fastfood transform. The results tagged with *(ND)* were obtained without dropout.

3.9.6 MNIST experiments

The first problem is the classical MNIST optical character recognition task. This simple task serves as an easy proof of concept for the method, and contrasting the results in this section with later experiments gives insights into the behaviour of the Adaptive Fastfood transform at different scales.

Caffe implementation of the LeNet convolutional network[⋆] is used as a reference model. It achieves an error rate of 0.87% on the MNIST dataset.

I jointly train all layers of the deep fried network (including convolutional layers) from scratch. I compare both the adaptive and non-adaptive Fastfood transforms using 1024 and 2048 features. For the non-adaptive transforms I report the best performance achieved by varying the standard deviation of the random Gaussian matrix over the set $0.001, 0.005, 0.01, 0.05$, and for the adaptive variant I learn these parameters by backpropagation as described in Section 3.9.3.

The results of the MNIST experiment are shown in Table 3.1. Because the width of the deep fried network is substantially larger than the reference model, I also experimented with adding dropout in the model, which increased performance in the deep fried case. Deep fried networks can obtain high accuracy using only a small fraction of parameters of the original network (11 times reduction in the best case). Interestingly, there seems to be no benefit from adaptation in this experiment, with the more powerful adaptive models performing equivalently or worse than their non-adaptive counterparts; however, this should

⋆ `https://github.com/BVLC/caffe/blob/master/examples/mnist/lenet.prototxt`

be contrasted with the ImageNet results reported in the following sections.

3.9.7 ImageNet experiments

I now examine how deep fried networks behave in a more realistic setting with a much larger dataset and many more classes. Specifically, I use the ImageNet ILSVRC-2012 dataset which has 1.2M training examples and 50K validation examples distributed across 1000 classes.

I use the Caffe ImageNet model* as the reference model in these experiments [Jia et al., 2014]. This model is a modified version of AlexNet [Krizhevsky et al., 2012], and achieves 42.6% top-1 error on the ILSVRC-2012 validation set. The initial layers of this model are a cascade of convolution and pooling layers with interspersed normalisation. The last several layers of the network take the form of an MLP and follow a 9216–4096–4096–1000 architecture. The final layer is a logistic regression layer with 1000 output classes. All layers of this network use the ReLU nonlinearity, and dropout is used in the fully-connected layers to prevent overfitting.

There is a total of 58,649,184 parameters in the reference model, of which 58,621,952 are in the fully-connected layers and only 27,232 are in the convolutional layers. The parameters of fully-connected layer take up 99.9% of the total number of parameters. It shows that the Adaptive Fastfood transform can be used to reduce the number of parameters in this model substantially.

ImageNet (fixed)	Error	Params
Dai et al. [Dai et al., 2014a]	44.50%	163M
Fastfood 16	50.09%	16.4M
Fastfood 32	50.53%	32.8M
Adaptive Fastfood 16	45.30%	16.4M
Adaptive Fastfood 32	**43.77%**	32.8M
MLP	**47.76%**	58.6M

Table 3.2: Imagenet fixed convolutional layers: MLP indicates that I re-train 9216–4096–4096–1000 MLP (as in the original network) with the convolutional weights pretrained and fixed. The proposed method is represented by *Fastfood 16* and *Fastfood 32*, using 16,384 and 32,768 Fastfood features respectively. [Dai et al., 2014a] report results of max-voting of 10 transformations of the test set.

* `https://github.com/BVLC/caffe/tree/master/models/bvlc_reference_caffenet`

3.9.7.1 Fixed feature extractor

Previous work on applying kernel methods to ImageNet has focused on building models on features extracted from the convolutional layers of a pre-trained network [Dai et al., 2014a]. This setting is less general than training a network from scratch but does mirror the common use case where a convolutional network is first trained on ImageNet and used as a feature extractor for a different task.

To compare Adaptive Fastfood transform directly to this previous work, I extract features from the final convolutional layer of a pre-trained reference model and train an Adaptive Fastfood transform classifier using these features. Although the reference model uses two fully-connected layers, I investigate replacing these with only a single Fastfood transform. I experiment with two sizes for this transform: *Fastfood 16* and *Fastfood 32* using 16,384 and 32,768 Fastfood features respectively. Since the Fastfood transform is a composite module, it is possible to apply dropout between any of its layers. In the experiments reported here, I applied dropout after the $\mathbf{\Pi}$ matrix and after the $\mathbf{S}$ matrix. Dropout is also applied to the last convolutional layer (that is, before the $\mathbf{B}$ matrix).

I also train an MLP with the same structure as the top layers of the reference model for comparison. In this setting, it is important to compare against the re-trained MLP rather than the jointly trained reference model, as training on features extracted from fixed convolutional layers typically leads to lower performance than joint training [Yosinski et al., 2014].

The results of the fixed feature experiment are shown in Table 3.2. Following [Yosinski et al., 2014] and [Dai et al., 2014a] I observe that training on ImageNet activations produces significantly lower performance than of the original, jointly trained network. Nonetheless, deep fried networks can outperform both the re-trained MLP model as well as the results in [Dai et al., 2014a] while using fewer parameters.

In contrast with the MNIST experiment, here I find that the Adaptive Fastfood transform provides a significant performance boost over the non-adaptive version, improving top-1 performance by 4.5-6.5%.

3.9.7.2 Jointly trained model

Finally, I train a deep fried network from scratch on ImageNet. With 16,384 features in the Fastfood layer the top-1 validation performance is reduced by less than 0.3%, but the number of parameters in the network is reduced from 58.7M to 16.4M which corresponds to a factor of 3.6x. When further increasing the number of features to 32,768, the performance

is 0.6% better than the reference model while using approximately half as many parameters. Results from this experiment are shown in Table 3.3.

ImageNet (joint)	Error	Params
Fastfood 16	46.88%	16.4M
Fastfood 32	46.63%	32.8M
Adaptive Fastfood 16	42.90%	16.4M
Adaptive Fastfood 32	**41.93%**	32.8M
Reference Model	**42.59%**	58.7M

Table 3.3: Imagenet jointly trained layers. The proposed method is represented by *Fastfood 16* and *Fastfood 32*, using 16,384 and 32,768 Fastfood features respectively. *Reference Model* shows the accuracy of the jointly trained Caffe reference model.

Nearly all of the parameters of the deep fried network reside in the final softmax regression layer, which still uses a dense linear transformation, and accounts for more than 99% of the parameters of the network. It is a side effect of the large number of classes in ImageNet. For a dataset with fewer classes the advantage of deep fried convolutional networks would be even greater. Moreover, as shown by [Denil et al., 2013b, Sainath et al., 2013], the last layer often contains considerable redundancy. Please also note that any of the techniques from [Collins and Kohli, 2014a, Chen et al., 2015] could be applied to the final layer of a deep fried network to reduce memory consumption at test time further. It is illustrated with a low-rank matrix factorisation in the following section.

3.9.8 Comparison with Post Processing

This section provides a comparison to some existing works on reducing the number of parameters in a convolutional neural network. The techniques I compare against are *post-processing* techniques, which start from a fully trained model and attempt to compress it, whereas the proposed method trains the compressed network from scratch.

Matrix factorisation is the most common method for compressing neural networks, and has proven to be very effective. Given the weight matrix of fully-connected layers $\mathbf{W} \in \mathbb{R}^{d \times n}$, it can be factorised as

$$\mathbf{W} = \mathbf{U}\mathbf{S}\mathbf{V}^\top$$

where $\mathbf{U} \in \mathbb{R}^{d \times d}$ and $\mathbf{V} \in \mathbb{R}^{n \times n}$ and $\mathbf{S}$ is a $d \times n$ diagonal matrix. To reduce the number of parameters all but the k largest singular values are truncated, leading to the approximation: $\mathbf{W} \approx \tilde{\mathbf{U}}\tilde{\mathbf{V}}^\top$, where $\tilde{\mathbf{U}} \in \mathbb{R}^{d \times k}$ and $\tilde{\mathbf{V}} \in \mathbb{R}^{n \times k}$ and $\mathbf{S}$ has been absorbed into the other two factors. If k is sufficiently small then storing $\tilde{\mathbf{U}}$ and $\tilde{\mathbf{V}}$ is less expensive than storing $\mathbf{W}$ directly, and this parameterisation is still learnable.

It has been shown that directly training a factorised representation directly leads to poor performance [Denil et al., 2013b] (although it does work when applied only to the final logistic regression layer [Sainath et al., 2013]). However, first training a full model, then performing an SVD of the weight matrices followed by a fine-tuning phase preserves much of the performance of the original model [Xue et al., 2013]. I compare deep fried approach to SVD followed by fine-tuning and show that the former approach achieves better performance per parameter despite training a compressed parameterisation from scratch. I also compare against a post-processed version, where I train a deep fried convnet and then apply SVD plus fine-tuning to the final softmax layer, which further reduces the number of parameters.

Results of these post-processing experiments are shown in Table 3.4. For the SVD decomposition of each of the three fully-connected layers in the reference model I set $k = \min(d, n)/2$ in SVD-half and $k = \min(d, n)/4$ in SVD-quarter. SVD-half-F and SVD-quarter-F mean that the model has been fine-tuned after the decomposition.

There is a 1% drop in accuracy for SVD-half and 3.5% drop for SVD-quarter. Even though the increase in the error for the SVD can be mitigated by fine-tuning (the drop decreases to 0.1% for SVD-half-F and 1.3% for SVD-quarter-F), deep fried convnets still perform better both in terms of the accuracy and the number of parameters.

Applying a rank 600 SVD followed by fine-tuning to the final softmax layer of the Adaptive Fastfood 32 model removes an additional 12.5M parameters at the expense of $\sim$0.7% top-1 error.

For reference, I also include the results of Collins and Kohli [Collins and Kohli, 2014a], who pre-train a full network and use a sparsity regularizer during fine-tuning to encourage connections in the fully-connected layers to be zero. They can achieve a significant reduction in the number of parameters this way. However, the performance of their compressed network suffers when compared to the reference model. Another drawback of this method is that using sparse weight matrices requires additional overhead to store the indexes of the non-zero values. The index storage takes up space and using sparse representation is better than using a dense matrix only when the number of nonzero entries is small.

3.9.9 Conclusion

Many methods have been advanced to reduce the size of convolutional networks at test time. In contrast to this trend, the Adaptive Fastfood transform introduced in this work is end-to-end differentiable and hence it enables us to attain reductions in the number of parameters even at train time.

Model	Error	Params	Ratio
Collins and Kohli [Collins and Kohli, 2014a]	44.40%	—	—
SVD-half	43.61%	46.6M	0.8
SVD-half-F	42.73%	46.6M	0.8
Adaptive Fastfood 32	**41.93%**	32.8M	0.55
SVD-quarter	46.12%	23.4M	0.5
SVD-quarter-F	43.81%	23.4M	0.5
Adaptive Fastfood 16	42.90%	16.4M	0.28
Ada. Fastfood 32 (F-600)	42.61%	20.3M	0.35
Reference Model	42.59%	58.7M	1

Table 3.4: Comparison with other methods. The result of [Collins and Kohli, 2014a] is based on the Caffe AlexNet model (similar but not identical to the Caffe reference model) and achieves $\sim$4x reduction in memory usage, (slightly better than Fastfood 16 but with a noted drop in performance). SVD-half: 9216-2048-4096-2048-4096-500-1000 structure. SVD-quarter: 9216-1024-4096-1024-4096-250-1000 structure. `F` means after fine-tuning.

Deep fried convnets capitalise on the proposed Adaptive Fastfood transform to achieve a substantial reduction in the number of parameters without sacrificing predictive performance on MNIST and ImageNet. They also compare favourably against simple test-time low-rank matrix factorisation schemes.

The experiments have also cast some light on the issue of random versus adaptive weights. The structured random transformations developed in the kernel literature perform very well on MNIST without any learning; however, when moving to ImageNet, the benefit of adaptation becomes clear, as it allows us to achieve substantially better performance. It is an important point which illustrates the importance of learning which would not have been visible from experiments only on small datasets.

The Fastfood transform allows for a theoretical reduction in computation from $\mathcal{O}(nd)$ to $\mathcal{O}(n \log d)$. However, the computation in convolutional neural networks is dominated by the convolutions, and hence deep fried convnets are not necessarily faster in practice.

It is clear looking at out results on ImageNet in Table 2 that the remaining parameters are mostly in the output softmax layer. The comparative experiment in Section 7 showed that the matrix of parameters in the softmax can be easily compressed using the SVD, but many other methods could be used to achieve this. One avenue for future research involves replacing the softmax matrix, at train and test times, using the abundant set of techniques that have been proposed to solve this problem, including low-rank decomposition, Adaptive Fastfood, and pruning.

The development of GPU optimised Fastfood transforms that can be used to replace linear layers in arbitrary neural models would also be of great value to the entire research

community, given the ubiquity of fully-connected layers.

3.10 ACDC: A Structured Efficient Linear Layer

3.10.1 Introduction

The linear layer is the central building block of nearly all modern neural network models. A notable exception to this is the convolutional layer, which has been extremely successful in computer vision; however, even convolutional networks typically feed into one or more linear layers after processing by convolutions. Other specialised network modules including LSTMs [Hochreiter and Schmidhuber, 1997b], GRUs [Cho et al., 2014], the attentional mechanisms used for image captioning [Xu et al., 2015a] and machine translation [Bahdanau et al., 2015], reading in Memory Networks [Sukhbaatar et al., 2015], and both reading and writing in Neural Turing Machines [Graves et al., 2015], are all built from compositions of linear layers and nonlinear modules, such as sigmoid, softmax and ReLU layers.

The linear layer is essentially a matrix-vector operation, where the input $\mathbf{x}$ is scaled with a matrix of parameters $\mathbf{W}$ as follows:

$$\mathbf{y} = \mathbf{x}\mathbf{W} \tag{3.19}$$

When the number of inputs and outputs is N, the number of parameters stored in $\mathbf{W}$ is $\mathcal{O}(N^2)$. It also takes $\mathcal{O}(N^2)$ operations to compute the output $\mathbf{y}$.

Despite the ubiquity and convenience of linear layers, their $\mathcal{O}(N^2)$ size is extremely uneconomical. Indeed, several studies focusing on feedforward perceptrons and convolutional networks have shown that the parameterisation of linear layers is hugely wasteful, with up to 95% of the parameters being redundant [Denil et al., 2013b, Gong et al., 2014, Sainath et al., 2013].

Given the importance of this research topic, we have witnessed a recent explosion of works introducing structured efficient linear layers (SELLs). I adopt the following notation to describe SELLs within a common framework

$$\mathbf{y} = \mathbf{x}\mathbf{\Phi} = \mathbf{x}\mathbf{\Phi}(\mathbf{D}, \mathbf{P}, \mathbf{S}, \mathbf{B}) \tag{3.20}$$

The capital bold symbol $\mathbf{D}$ is reserved for diagonal matrices, $\mathbf{P}$ for permutations, $\mathbf{S}$ for sparse matrices, and $\mathbf{B} \in \{\mathbf{F}, \mathbf{H}, \mathbf{C}\}$ for bases such as Fourier, Hadamard and Cosine transforms respectively. In this setup, the parameters are typically in the diagonal or sparse

Torch implementation of ACDC is available at `https://github.com/mdenil/acdc-torch`

entries of the matrices $\mathbf{D}$ and $\mathbf{S}$. Sparse matrices aside, the computational cost of most SELLs is $\mathcal{O}(N \log N)$, while the number of parameters is reduced from $\mathcal{O}(N^2)$ to a mere $\mathcal{O}(N)$. These costs are a consequence of the facts that only the diagonal matrices require storage, and that Fourier, Hadamard transform or Discrete Cosine transform can be efficiently computed in $\mathcal{O}(N \log N)$ steps.

Often the diagonal and sparse matrices have fixed random entries. When this is the case, I will use tildes to indicate this fact (*e.g.*, $\tilde{\mathbf{D}}$).

The first example of SELL is the Fast Random Projections method of [Ailon and Chazelle, 2009]

$$\mathbf{\Phi} = \tilde{\mathbf{D}}\mathbf{H}\tilde{\mathbf{S}} \tag{3.21}$$

Here, the sparse matrix $\tilde{\mathbf{S}}$ has Gaussian entries, the diagonal $\tilde{\mathbf{D}}$ has $\{+1, -1\}$ entries drawn independently with probability $1/2$, and $\mathbf{H}$ is the Hadamard matrix. The embeddings generated by this SELL preserve metric information with high probability, as formalised by the theory of random projections.

Fastfood [Le et al., 2013], the second SELL example, extends fast random projections as follows

$$\mathbf{\Phi} = \tilde{\mathbf{D}}_1\mathbf{H}\mathbf{P}\tilde{\mathbf{D}}_2\mathbf{H}\tilde{\mathbf{D}}_3 \tag{3.22}$$

In [Yang et al., 2015], the authors introduce an adaptive variant of Fastfood, with the random diagonal matrices replaced by diagonal matrices of parameters, and show that it outperforms the random counterpart when applied to the problem of replacing one of the fully-connected layers of a convolutional neural network for ImageNet [Jia et al., 2014]. Interestingly, while the random variant is competitive in simple applications (MNIST), the adaptive variant has a considerable advantage in more demanding applications (ImageNet).

The adaptive SELLs discussed subsequently, including Adaptive Fastfood and the alternatives, are end to end differentiable. They require only $\mathcal{O}(N)$ parameters and $\mathcal{O}(N \log N)$ operations in both the forward and backward passes of backpropagation. These benefits can be achieved both at train and test time.

[Cheng et al., 2015] introduced a SELL consisting of the product of a circulant matrix ($\mathbf{R}$) and a random diagonal matrix ($\tilde{\mathbf{D}}_1$). Since circulant matrices can be diagonalised with the discrete Fourier transform [Golub and Van Loan, 1996], this SELL falls within the general notation

$$\mathbf{\Phi} = \tilde{\mathbf{D}}_1\mathbf{R} = \tilde{\mathbf{D}}_1\mathbf{F}\mathbf{D}_2\mathbf{F}^{-1} \tag{3.23}$$

[Sindhwani et al., 2015] introduced a Toeplitz-like structured transform, within the framework of displacement operators. Since Toeplitz matrices can be "embedded" in circulant

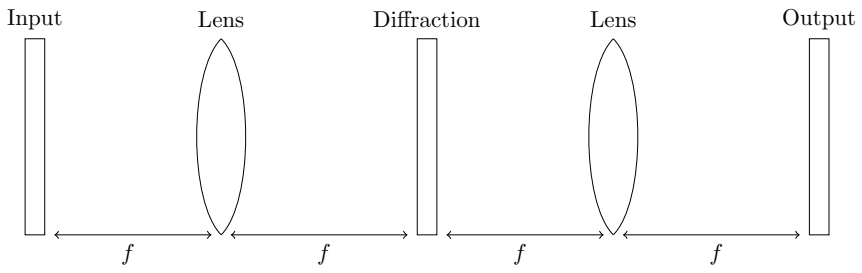


Figure 3.2: Example of a $4f$ system. This system implements the multiplication of an optical signal by a circulant matrix $\mathbf{FDF}^{-1}$. The lenses apply Fourier transforms to the signal and the diffraction element applies the diagonal multiplication.

matrices, they can also be diagonalised with the discrete Fourier transform [Golub and Van Loan, 1996].

In this work I introduce a SELL that could be thought of as an adaptive variant of the method of [Cheng et al., 2015]. Also instead of using a (single) shallow SELL as in previous works [Yang et al., 2015, Cheng et al., 2015, Sindhwani et al., 2015], I consider deep SELLs

$$\mathbf{\Phi} = \prod_{k=1}^{K} \mathbf{A}_k \mathbf{F} \mathbf{D}_k \mathbf{F}^{-1} \tag{3.24}$$

Here, $\mathbf{A}$ is also a diagonal matrix of parameters, but a different symbol is used to emphasise that $\mathbf{A}$ scales the signal in the original domain while $\mathbf{D}$ scales it in the Fourier domain.

While adaptive SELLs perform better than their random counterparts in practice, there is a lack of theory for adaptive SELLs. Moreover, the empirical studies of recent adaptive SELLs have many deficiencies. For instance, it is often not clear how performance varies depending on implementation, and many critical details such as initialisation and the treatment of biases are typically obviated. In addition, the gains are often demonstrated in models of different size, making objective comparison very difficult.

In addition to demonstrating good performance replacing the fully-connected layers of CaffeNet, I present a theoretical approximation guarantee for deep SELL in Section 3.10.4. I also discuss the crucial issue of implementing deep SELLs efficiently in modern GPU architectures in Section 3.10.6. This engineering contribution is essential as many of the recently proposed methods for accelerating linear layers often fail to take into account the attributes and limitations of GPUs, and hence fail to be adopted.

3.10.2 Lightning fast deep SELL

Deep SELL (equation (3.24)) offers several possibilities for analogue physical implementation. Given the great demand for fast low energy neural networks, the possibility of har-

nessing physical phenomena to perform efficient computation in deep networks is worthy of consideration.

In the Fourier optics field, it is well-known that the two-dimensional Fourier transform can be implemented with a paraxial optical system consisting of a lens of focal length f in free space. In this setup, known as a $2f$ system, a waveform in the frontal focal plane of the lens, viewed as a two-dimensional complex array, is transformed to another one in the focal plane behind the lens that corresponds to the Fourier transform of the array. A $4f$ system is obtained by placing a diffractive element in between two $2f$ systems at a distance f from each (shown in Figure 3.2).

Every circulant matrix $\mathbf{R} = \mathbf{FDF}^{-1}$ can be realised optically using a $4f$ system, with the transformation by the diffractive optical device corresponding to the multiplication by the complex diagonal matrix $\mathbf{D}$ [Reif and Tyagi, 1997, Müller-Quade et al., 1998, Huhtanen, 2008, Schmid et al., 2000]. Moreover, paraxial diffractive optical systems with consecutive products of circulant and diagonal matrices can factor a complex matrix into products of diagonal and circulant matrices [Müller-Quade et al., 1998, Huhtanen and Perämäki, 2015]. Hence, in principle, the mapping of equation (3.24) can be implemented with optical elements.

In a separate research community, [Hermans and Vaerenbergh, 2015] recently discussed using waves in a trainable medium for learning linear layers by backpropagation, and suggested a potential implementation using an integrated photonics chip. The nanophotonic chip consists of a cascade of unitary transformations of the optical signals interleaved with tuneable waveguides (phase shifters). [Hermans and Vaerenbergh, 2015] present an abstraction of this chip. In particular let $\mathbf{o}$ and $\mathbf{o}'$ represent the optical fields at the input and output waveguides. Then the chip implements the following transformation

$$\mathbf{o}' = \prod_{k=1}^{K} \mathbf{D}_k \mathbf{U}_k \mathbf{o} \tag{3.25}$$

where $\mathbf{U}_k$ is a unitary transformation of the signal and $\mathbf{D}_k$ is a diagonal matrix $\mathbf{D}_k = \mathrm{diag}(\exp(j\boldsymbol{\varphi}_k))$ with tuneable phase shifts $\boldsymbol{\varphi}_k$. By restricting the diagonal matrices in equation (3.24) to be of this complex form, the circulant $\mathbf{R} = \mathbf{FDF}^{-1}$ is unitary, so there is an equivalence between equations (3.24) and (3.25). This points to a potential nanophotonic implementation of complex deep SELL.

More recently, [Saade et al., 2015] disclosed an invention that performs optical analogue random projections.

3.10.3 Further related works

The literature on this topic is vast, and consequently, this section only aims to capture some of the significant trends. I would like to recommend the related work sections of the papers cited in the previous and the present section for further details.

As mentioned earlier, many studies have shown that the parameterisation of linear layers is exceptionally wasteful [Denil et al., 2013b, Gong et al., 2014, Sainath et al., 2013]. In spite of this redundancy, there has been little success in improving the linear layer. Natural extensions, such as low-rank factorisations, lead to poor performance when they are trained end to end. For instance, [Sainath et al., 2013] demonstrate significant improvements in reducing the number of parameters of the output softmax layers, but only modest improvements for the hidden linear layers.

Several methods based on low-rank decomposition and sparseness have been proposed to eliminate parameter redundancy at test time, but they provide only a partial solution as the full network must be instantiated during training [Collins and Kohli, 2014a, Xue et al., 2013, Blundell et al., 2015, Liu et al., 2015, Han et al., 2015b]. That is, these approaches require training the original full model. Hashing techniques have been proposed to reduce the number of parameters [Chen et al., 2015, Bakhtiary et al., 2015]. Hashes have irregular memory access patterns and, consequently, good performance on large GPU-based platforms is an open problem. Distillation [Hinton et al., 2015, Romero et al., 2015] also offers a way of compressing neural networks, as a post-processing step.

[Novikov et al., 2015] use a multi-linear transform (Tensor-Train decomposition) to attain significant reductions in the number of parameters in some of the linear layers of convolutional networks.

3.10.4 Deep SELL

I define a single component of deep SELL as $\mathrm{AFDF}(\mathbf{x}) = \mathbf{xAFDF}^{-1}$, where $\mathbf{F}$ is the Fourier transform and $\mathbf{A}, \mathbf{D}$ are complex diagonal matrices. It is straightforward to see that the AFDF transform is not sufficient to express an arbitrary linear operator $\mathbf{W} \in \mathbb{C}^{n \times n}$. An AFDF transform has $2n$ degrees of freedom, whereas an arbitrary linear operator has n^2 degrees of freedom.

To this end, the attention should be turned to studying compositions of AFDF transforms. By composing AFDF transforms one can boost the number of degrees of freedom, and expect that any linear operator could be constructed as a composition of sufficiently many AFDF transforms. In the following, I show that this is indeed possible and that a bounded number of AFDF transforms is sufficient.

The order-K AFDF transformation is the composition of K consecutive AFDF operations with (optionally) different $\mathbf{A}$ and $\mathbf{D}$ matrices. An order-K complex AFDF transformation is written as follows

$$\mathbf{y} = \mathrm{AFDF}_K(\mathbf{x}) = \mathbf{x}\left[\prod_{k=1}^{K} \mathbf{A}_k \mathbf{F} \mathbf{D}_k \mathbf{F}^{-1}\right] \tag{3.26}$$

It is correct to assume without loss of generality, that $\mathbf{A}_1 = \mathbf{I}$ so that $\mathrm{AFDF}_1(\mathbf{x}) = \mathbf{x}\mathbf{F}\mathbf{D}_1\mathbf{F}^{-1}$.

For the analysis it will be convenient to rewrite the AFDF transformation in a different way, which is referred to as the *optical presentation*.

If $\mathbf{y} = \mathrm{AFDF}_K(\mathbf{x})$ then define the optical presentation of an order-K AFDF transform as

$$\hat{\mathbf{y}} = \hat{\mathbf{x}}\left[\prod_{k=1}^{K-1} \mathbf{D}_k \mathbf{R}_{k+1}\right]\mathbf{D}_K$$

where $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ are the Fourier transforms of $\mathbf{x}$ and $\mathbf{y}$, and $\mathbf{R}_{k+1} = \mathbf{F}^{-1}\mathbf{A}_{k+1}\mathbf{F}$.

The matrix $\mathbf{R} = \mathbf{F}^{-1}\mathbf{A}\mathbf{F}$ is circulant. This follows from the duality between convolution in the spatial domain and pointwise multiplication in the Fourier domain.

The optical presentation shows how the spectrum of $\mathbf{x}$ is related to the spectrum of $\mathbf{y}$. Importantly, it shows that one can express an order-K AFDF transform as a linear operator in Fourier space that is composed of a product of circulant and diagonal matrices. Transformations of this type are well studied in the Fourier optics literature, as they can be realised with cascades of lenses.

Of particular relevance is the main result of [Huhtanen and Perämäki, 2015] which states that almost all (in the Lebesgue sense) matrices $\mathbf{M} \in \mathbb{C}^{N\times N}$ can be factored as

$$\mathbf{M} = \left[\prod_{i=1}^{N-1} \mathbf{D}_{2i-1}\mathbf{R}_{2i}\right]\mathbf{D}_{2N-1}$$

where $\mathbf{D}_{2j-1}$ is diagonal and $\mathbf{R}_{2j}$ is circulant. This factorisation corresponds exactly to the optical presentation of an order-N AFDF transform. Therefore one can conclude the following: An order-N AFDF transform is sufficient to approximate any linear operator in $\mathbb{C}^{N\times N}$ to arbitrary precision.

Proof. Every AFDF transform has an optical presentation and by the main result of [Huhtanen and Perämäki, 2015] operators of this type are dense in $\mathbb{C}^{N\times N}$.

□

3.10.5 ACDC: A practical deep SELL

Thus far I have focused on a complex SELL, where theoretical guarantees can be obtained. In practice it is useful to consider a real SELL instead. The real version of AFDF_K, denoted ACDC_K has the same form as Equation (3.26), with complex diagonals replaced with real diagonals, and Fourier transforms replaced with Cosine Transforms. This change departs from the theory of Section 3.10.4; however, the experiments show that this does not appear to be a problem in practice.

The reasons for considering ACDC over AFDF are purely practical.

1. Most existing deep learning frameworks support only real numbers, and thus working with real-valued transformations simplifies the interface between SELL and the rest of the network.

2. Working with complex numbers effectively doubles the memory footprint of the transform itself, and more importantly, of the activations that interact with it.

The importance of the second point should not be underestimated since the computational complexity of SELL is quite low, a typical GPU implementation will be bottlenecked by the overhead of moving data through the GPU memory hierarchy. Reducing the amount of data to be moved allows for a significantly faster implementation. These concerns are discussed in more detail in Section 3.10.6.

This work uses the DCT (type II) matrix with entries

$$c_{nk} = \sqrt{\frac{2}{N}} \left[\epsilon_k \cos\left(\frac{\pi(2n+1)k}{2N} \right) \right] \tag{3.27}$$

for $n, k = 0, 1, \ldots, N$, and where $\epsilon_k = 1/\sqrt{2}$ for $k = 0$ or $k = N$ and $\epsilon_k = 1$ otherwise. DCT matrices are real and orthogonal: $\mathbf{C}^{-1} = \mathbf{C}^T$. Moreover, the DCTs are separable transforms. That is, the DCT of a multi-dimensional signal can be decomposed in terms of successive DCTs of the appropriate one-dimensional components of the signal. The DCT can be computed efficiently using the Fast Fourier Transform (FFT) algorithm (or the specialised fast cosine transform).

Denoting $\mathbf{h}_1 = \mathbf{x}_i\mathbf{A}$, $\mathbf{h}_2 = \mathbf{h}_1\mathbf{C}$, $\mathbf{h}_3 = \mathbf{h}_2\mathbf{D}$, $\mathbf{y}_i = \mathbf{h}_3\mathbf{C}^{-1}$, and $\mathbf{A} = \mathrm{diag}(\mathbf{a}), \mathbf{D} = \mathrm{diag}(\mathbf{d})$ one obtains the following derivatives in the backward pass

$$L\mathbf{d} = \mathbf{y}_i \mathbf{d} L\mathbf{y}_i = \mathbf{h}_2 \mathbf{D} \mathbf{d} \mathbf{h}_3 \mathbf{C}^{-1} \mathbf{h}_3 L\mathbf{y}_i = \mathrm{diag}(\mathbf{h}_2)\mathbf{C} L\mathbf{y}_i = \mathbf{h}_2 \odot \mathbf{C} L\mathbf{y}_i \tag{3.28}$$

$$L\mathbf{D} = \mathrm{diag}(L\mathbf{d}) \tag{3.29}$$

$$L\mathbf{a} = \mathbf{y}_i \mathbf{a} L\mathbf{y}_i = \mathbf{x}_i \mathbf{A} \mathbf{a} \mathbf{h}_1 \mathbf{C} \mathbf{h}_1 \mathbf{h}_2 \mathbf{D} \mathbf{h}_2 L\mathbf{h}_3 = \mathbf{x}_i \odot \mathbf{C}^{-1}\mathbf{d} \odot \mathbf{C} L\mathbf{y}_i \tag{3.30}$$

$$L\mathbf{A} = \mathrm{diag}(L\mathbf{a}) \tag{3.31}$$

$$L\mathbf{x}_i = \mathbf{y}_i \mathbf{x}_i L\mathbf{y}_i = \mathbf{x}_i \mathbf{A} \mathbf{x}_i L\mathbf{h}_1 = \mathbf{a} \odot \mathbf{C}^{-1}\mathbf{d} \odot \mathbf{C} L\mathbf{y}_i \tag{3.32}$$

3.10.6 Efficient implementation of ACDC

The processor used to benchmark the ACDC layer was an NVIDIA Titan X. The peak floating point throughput of the Titan X is 6605 GFLOPs, and the peak memory bandwidth is 336.5GB/s[⋆]. This gives an arithmetic intensity (FLOPs per byte) of approximately 20. In the ideal case, where there is enough parallelism for the GPU to hide all latencies, an algorithm with a higher arithmetic intensity than this would be expected to be floating point throughput bound, while an algorithm with lower arithmetic intensity would be expected to be memory throughput bound.

The forward pass of a single example through a size-N ACDC layer when calculated using 32-bit floating point arithmetic requires at least $24N$ bytes to be moved to and from main memory. Eight bytes per element for each of $\mathbf{A}$ and $\mathbf{D}$, four bytes per element for the input, and four bytes per element for the output. It also requires approximately $4N + 5N\log_2(N)$ floating point operations[⋆]. When batching, the memory transfers for $\mathbf{A}$ and $\mathbf{D}$ are expected to be cached as they are reused for each example in the batch, so for the purposes of calculating arithmetic intensity in the batched case, it is reasonable to discount them. The arithmetic intensity of a minibatch passing through an ACDC layer is therefore approximately

$$AI = (4 + 5\log_2(N))/8$$

For the values of N we are interested in ($128 \longrightarrow 16,384$) this arithmetic intensity varies between 4.9 and 9.3, indicating that the peak performance of a large ACDC layer with a large batch size is expected to be limited by the peak memory throughput of the GPU (336.5GB/s), and that optimisation of an ACDC implementation should concentrate on removing any extraneous memory operations.

⋆ http://www.geforce.co.uk/hardware/desktop-gpus/geforce-gtx-titan-x/specifications

⋆ http://www.fftw.org/speed/method.html

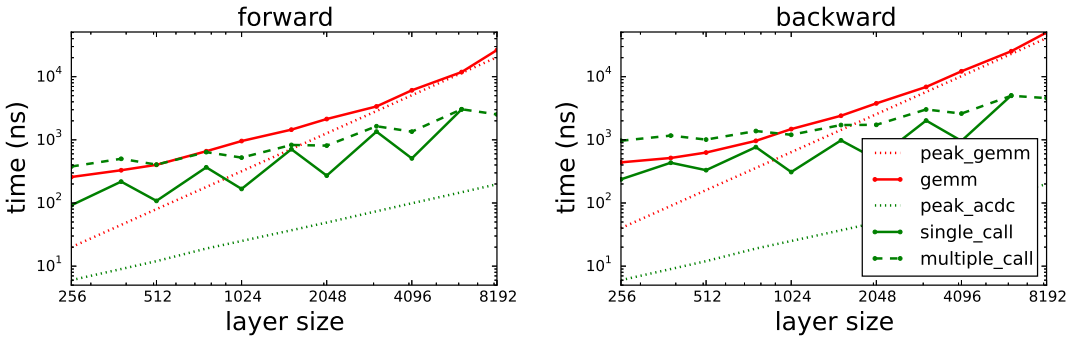


Figure 3.3: Comparison of the theoretical performance and the actual performance of ACDC implementations to an ordinary dense linear layer using a batch size of 128. Peak curves show maximum theoretical performance achievable by the hardware.

Two versions of ACDC have been implemented. One performs the ACDC in a single call, with the minimum of $8N$ bytes moved per layer (assuming perfect caching of $\mathbf{A}$ and $\mathbf{D}$). The other performs ACDC with multiple calls, with significantly more than $8N$ bytes moved per layer.

3.10.6.1 Single call implementation

To minimise traffic to and from main memory, intermediate loads or stores during the layer must be eliminated. To accomplish this kernel fusion is used to fuse all of the operations of ACDC into a single call, with intermediate values being stored in temporary low-level memory instead of main memory. This presents two challenges to the implementation.

Firstly, the size of the ACDC layer is limited by the availability of temporary memory on the GPU. This limits the size of the ACDC layer that can be calculated. It also has performance implications: the temporary memory used to store intermediate values in the computation is shared with the registers required for basic calculation, such as loop indices. The more of this space that is used by data, the fewer threads can fit on the GPU at once, limiting parallelism.

Secondly, the DCT and IDCT layers must be written by hand so that they can be efficiently fused with the linear layers. Implementations of DCT and IDCT are non-trivial, and a generic implementation able to handle any input size would be a large project in itself. For this reason, the implementation is constrained to power-of-two and multiples of large power-of-two layer sizes.

3.10.6.2 Multiple call implementation

While expected to be less efficient a multiple call implementation is both much simpler programmatically, and much more generically usable. Using the method of [Makhoul, 1980] it is possible to perform size-N DCTs and IDCTs using size-N FFTs. As such, the NVIDIA library cuFFT can be used to greatly simplify the code required, as well as achieve reasonable performance across a wide range of ACDC sizes. The procedure is as follows:

1. Multiply input by $\mathbf{A}$ and set up $\mathbf{C}_1$
2. Perform $\mathbf{C}_1$ using a C2C cuFFT call
3. Finalise $\mathbf{C}_1$, multiply by $\mathbf{D}$ and setup $\mathbf{C}_2$
4. Perform $\mathbf{C}_2$ using a C2C cuFFT call
5. Finalise $\mathbf{C}_2$

The total memory moved for this implementation is significantly higher as each call requires a load and a store for each element. The performance trade-off with the single call method is therefore one of parallelism against memory traffic.

3.10.6.3 Performance comparison

Figure 3.3 compares the speed of the single and multiple call implementations of ACDC against dense matrix-matrix multiplication for a variety of layer sizes.

It is clear that in both the forward and backward pass ACDC layers have a significantly lower runtime than fully-connected layers using dense matrices. Even if the matrix-matrix operations were running at peak, ACDC still would outperform them by up to 10 times.

As expected, the single call version of ACDC outperforms the multiple call version, although for smaller layer sizes the gap is larger. When the layer size increases the multiple call version suffers significantly more from small per-call overheads. Both single and multiple call versions of ACDC perform significantly worse on layer sizes that are not a power of two. This is because they rely on FFT operations, which are known to be more efficient when the input sizes are of lengths z^n, where z is a small integer[⋆].

While the backward pass of ACDC is expected to take approximately the same time as the forward pass, it takes noticeably longer. To compute the parameter gradients one needs the input into the $\mathbf{D}$ operation and the gradient of the output from the $\mathbf{A}$ operation. As the aim of the layer is to reduce memory footprint, it was decided instead to recompute these during the backward pass, increasing runtime while saving memory.

⋆ `http://docs.nvidia.com/cuda/cufft/#accuracy-and-performance`

3.10.7 Experiments

3.10.7.1 Linear layers

This section shows that it is viable to approximate linear operators using ACDC as predicted by the theory of Section 3.10.4. These experiments serve two purposes

1. They show that recovery of a dense linear operator by SGD is feasible in practice. The theory of Section 3.10.4 guarantees that it is possible to approximate any operator, but does not provide guidance on how to find this approximation. Additionally, [Huhtanen and Perämäki, 2015] suggest that this is a difficult problem.

2. They empirically validate that the decision to focus on ACDC over the complex AFDF does not introduce obvious difficulties into the approximation. The theory provides guarantees only for the complex case, and the experiments in this section suggest that restricting ourselves to real matrices is not a problem.

I investigate using ACDC on a synthetic linear regression problem

$$\mathbf{Y} = \mathbf{X}\mathbf{W}_{true} + \boldsymbol{\epsilon} \tag{3.33}$$

where $\mathbf{X}$ of size $10,000 \times 32$ and $\mathbf{W}_{true}$ of size 32×32 are both constructed by sampling their entries uniformly at random in the unit interval. Gaussian noise $\boldsymbol{\epsilon} \sim \mathcal{N}(0, 10^{-4})$ is added to the generated targets.

The results of approximating the operator $\mathbf{W}_{true}$ using ACDC_K for different values of K are shown in Figure 3.4. The theory of Section 3 predicts that, in the complex case, for a 32×32 matrix it should be sufficient to have 32 layers of ACDC to express an arbitrary $\mathbf{W}_{true}$.

It was found that initialisation of the matrices $\mathbf{A}$ and $\mathbf{D}$ to the identity $\mathbf{I}$, with Gaussian noise $\mathcal{N}(0, 10^{-2})$ added the diagonals to break symmetries, is essential for models having many ACDC layers. Furthermore, it was discovered that the initialisation is robust to the specification of the noise added to the diagonals.

The need for thoughtful initialisation is evident in Figure 3.4. With the right initialisation (leftmost plot), the approximation results of Section 3 are confirmed, with improved accuracy as the number of ACDC layers is increased. However, if standard strategies for initialising linear layers are used (rightmost plot), one observes inferior optimisation results as the number of ACDC layers increases.

This experiment suggests that fewer layers suffice to arrive at a reasonable approximation of the original $\mathbf{W}_{true}$ than what the theory guarantees. With neural networks in mind this is a very relevant observation. It is well-known that the linear layers of neural networks

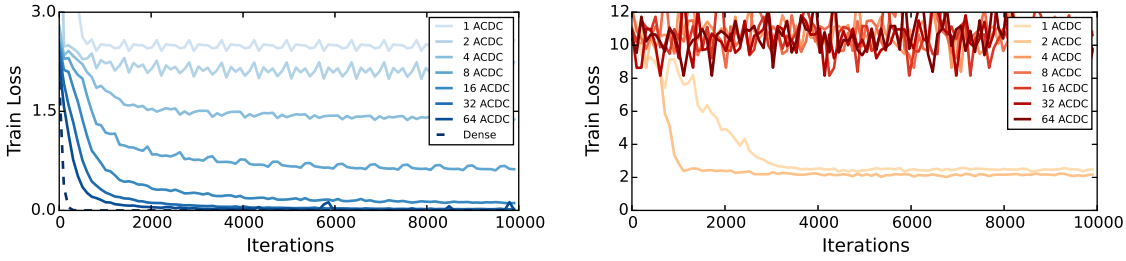


Figure 3.4: Training loss for the different number of ACDC layers compared to the loss for the dense matrix. **Left:** Initialisation: $\mathcal{N}(1, \sigma^2)$ with $\sigma = 10^{-1}$. **Right:** Initialisation: $\mathcal{N}(0, \sigma^2)$ with $\sigma = 10^{-3}$. Note the difference in scale on the y-axis.

are compressible, indicating that one does not need to express an arbitrary linear operator to achieve good performance. Instead, it is necessary to express only an interesting subset of matrices, and the result with 16 ACDC layers points to this being the case.

It is shown in Section 3.10.7.2 that by interspersing nonlinearities between ACDC layers in a convolutional network it is possible to use dramatically fewer ACDC layers than the theory suggests are needed while still achieving good performance.

3.10.7.2 Convolutional networks

In this section I investigate replacing the fully-connected layers of a deep convolutional network with a cascade of ACDC layers. In particular, I use the CaffeNet architecture* for ImageNet [Deng et al., 2009]. I target the two fully-connected layers located between features extracted from the last convolutional layer and the final logistic regression layer, which was replaced with 12 stacked ACDC transforms interleaved with ReLU nonlinearities and permutations. The permutations assure that adjacent SELLs are incoherent.

The model was trained using the SGD algorithm with learning rate 0.1 multiplied by 0.1 every $100{,}000$ iterations, momentum 0.65 and weight decay 0.0005. The output from the last convolutional layer was scaled by 0.1, and the learning rates for each matrix $\mathbf{A}$ and $\mathbf{D}$ were multiplied by 24 and 12. All diagonal matrices were initialised from $\mathcal{N}(1, 0.061)$ distribution. No weight decay was applied to $\mathbf{A}$ or $\mathbf{D}$. Additive biases were added to the matrices $\mathbf{D}$, but not to $\mathbf{A}$, as this sufficed to provide the ACDC layer with bias terms just before the ReLU nonlinearities. Biases were initialised to 0. To prevent the model from overfitting dropout regularisation was placed before each of the last 5 SELL layers with dropout probability equal to 0.1.

* `https://github.com/BVLC/caffe/tree/master/models/bvlc_reference_caffenet`

Test Time Post-Processing	Top-1 Err Increase	# of Param	Reduction
[Collins and Kohli, 2014a]	*1.81%*	*15.2M*	*x4.0*
[Han et al., 2015b]	0.00%	6.7M	x9
[Han et al., 2015a] (P+Q)	0.00%	$\sim$2.3M	x27
Train and Test Time Reduction			
[Cheng et al., 2015] (Circulant CNN 2)	0.40%	$>$ 16.3M	$<$ x3.8
*[Novikov et al., 2015] (TT4 FC FC)	0.30%	-	x3.9
**[Novikov et al., 2015] (TT4 TT4 FC)*	*1.30%*	-	*x7.4*
[Yang et al., 2015] (Fine-tuned SVD 1)	0.14%	46.6M	x1.3
[Yang et al., 2015] (Fine-tuned SVD 2)	*1.22%*	*23.4M*	*x2.0*
[Yang et al., 2015] (Adaptive Fastfood 16)	0.30%	16.4M	x3.6
ACDC	**0.67%**	**9.7M**	**x6.0**
CaffeNet Reference Model	0.00%	58.7M	x1.0

Table 3.5: Comparison of SELL with alternative factorisation methods achieving marginal performance drop on the ImageNet dataset. Entries in italics correspond to a $>$1.0% increase in the top-1 error. Entries marked with a star use VGG16, what makes them not directly comparable to SELL. Previous works have shown that typically it is possible to achieve $\sim$ 30% greater compression factors on VGG16 than on AlexNet-style architectures [Han et al., 2015a, Han et al., 2015b].

The resulting model arrives at 43.26% error which is only 0.67% worse when compared to the reference model, so SELL confidently stays within 1% of the performance of the original network. This result, as well as a comparison to several other works, are reported in Table 3.5.

The two fully-connected layers of CaffeNet, consisting of more than 41 million parameters, are replaced with SELL modules which contain a combined $165,888$ parameters. These results agree with the hypothesis that neural networks are over-parameterised formulated by [Denil et al., 2013b] and supported by [Yang et al., 2015]. At the same time such a tremendous reduction without significant loss of accuracy suggests that SELL is a powerful concept and a way to use parameters efficiently.

This approach is an improvement over Deep Fried Convnets [Yang et al., 2015] and other FastFood [Le et al., 2013] based transforms in the sense that the layers remain narrow and become deep (potentially interleaved with nonlinearities) as opposed to wide and shallow, while maintaining comparable or better performance. With narrower layers the final softmax layer requires substantially fewer parameters. Hence the resulting compression ratio is higher.

The experiment shows that ACDC transforms are an attractive building block for feed-forward convolutional architectures, that can be used as a structured alternative to fully-

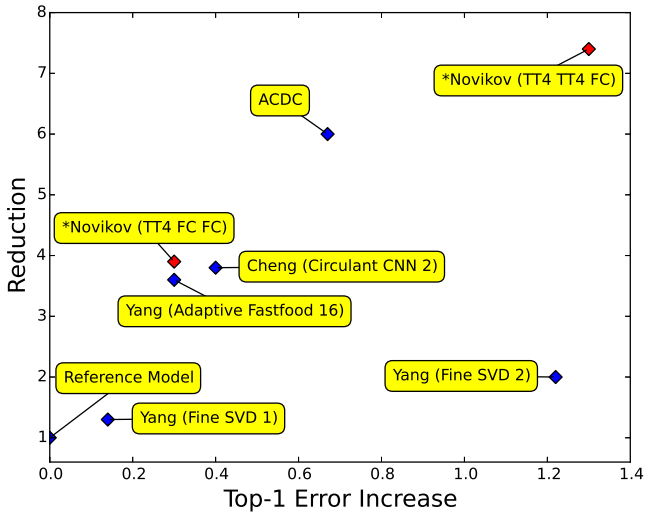


Figure 3.5: Visual comparison of the tradeoff between parameter and accuracy reduction for train time applicable SELLs. Red entries (marked with a star in the labels) use VGG16, which makes them not directly comparable to the others, as discussed in the caption of Table 3.5.

connected layers, while fitting very well into the deep learning philosophy of introducing transformations executed in steps as the signal is propagated down the network rather than projecting to higher-dimensional spaces.

It should be noted that the method of pruning proposed in [Han et al., 2015b] and the follow-up method of pruning, quantising and Huffman coding proposed in [Han et al., 2015a] achieve compression rates between x9 and x27 on AlexNet* by applying a pipeline of reducing operations on trained models. Usually, it is necessary to perform at least a few iterations of such reductions to arrive at the stated compression rates. For the AlexNet model one iteration takes 173 hours according to [Han et al., 2015b]. On top of that as this method requires training the original full model the time cost of that operation should be taken into consideration as well.

Compressing pipelines target models that are ready for deployment and function in the environment where the time spent on training is absolutely dominated by the time spent evaluating predictions. In contrast, SELL methods are appropriate for incorporation into the design of a model.

* [Han et al., 2015a] report x35 compression by using Huffman coding and counting bytes. I report the number of parameters for consistency.

3.10.8 Conclusion

This work introduces new Structured Efficient Linear Layer, which adds to a growing literature on using memory-efficient structured transforms as efficient replacements for the dense matrices in the fully-connected layers of neural networks. The structure of SELL is motivated by matrix approximation results from Fourier optics, but has been specialised for efficient implementation on NVIDIA GPUs.

It was shown that a proper initialisation of SELL allows building very deep cascades of SELLs that can be optimised using SGD. The proper initialisation is simple but is essential for training cascades of SELLs with more than a few layers. Working with deep and narrow cascades of SELLs makes networks more parameter efficient than previous works using shallow and wide cascades because the cost of layers interfacing between the SELL and the rest of the network is reduced (e.g. the size of the input to the dense logistic regression layer of the network is much smaller).

In future work I plan to investigate replacing the diagonal layers of ACDC with other efficient structured matrices such as band or block diagonals. These alternatives introduce additional parameters in each layer, but may give us the opportunity to explore the continuum between depth and expressive power per layer more precisely.

Another exciting avenue for investigation is to include SELL layers in other neural network models such as RNNs or LSTMs. Recurrent nets are particularly attractive targets as they are typically composed entirely of linear layers. The potential parameter savings are substantial. Moreover since the computational bottleneck in these models comes from matrix-matrix multiplications, there is a potential for speed improvement as well.

Chapter 4

Improving neural networks by noise injection

4.1 On the role of noise in SGD optimisation

A core aspect of deep learning is the concept of noise. There exist many possible ways a generated noise can be added to different components of a model during training. Although the idea of using noise has been mentioned in the literature [An, 1996], [Bottou, 1991a], it seems that this direction of research is relatively underexplored.

4.1.1 Supervised learning

The result that we hope for when using noise with supervised learning is an increased regularisation and enhanced exploration. Typically adding noise will produce a gradient different from the usual stochastic gradient. It will encourage more exploration during optimisation. When and how noise is injected can take many variants.

4.2 Related work

4.2.1 Adding noise directly to the gradient

One of the well-known papers on the subject is [Neelakantan et al., 2015b], where a zero-mean Gaussian noise is added directly to the stochastic gradient. Parameters of the noise distribution are not learned, but the variance is annealed as the training progresses.
The paper contains an extensive empirical study of the impact of the method on deep fully-connected networks with ReLUs [Nair and Hinton, 2010a], End-To-End Memory Networks [Sukhbaatar et al., 2015] which are related to Neural Turing Machine [Graves et al., 2015], Neural Programmer [Neelakantan et al., 2015a] similar to Neural Programmer-Interpreter

[Reed and de Freitas, 2015], Neural RAM [Kurach et al., 2015] and Neural GPU [Kaiser and Sutskever, 2015]. It improves generalisation and makes it easier to train complicated neural models.

4.3 Noisy Activation Functions

4.3.1 Introduction

The introduction of the piecewise-linear activation functions such as ReLU and Maxout [Goodfellow et al., 2013] units had a profound effect on deep learning and was a major catalyst allowing training much deeper networks. It was shown [Glorot et al., 2011] that using ReLU allows training deeper models as compared to tanh nonlinearity. A plausible hypothesis about the recent surge of interest in these piecewise-linear activation functions [Glorot et al., 2011] is that they are easier to optimise with SGD and backpropagation than their smooth counterparts, such as sigmoid and tanh. The recent successes of piecewise-linear functions are particularly evident in computer vision, where ReLU has become the default choice in convolutional networks.

In this thesis I propose a new technique to train neural networks with activation functions which saturate when the absolute value of an input is large. This is achieved by injecting a learnable noise to the activation function when operating in its saturated regime. This approach allows training neural networks with a broader family of activation functions.

Adding noise to the activation function has been considered for ReLU units and was explored in [Bengio et al., 2013, Nair and Hinton, 2010b] for feed-forward networks and Boltzmann machines to encourage units to explore more and make the optimisation easier. More recently there has been a significant increase in popularity of more elaborated "gated" architectures such as LSTMs [Hochreiter and Schmidhuber, 1997b] and GRUs [Cho et al., 2014]. These designs are often equipped with neural attention mechanisms that have been used in the NTM [Graves et al., 2014], Memory Networks [Weston et al., 2014], automatic image captioning [Xu et al., 2015b], video caption generation [Yao et al., 2015] and wide areas of applications [LeCun et al., 2015]. A common thread running through these works is the use of soft-saturating nonlinearities, such as the sigmoid or the softmax to emulate hard decisions of digital logic circuits. Despite its success, there are two key problems with this approach.

1. Since the nonlinearities still saturate there are problems with backpropagating gradient information flowing through the gates.

2. Since the nonlinearities only softly saturate they do not allow to take hard decisions.

Although the gates often operate in the soft-saturated regime [Karpathy et al., 2015, Bahdanau et al., 2014, Hermann et al., 2015] the architecture prevents them from being fully open or closed.

This work follows a novel approach to address these problems. The method eliminates the second problem with the use of hard-saturating nonlinearities, which allow gates to make clear "on" or "off" decisions when they saturate. Since the gates can be completely open or closed, no information is lost through the leakiness of the soft-gating architecture.

Unfortunately, by introducing hard-saturating nonlinearities the vanishing gradient problem is exacerbated since gradients in the saturated regime are now precisely zero instead of being negligible. In such situation the method encourages random exploration by introducing noise into the activation function. The amount of noise can change based on the magnitude of saturation.

At test time the noise in the activation function can be removed or replaced with its expected value. The experiments show that resulting deterministic networks outperform their soft-saturating counterparts on a variety of tasks. A very good performance is achieved by a straightforward drop-in replacement of the nonlinearities in existing codebase.

The proposed idea addresses the difficulty of optimisation of softly saturating units common in gating architectures. Moreover, I describe a way of performing simulated annealing for neural networks.

While [Hannun et al., 2014, Le et al., 2015] use ReLU activation functions with simple RNNs, it turns out that with noise injection it is possible to use piecewise-linear activation functions with well-performing gated recurrent networks such as LSTM or GRU.

4.3.2 Saturating activation functions

Definition 4.3.1. (Activation Function). An activation function is a function $h : \mathcal{R} \to \mathcal{R}$ that is differentiable almost everywhere.

Definition 4.3.2. (Hard Saturation). An activation function $h(x)$ with the derivative $h'(x)$ is said to hard saturate if there exists $c_1 \geq 0$ such that for $x > c_1$ $h'(x) = 0$ and there exists $c_2 \leq 0$ such that for $x < c_2$ $h'(x) = 0$.

Definition 4.3.3. (Soft Saturation). An activation function $h(x)$ with the derivative $h'(x)$ is said to soft saturate if it does not hard saturate and $\lim_{x \to +\infty} h'(x) = 0$ and $\lim_{x \to -\infty} h'(x) = 0$.

Most common activation functions used in recurrent networks soft saturate, for example tanh and sigmoid. It is possible to construct hard saturating versions of common soft

saturating activation functions by taking the first-order Taylor expansion around some point and clipping the resulting function to an appropriate range. For instance, expanding tanh and sigmoid around $x_0 = 0$ produces functions u^t and u^s which are linearised versions of respectively tanh and sigmoid

$$\tanh(x) \approx \mathrm{u}^t(x) = x \tag{4.1}$$

$$\text{sigmoid}(x) \approx \mathrm{u}^s(x) = 0.25x + 0.5 \tag{4.2}$$

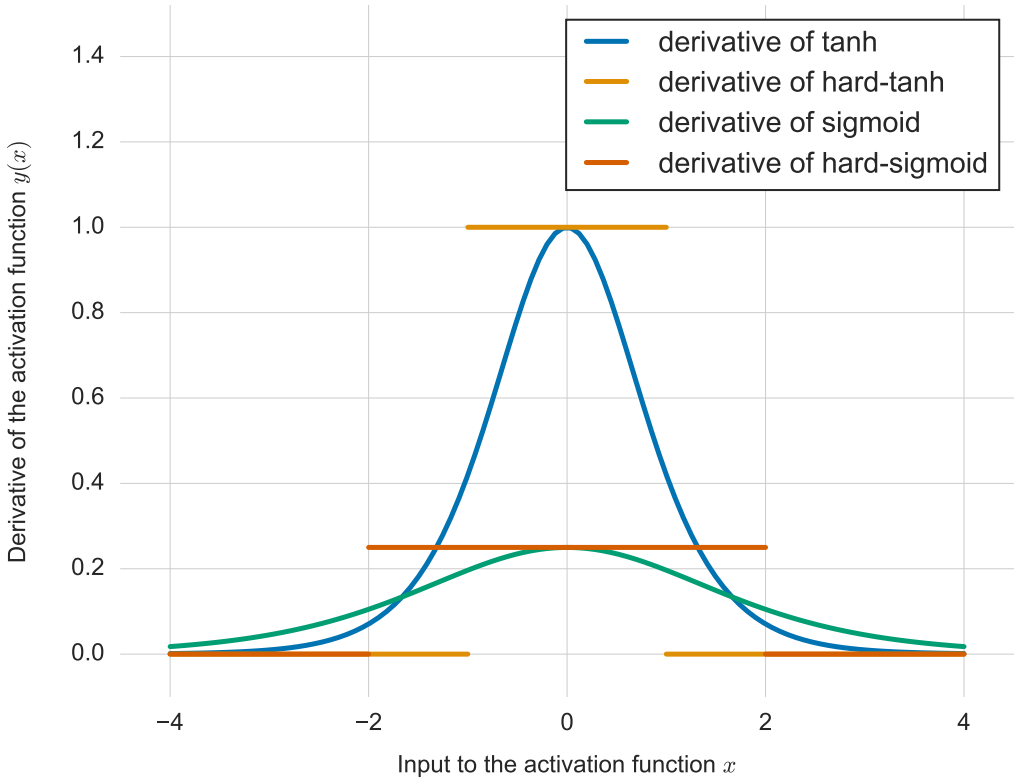
Clipping the linear approximations gives

$$\text{hard-tanh}((x) = \max(\min(\mathrm{u}^t(x),\ 1),\ -1). \tag{4.3}$$

$$\text{hard-sigmoid}(x) = \max(\min(\mathrm{u}^s(x),\ 1),\ 0) \tag{4.4}$$

The motivation behind this construction is to introduce a linear behaviour around zero to allow gradients to flow easily when the unit is not saturated while providing a firm decision in the saturated regime.

Figure 4.1: A plot of derivatives of different activation functions.



The ability of the hard-sigmoid and hard-tanh to hard saturate comes at the cost of exactly zero gradients in the saturated state. This can cause difficulties during training as described above. The derivative is equal to zero so an infinitesimal change to x would not have an impact on the output of the activation function. However, an actual perturbation of x (for example returning x to nonlinearity's linear regime) may be desired in order to minimise the objective function. Unfortunately, the derivative does not reflect that.

For the rest of this thesis I use $h(x)$ to refer to a hard saturating activation function such as the hard-sigmoid and the hard- tanh. I use $u(x)$ to denote its linearisation based on the first-order Taylor expansion around zero.

hard-sigmoid saturates when $x \leqslant -2$ or $x \geqslant 2$ and hard-tanh saturates when $x \leqslant -1$ or $x \geqslant 1$. The absolute value of the threshold will be denoted by x_t. For hard-sigmoid $x_t = 2$ and for hard-tanh $x_t = 1$.

Remark. hard-sigmoid(x), sigmoid(x) and tanh(x) are all contractive mappings. hard-tanh(x), becomes a contractive mapping when the absolute value of the input is greater than the threshold. An important difference between these activation functions is their fixed-points. hard-sigmoid(x) has a fixed-point at $x = \frac{2}{3}$. The fixed-point of sigmoid(x) is $x \approx 0.69$. Any $x \in \mathcal{R}$ between -1 and 1 can be a fixed-point of hard-tanh(x), but the fixed-point of tanh(x) is 0. tanh(x) and sigmoid(x) have attractors at their fixed-points. Those mathematical differences can make activation functions behave differently when used with recurrent neural networks.

When using classical activation functions like tanh or sigmoid, a highly complicated and noisy trajectory of the stochastic gradient descent optimisation may bring the parameters into a state where many units are in a zero-gradient regime for a significant subset of training examples. It may be hard to escape such situation and the units may get "stuck". When units saturate and gradients vanish, the optimisation algorithm usually requires many iterations and a considerable amount of computation to recover.

4.3.3 Annealing with Noisy Activation Functions

Before focusing on the proposed noisy activation functions that have been used in the experiments, consider a general noisy activation function $\phi(x, \xi)$ with injected i.i.d. noise ξ. Allow it to replace the saturating nonlinearities such as hard-sigmoid and hard-tanh introduced in the previous section. Assume the whole model is trained with a variant of stochastic gradient descent (SGD).

Let ξ have variance σ^2 and $\mu = 0$. The goal is to characterise what happens when the noise is gradually annealed, going from large noise levels ($\sigma \to \infty$) to no noise at all

$(\sigma \to 0)$. Furthermore, let us assume that ϕ is a function such that when the noise level becomes large, so does the absolute value of its derivative with respect to x

$$\lim_{|\xi|\to\infty} |\frac{\partial \phi(x,\xi)}{\partial x}| \to \infty. \tag{4.5}$$

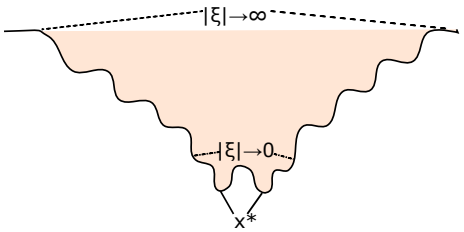


Figure 4.2: An example of a one-dimensional nonconvex objective function where a simple stochastic gradient descent performs poorly. However, with a large noise ($|\xi| \to \infty$) SGD can escape from saddle points and bad local minima as a result of exploration. As the noise level is annealed ($|\xi| \to 0$), SGD will eventually converge to one of the local minima x^*.

When there is no noise ($\sigma = 0$, $\mu = 0$) a deterministic nonlinearity $\phi(x, 0)$ is recovered, which in the experiments is piecewise-linear. As illustrated in Figure 4.2 when σ is large (the level of noise is high) backpropagating through ϕ gives rise to large derivatives. Hence the noise drowns the signal. The example-wise gradient with respect to parameters is much larger than it would have been with $\sigma = 0$. Therefore the gradient is very noisy and randomly moves around in the parameter space without being attracted to any trend.

Annealing is also related to the signal to noise ratio which can be defined as $SNR = \frac{\sigma_{\text{signal}}}{\sigma_{\text{noise}}}$. When $SNR \approx 0$ the model performs a purely random exploration. As the σ_{noise} is annealed the SNR will increase and when σ_{noise} converges to 0 the only source of exploration during training is the standard noise from the Monte Carlo estimation of the gradient.

It is precisely what makes methods such as simulated annealing [Kirkpatrick et al., 1983] and continuation methods [Allgower and Georg, 1980] helpful when optimising a difficult nonconvex objective. When the level of noise is high, SGD is free to explore all parts of the space. As the noise level is decreased, it will prefer regions where the signal is strong enough to be "visible". When the noise level is further reduced SGD spends more time in "globally better" subset of the parameter space. As the noise approaches zero, the algorithm is fine-tuning the solution and converges to a local minimum of the noise-free objective function. A related approach of adding noise to gradients and annealing it was investigated in [Neelakantan et al., 2015c]. [Ge et al., 2015] showed that SGD with annealed noise would globally converge to a local-minima for nonconvex objective functions

in the polynomial number of iterations. Recently [Mobahi, 2016] proposed an optimisation method that applies Gaussian smoothing on the loss function, where the weight noise is annealed.

4.3.4 Adding noise when units saturate

A novel idea behind the proposed noisy activation is that the amount of noise added to the nonlinearity can be related to the magnitude of the saturation of the function. Effectively for hard-sigmoid(x) and hard-tanh(x) the noise is only added when they saturate. This is different from previous proposals such as the noisy rectifier from [Bengio, 2013], where noise is added just before a rectifier unit (ReLU) independently of whether the input is in the linear regime or in the saturating regime of the nonlinearity.

The motivation is to keep the training signal clean when the unit is in the non-saturating regime and provide a noisy signal otherwise.

Consider noisy activation function of the following form

$$\phi(x, \xi) = \begin{cases} \text{h}(x) & \text{if h(x) is not saturated} \\ \text{h}(x) + s & \text{if h(x) is saturated} \end{cases} \tag{4.6}$$

where

$$s = \mu + \sigma\xi \tag{4.7}$$

h(x) refers to a hard saturating activation function such as the hard-sigmoid and the hard-tanh introduced in Sec. 4.3.2. ξ is an i.i.d. random variable drawn from some generating distribution and μ and σ are adjustable parameters.

Intuitively, when a unit saturates, its output is pinned and the noise is added. The exact behaviour of the method depends on the type of noise ξ and the choice of μ and σ which can be parameterised as functions of x allowing gradients to be propagated even when $h(x)$ is in the saturated regime.

It is desirable that $\phi(x, \xi)$ does not systematically take values distant from the $h(x)$

$$\mathbb{E}_\xi[\phi(x, \xi)] \approx h(x) \tag{4.8}$$

If the distribution of ξ has zero mean then this property can be satisfied by setting $\mu = 0$. However, in other cases it is necessary to make a different choice for μ. In practice a slightly biased $\phi(x, \xi)$ gave best results.

Intuitively when x is far into the saturated regime larger changes to parameters are required to desaturate $h(x)$. In such conditions it is reasonable to add proportionally more

noise. Conversely, when x is close to the saturation threshold a small change to parameters would be sufficient to escape back to linearity. Therefore it seems natural to provide a measure of saturation to the function computing the amount of applied noise. To that end I make use of the difference between the original activation function $h(x)$ (piecewise-linear) and its linearisation $u(x)$ when computing the parameters of the noise distribution

$$\Delta(x) = \mathrm{h}(x) - \mathrm{u}(x) \tag{4.9}$$

See Equations 4.1 for definitions of $\mathrm{u}(x)$ for the hard-sigmoid and hard-tanh respectively. Figure 4.3 is an illustration of $\Delta(x)$. $|\Delta(x)|$ is referred to as the magnitude of saturation. Experimental exploration revealed that the following relationship between σ and $\Delta(x)$ works best

$$\sigma(x) = c\,(\mathrm{sigmoid}(p\Delta(x)) - 0.5)^2 \tag{4.10}$$

Note that p is a learnable parameter and c is a hyperparameter. By changing p the model can adjust the variance of the injected noise to the problem. Moreover, p also determines the direction of the relationship between the variance and $\Delta(x)$ which can be positive or negative. The final scale of the variance is controlled by the hyperparameter c.

4.3.4.1 Derivatives in the saturated regime

In the simplest case ξ is drawn from a zero-mean distribution ($\mu = 0$), for example a standard normal. In such situation the Equation 4.8 is perfectly satisfied

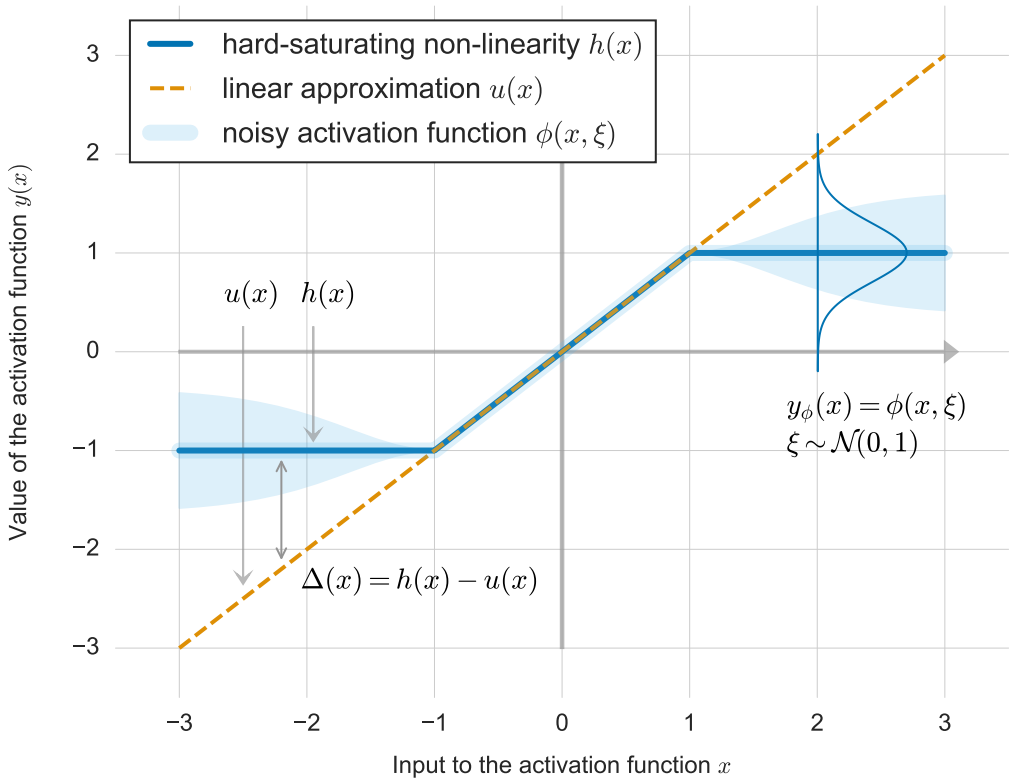
$$\mathbb{E}_\xi[\phi(x, \xi)] = h(x)$$

Due to the parameterisation of $\sigma(x)$, if x is in a non-saturating part of the domain of $h(x)$, then $\Delta(x)$ is zero and consequently $\sigma(x)$ is zero. The activation function behaves exactly as the linear function $\mathrm{u}(x)$. On the other hand, if $h(x)$ is saturated the derivative $h'(x)$ equals zero, which in a traditional setting would mean zero gradient. However, because $\sigma(x)$ depends on x the gradient flows through $\sigma(x)$ even if $h'(x)$ equals zero

$$\phi'(x, \xi) = \frac{\partial}{\partial x}\phi(x, \xi) = \sigma'(x)\xi \tag{4.11}$$

In the non-saturated regime $\phi'(x, \xi) = \mathrm{h}'(x)$ so the optimisation can exploit the linear structure of $h(x)$ near the origin. In the saturated regime the randomness of ξ drives exploration and the gradient still flows through the network since the variance of the noise depends on x. Therefore a non-zero gradient information is available at almost every point despite the saturation of $h(x)$. In the saturated regime the variance of the gradient depends on the variance of $\sigma'(x)\xi$.

Figure 4.3: A depiction of a zero-mean Gaussian noise added to the hard-saturating non-linearity $h(x)$. A linear approximation of $h(x)$ at $x = 0$ is denoted by $u(x)$. The difference $h(x) - u(x)$ is represented by $\Delta(x)$ and indicates the discrepancy between the linear approximation and the actual function. Note that $\Delta(x) = 0$ at non-saturating parts of the domain where $u(x)$ and $h(x)$ match perfectly. $\phi(x, \xi)$ is the noisy activation.



4.3.4.2 Leaky formulation

Consider also a leaky version of the activation function presented above. The motivation is to further improve the flow of gradients by introducing a small slope to the h(x) in a saturated regime

$$\phi(x,\xi) = \begin{cases} \mathrm{h}(x) & \text{if h(x) is not saturated} \\ \alpha\mathrm{h}(x) + (1-\alpha)\mathrm{u}(x) + s & \text{if h(x) is saturated} \end{cases} \tag{4.12}$$

$(1-\alpha)$ should be small, e.g. $\alpha = 0.9$. Although adding the slope improves the signal in the backward pass, it deviates the value of $\phi(x,\xi)$ from $\mathrm{h}(x)$. In other words the Equation 4.8 is no longer satisfied. It is possible to use a directed noise to compensate for that

$$s = d(x)\sigma(x)|\xi|, \text{ where} \tag{4.13}$$

$$d(x) = -\operatorname{sgn}(x)\operatorname{sgn}(1-\alpha) \tag{4.14}$$

$\operatorname{sgn}(x)$ is the sign function

$$\operatorname{sgn}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

If ξ is distributed according to a normal distribution then $|\xi|$ is distributed according to a half-normal distribution. The sign of $d(x)\sigma(x)|\xi|$ is determined by $d(x)$ because other terms are always positive. Intuitively the noise "pushes" the activations back towards $\mathrm{h}(x)$. It is important to keep the value of the hyperparameter α close to 1, so the added slope is not too steep as illustrated in Fig. 4.4.

Eqn 4.15 and Eqn 4.16 provide the formulation of the activation function where $\epsilon = |\xi|$ if the noise is sampled from a half-normal distribution and $\epsilon = \xi$ if the noise is sampled from a normal distribution. The relationship between $\Delta(x)$, $\mathrm{u}(x)$ and $\mathrm{h}(x)$ can be expressed in the following way

$$\phi(x,\epsilon) = \alpha\mathrm{h}(x) + (1-\alpha)\mathrm{u}(x) + \mathrm{d}(x)\sigma(x)\epsilon \tag{4.15}$$

Or equivalently

$$\phi(x,\epsilon) = \mathrm{u}(x) + \alpha\Delta(x) + \mathrm{d}(x)\sigma(x)\epsilon \tag{4.16}$$

Figure 4.3 illustrates the case of a normally distributed noise.

After experimenting with different types of noise distribution, it turned out that in terms of performance half-normal and normal noise tend to work very well.

Eqn 4.15 provides a good illustration of the three paths that gradients can flow through during optimisation: the linear path $\mathrm{u}(x)$, the nonlinear path $\mathrm{h}(x)$ and the stochastic path

$\sigma(x)$. The flow of gradients through these different pathways across different layers makes the optimisation of a noisy activation function easier as compared to saturating nonlinearities.

At test time it is reasonable to use the expected value of $\phi(x, \xi)$ with respect to the injected noise in order to get a deterministic behaviour

$$\mathbb{E}_\xi[\phi(x, \xi)] = \alpha \mathrm{h}(x) + (1 - \alpha)\mathrm{u}(x) + \mathrm{d}(x)\sigma(x)\mathbb{E}_\xi[\epsilon] \tag{4.17}$$

If $\epsilon = \xi$ then $\mathbb{E}_\xi[\epsilon] = 0$. Otherwise if $\epsilon = |\xi|$ then $\mathbb{E}_\xi[\epsilon] = \sqrt{\frac{2}{\pi}}$.

Algorithm 1 Noisy Activations with half-normal noise for hard-saturating functions

1: $\Delta \leftarrow h(x) - u(x)$
2: $\mathrm{d}(x) \leftarrow -\operatorname{sgn}(x)\operatorname{sgn}(1 - \alpha)$
3: $\sigma(x) \leftarrow c\,(g(p\Delta(x)) - 0.5)^2$
4: $\xi \sim \mathcal{N}(0,\ 1)$
5: $\phi(x, \xi) \leftarrow \alpha \mathrm{h}(x) + (1 - \alpha)\mathrm{u}(x) + \mathrm{d}(x)\sigma(x)|\xi|$

Fig 4.4 presents the impact of α on a noisy activation function based on hard-tanh.

4.3.5 Adding noise to the input of the function

Adding noise with a fixed standard deviation to the input of an activation function has been investigated for ReLU nonlinearities in [Nair and Hinton, 2010b, Bengio et al., 2013]. That work is closely related to the idea described here, so it seems natural to formulate it and include it in the experiments for comparison and completeness.

$$\phi(x, \xi) = \mathrm{h}(x + \sigma\xi) \text{ and } \xi \sim \mathcal{N}(0, 1) \tag{4.18}$$

Eqn 4.18 provides a parameterisation of the activation function with noise injected at the input. Please note that in a broader context σ can be either learned as in Eqn 4.10 or fixed as a hyperparameter.

Experimental verification has revealed that small values of σ tend to work better. This can be counterintuitive because as x gets larger and moves further away from the threshold x_t, it becomes less likely that the injected noise can push the activations back to the linear regime. Nevertheless, a smaller σ means less noisy gradients, which can be important if the activation function has the noise injected in the whole domain due to the absence of the distinction between non-saturated and saturated intervals in Eqn 4.18.

Therefore an alternative version of the above function is introduced with noise injected only when there is saturation

$$\phi(x, \xi) = \mathrm{h}(x + \mathbf{1}_{|x| \geq |x_t|}(\sigma\xi)) \text{ and } \xi \sim \mathcal{N}(0, 1) \tag{4.19}$$

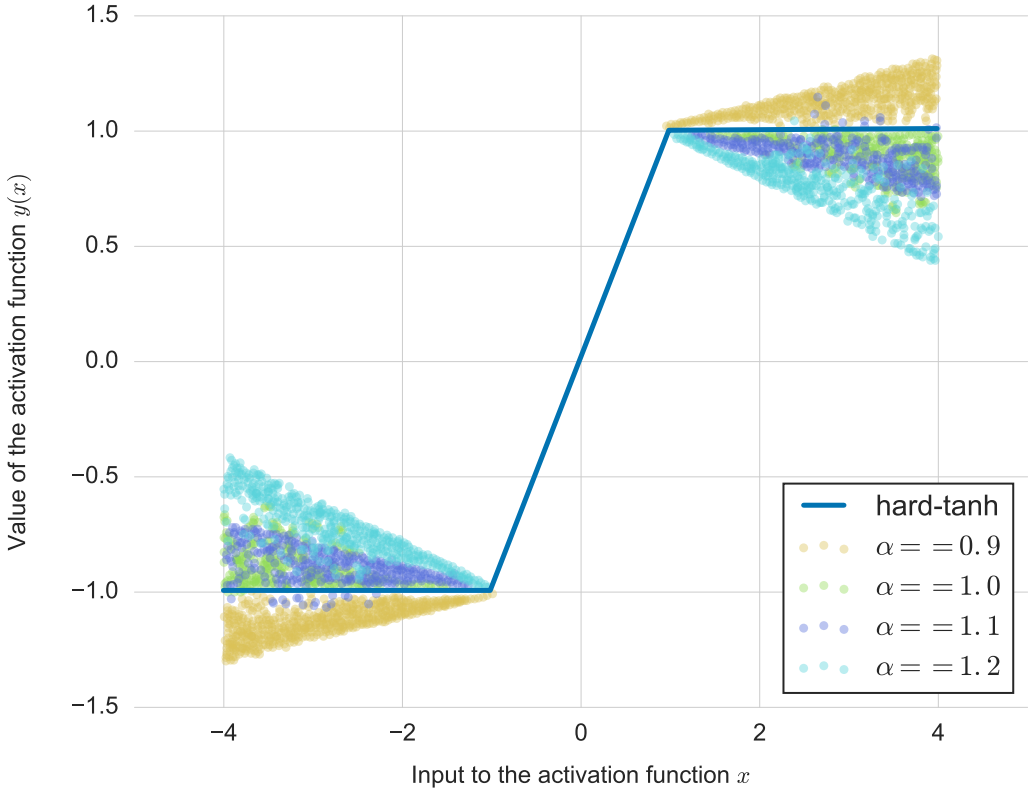


Figure 4.4: Stochastic behaviour of the proposed noisy activation function approximating hard-tanh nonlinearity. Different colours represent different values of α. The noise is sampled from a half-normal distribution.

4.3.6 Experimental results

In the experiments the noise is injected only during training. At test time it is replaced with its expected value. Please note that the experiments involve only a drop-in replacement of the activation functions in existing codebases without changing the original hyperparameters. Hence, it is plausible that one could obtain better results by performing a careful hyperparameter tuning for the models with noisy activation functions. In all experiments p is initialised randomly using the uniform distribution $[-1,\ 1]$.

The following abbreviations correspond to different types of the experimental setup:

- **(NAN)** normal noise injected at the output
- **(NAH)** half-normal noise injected at the output
- **(NANI)** normal noise injected at the input of the function with fixed σ

- **(NANIL)** normal noise injected at the input of the function with learned σ
- **(NANIS)** normal noise injected at the input of the function when the unit saturates

4.3.6.1 Exploratory analysis

As the first step let us focus on small-scale control experiments. Fig 4.5 shows the learning curves of different types of activation functions with various types of injected noise contrasted with the tanh and the hard-tanh units. The models are single-layer MLPs trained on MNIST for classification and the plot presents the average negative log-likelihood. The results suggest that models with noisy activations converge faster than those using tanh and hard-tanh functions. The final NLL also seems to be lower when compared to tanh and hard-tanh.

As the next step consider a dataset generated from a mixture of three Gaussian distributions with different means and standard deviations. The problem is approached with a three-layer MLP. Each layer of the MLP consists of 8 hidden units. The model with the tanh and the model with the noisy tanh activation functions were able to solve this task almost perfectly. Figure 4.6 and Figure 4.7 show the scatter plot of the activations of each unit at each layer and the derivative function of each unit at each layer with respect to its input when using the learned values of the parameter p.

The next experiment is an investigation of the performance of a neural network equipped with NAN, NANI and NANIS activation functions on character-level language modelling using Penn Treebank (PTB) dataset. A GRU language model is trained on sequences of length 200. The same model is used with all the activation functions. The hyperparameters mostly remain untouched except a small grid search for σ for NANI and NANIS in the interval $[0.01, 1]$ with eight possible values. The best σ was chosen using validation based on bit-per-character (BPC). There does not seem to be an important difference between NAN and NANI regarding training performance as suggested in Figure 4.8.

4.3.6.2 Learning to Execute

The problem of predicting the output of a short program introduced in [Zaremba and Sutskever, 2014b]* proved challenging for modern deep learning architectures. The authors turn to curriculum learning [Bengio et al., 2009a] to let the model capture knowledge

* The code resides at `https://github.com/wojciechz/learning_to_execute`. I am very thankful to the authors for making the code publicly available.

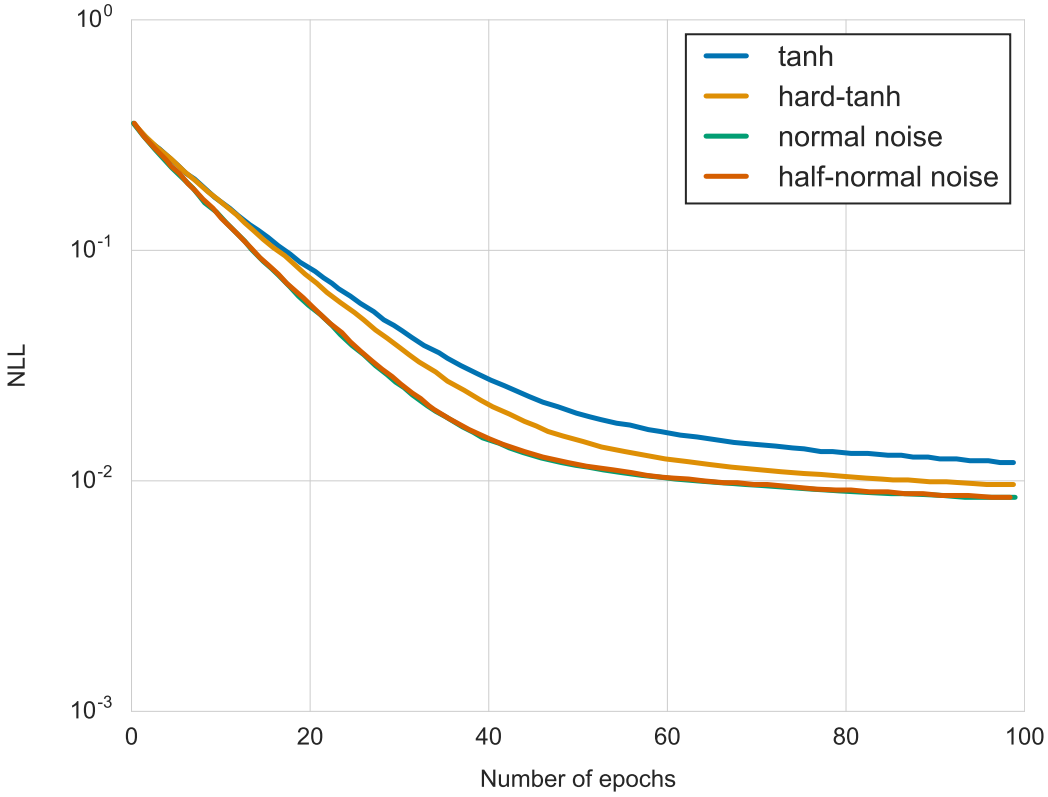


Figure 4.5: Training loss curves of a single layer MLP optimised with RMSProp using different activation functions with different types of noise.

about the easier examples first. Then the level of difficulty is further increased during training. It is known that there exist tasks that impose extremely challenging optimisation issues unless they are approached with a curriculum method.

The idea is to try noisy activation functions expecting an improvement from an adaptation of the parameters of the noise to the obstacles in optimisation. In this experiment all sigmoid and tanh nonlinearities in the reference model were replaced with their noisy counterparts. The only other change is the modification of the default gradient clipping from 10 to 5 to avoid problems with numerical stability. When evaluating the network the length (number of lines) of the executed programs was set to 6 and the nesting was set to 3 which are the default settings for these tasks in the released code. Both the reference model and the model with noisy activations were trained with "combined" curriculum which is the most sophisticated and the best performing one.

The results presented in Table 4.1 and in Figure 4.10 show that applying the proposed activation functions leads to better performance than the reference model. Moreover, the

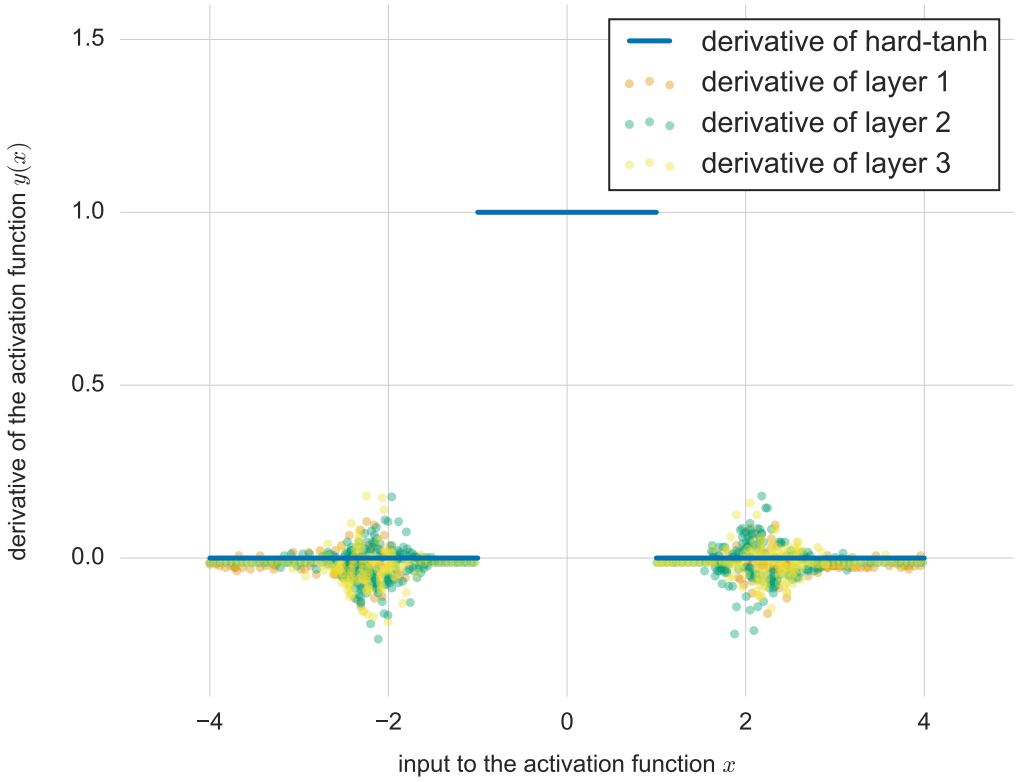


Figure 4.6: Derivatives of noisy activations layers with respect to their inputs in an MLP with three layers. The network was trained on a dataset generated by three normal distributions with different means and standard deviations. $\alpha = 1$ (no additional slope).

method is easy to combine with a non-trivial curriculum learning.

4.3.6.3 Word-level LSTM on Penn Treebank

A 2-layer word-level LSTM language model proposed by [Zaremba et al., 2014]* was trained on Penn Treebank. The only architectural change is a replacement of all sigmoid and tanh units with noisy hard-sigmoid and noisy hard-tanh units. Experiments with noisy activations were conducted in a setting almost identical to the default one but with a changed gradient clipping threshold from 10 to 5. Please note that the reference model in [Zaremba et al., 2014] is a well-tuned, strong baseline.

The results are contained in Table 4.2. There is a clear improvement due to noise injection. In this case both normal and half-normal noise distributions seem to give very similar results.

* The code is available at `https://github.com/wojzaremba/lstm`

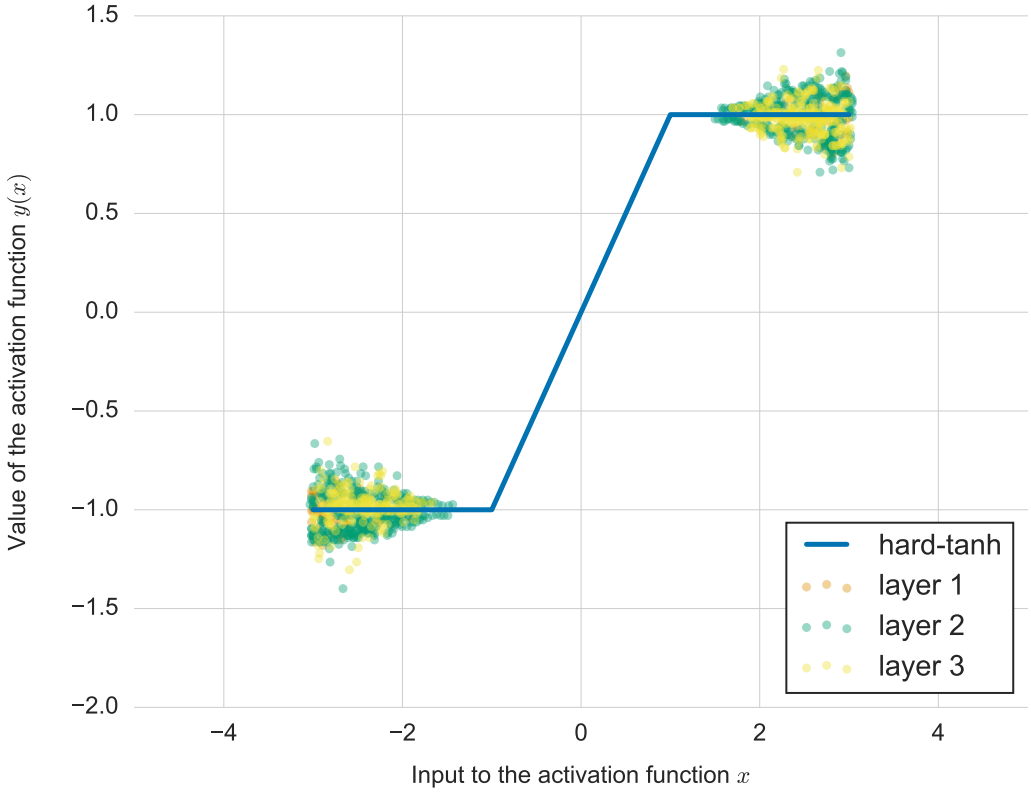


Figure 4.7: Noisy activations in an MLP with three layers. The network was trained on a dataset generated by three normal distributions with different means and standard deviations. $\alpha = 1$ (no additional slope).

4.3.6.4 Neural Machine Translation experiments

This experiment involves training a neural machine translation (NMT) model with the neural attention [Bahdanau et al., 2014]* on the Europarl dataset. Again all sigmoid and tanh units were swapped with their noisy formulations. Weight matrices initialised for orthogonality were scaled down by 0.01. All models were trained with early-stopping. The evaluation was done on the `newstest2011` test set.

Table 4.4 illustrates the results including a comparison with a model with hard-tanh and hard-sigmoid units. Consistently with previous findings, one observes a substantial improvement (more than two BLEU points) with respect to the reference model for English to French machine translation. Noisy activations outperformed both baselines.

* The code is available at `https://github.com/kyunghyuncho/dl4mt-material`

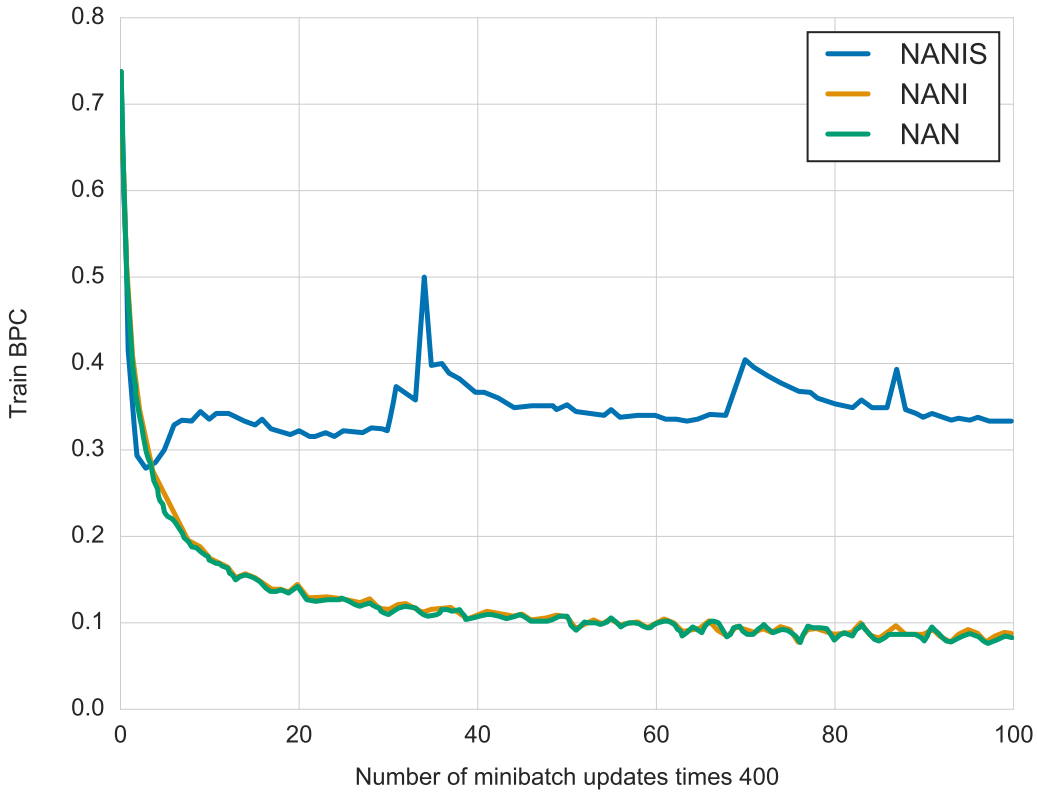


Figure 4.8: The learning curves of a simple character-level GRU language model on sequences of length 200 on PTB dataset. NANI and NAN have very similar learning curves.

Table 4.1: Performance of the noisy network on Learning to Execute [Zaremba and Sutskever, 2014b] task. Just changing the activation function to its noisy variant yields about 2.5% improvement in accuracy.

Model	Test accuracy
Reference model	46.45%
Noisy network (**NAH**)	**48.09%**

Table 4.2: Word-level Penn Treebank comparative perplexities. The reference model is described in [Zaremba et al., 2014]. A drop-in replacement of classical activation functions, like sigmoid and tanh, with their corresponding noisy variants leads to a substantial improvement in perplexity.

	Validation perplexity	Test perplexity
Noisy LSTM + **NAN**	111.7	**108.0**
Noisy LSTM + **NAH**	112.6	**108.7**
LSTM (reference model)	119.4	115.6

4.3.6.5 Image Caption Generation experiments

Caption generation is a problem of generating a textual description of natural images. A reference network is the publicly available soft neural attention model proposed in [Xu et al., 2015b]*. Weight matrices with orthogonal initialisation were scaled by 0.01. All generators were trained on the Flickr8k dataset.

As shown in Table 4.3 noisy activations functions obtain better results than the reference model in terms of NLL, higher-order BLEU score and the METEOR score. Especially in terms of the METEOR score they significantly outperform the best network found in [Xu

* The code is available at `https://github.com/kelvinxu/arctic-captions`

Table 4.3: Image caption generation on Flickr8k. The noisy activations are added to the code from [Xu et al., 2015b]. It results in a substantial improvements in terms of NLL, higher-order BLEU score and METEOR score. Soft attention and hard attention refer to using backpropagation or REINFORCE algorithm when training the attention mechanism. $\sigma = 0.05$ for NANI and $c = 0.5$ for both NAN and NANIL.

Model	BLEU -1	BLEU-2	BLEU-3	BLEU-4	METEOR	Test NLL
Soft attention (sigmoid and tanh) (reference model)	**67**	44.8	29.9	19.5	18.9	40.33
Soft attention (**NAH sigmoid & tanh**)	66	**45.8**	**30.69**	**20.9**	**20.5**	40.17
Soft attention (**NAH sigmoid & tanh** wo dropout)	64.9	44.2	**30.7**	**20.9**	20.3	**39.8**
Soft attention (**NANI sigmoid & tanh**)	66	45.0	30.6	20.7	**20.5**	40.0
Soft attention (**NANIL sigmoid & tanh**)	66	44.6	30.1	20.0	**20.5**	39.9
Hard attention (sigmoid and tanh)	67	45.7	31.4	21.3	19.5	-

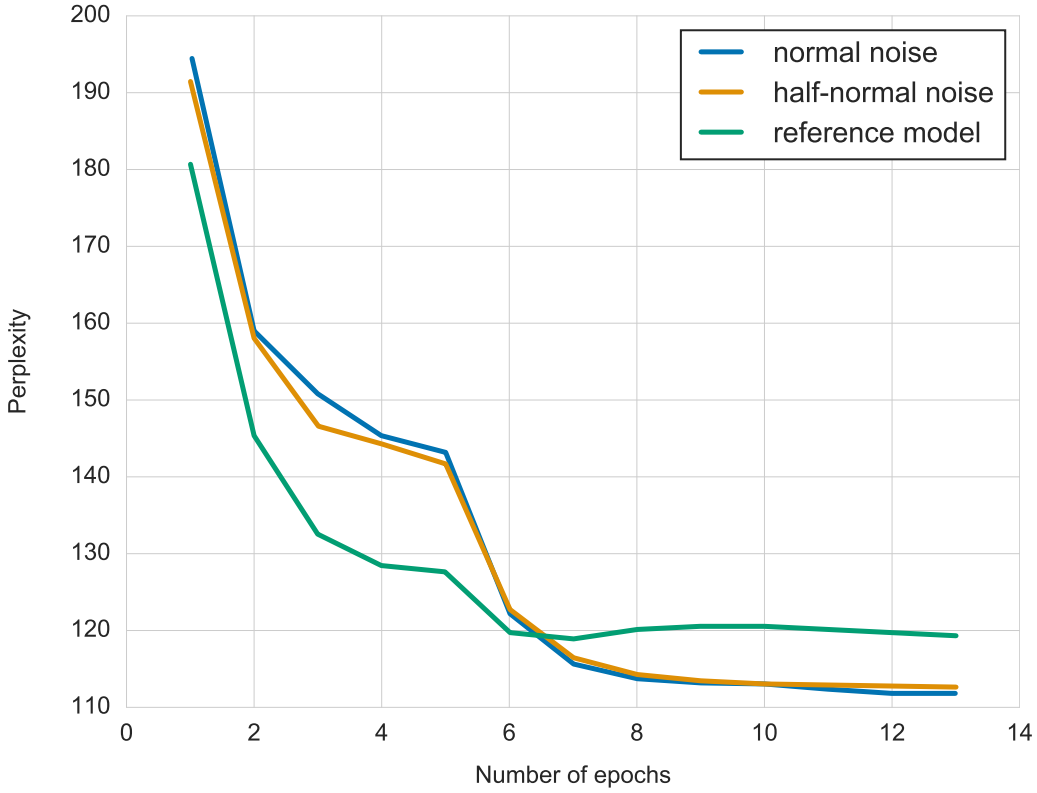


Figure 4.9: Validation perplexity curves for the word-level LSTM language model on Penn Treebank dataset.

et al., 2015b].

The reference model uses dropout with probability 0.5 on the output of the LSTM layers and on the context. It is interesting to try noisy activation functions with and without dropout on this problem. Table 4.3 illustrates that results have improved further with the addition of dropout. It suggests that the two techniques can be combined for better performance.

4.3.6.6 Experiments with noise annealing

One captivating question is about the impact of noise annealing on the training of neural networks. It is closely related to the idea of continuation methods where the optimisation takes place over a number of related loss functions ordered by the amount of noise injected during the procedure.

For that purpose a new special task is designed. Given a sequence of random integers, the challenge is to output the number of unique elements in the sequence. Specifically, the

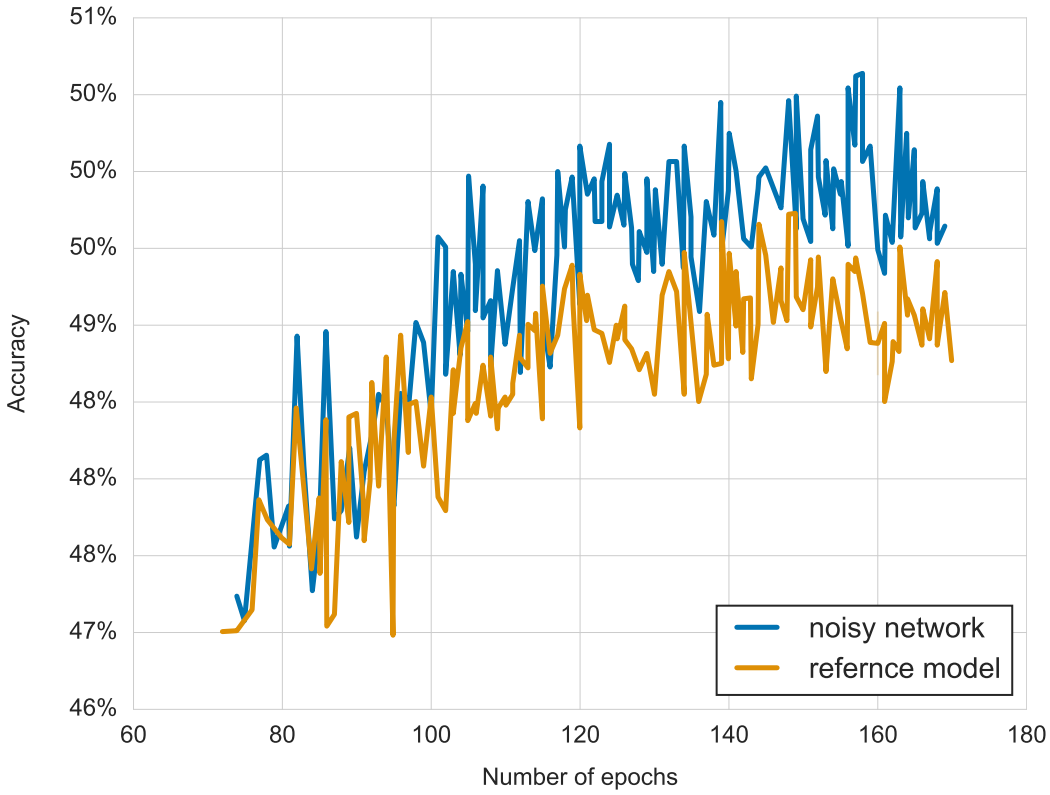


Figure 4.10: Training curves of the reference model [Zaremba and Sutskever, 2014a] and its noisy variant on Learning To Execute [Zaremba and Sutskever, 2014b] problem. The noisy network converges faster and reaches a higher accuracy showing that the noisy activations are helpful when attempting a highly non-trivial optimisation tasks.

length of an input sequence was set to 26 and the input values live between 0 and 10.

The model consists of an LSTM running over an input sequence with a time average pooling computed over its hidden states to obtain a fixed-size vector representation. Then a regular 1-layer deep MLP with ReLU activation function is used to output the number of unique elements in the input.

As illustrated with Eqn 4.10 the hyperparameter c is responsible for scaling the standard deviation of the injected noise. This is precisely the quantity that is annealed in the experiments in this section. The annealing starts with $c = 30$ and decreases to $c = 0.5$ following a schedule $\frac{c}{\sqrt{t+1}}$ where t is incremented by 1 every 200 minibatch updates. Table 4.5 contains the results. When working with sequences one possible measure of difficulty is the length of a sequence. Therefore a curriculum method took a form of starting with short sequences and gradually increasing the length. Please note that annealing the noise gave as good or better results than the curriculum learning.

Table 4.4: Neural machine translation on Europarl. The performance is improved by around 2 BLEU points when using existing code from [Bahdanau et al., 2014] with nonlinearities replaced by their noisy versions. Just using the hard versions of the nonlinearities is responsible for about half of the gain.

Model	Validation NLL	BLEU
Sigmoid and tanh NMT (reference model)	65.26	20.18
Hard-tanh and hard-sigmoid NMT	64.27	21.59
Noisy (**NAH**) tanh and sigmoid NMT	**63.46**	**22.57**

Table 4.5: Experimental results on the task of finding the number of unique elements in a sequence of random integers. Annealing the noise turns the training procedure into a continuation method. Note that annealing yields better results than the curriculum learning.

Model	Test error
LSTM+MLP (reference model)	33.28%
Noisy LSTM+MLP(**NAN**)	31.12%
Curriculum LSTM+MLP	14.83%
Noisy LSTM+MLP(**NAN**), annealed noise	**9.53%**
Noisy LSTM+MLP(**NANIL**), annealed noise	20.94%

It is an encouraging result because coming up with a curriculum often involves problem-specific information and precious time of human experts. Annealing, on the other hand, is more general and there is no reason to believe that coming up with a well-performing schedule requires any domain knowledge. Moreover searching for desirable learning schedules can be automated and scaled up using modern computational resources.

The next challenge involves a Neural Turing Machine (NTM) and the associative recall task [Graves et al., 2014]. NTM is an impressive model with extraordinary properties. However, often it is not the easiest to train. The motivation is to find out if using noisy activations functions with a noise annealing schedule can alleviate the optimisation difficulties.

The model had a minimum of 2 and a maximum of 16 items. Annealed noisy activation functions were placed in the NTM's controller. A comparison with a regular NTM in terms of validation error can be found in Figure 4.11. NTM using noisy activations converges faster and quickly solves the task, whereas the original network struggles to minimise the loss.

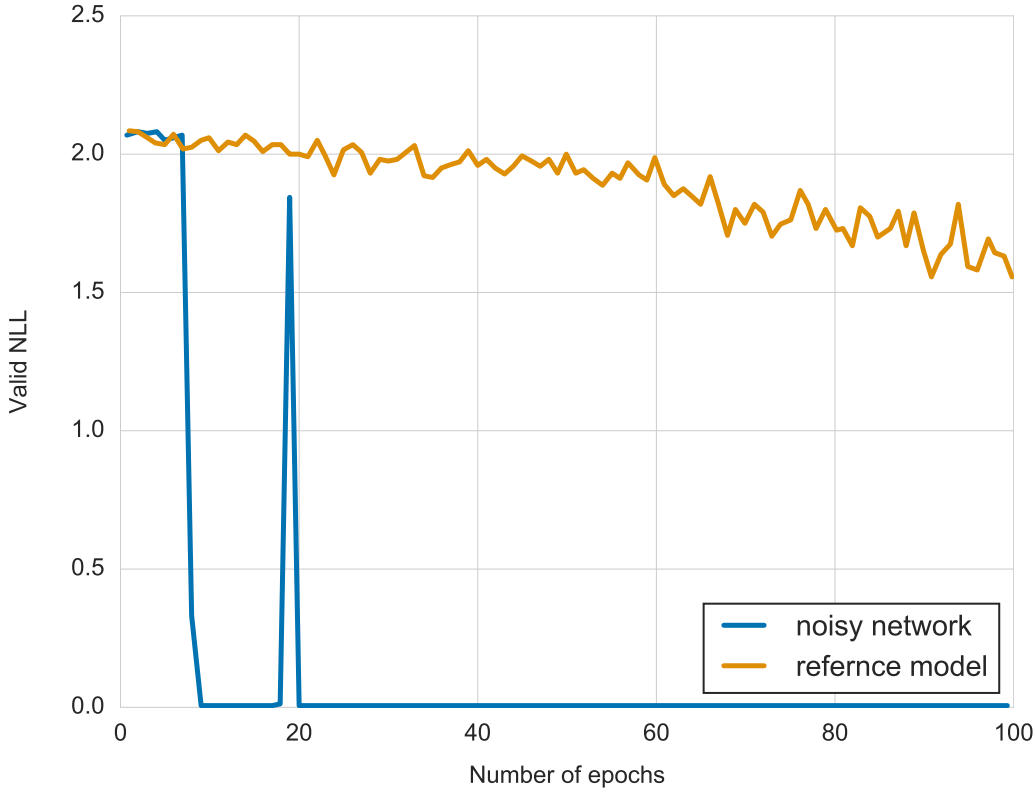


Figure 4.11: Learning curve on a validation set for the NTM model on the associative recall task. Both models were evaluated with items of length 2 and 16. The NTM with noisy controller converges quicker and solves the task.

4.3.7 Conclusion

Nonlinearities in neural networks are both a blessing and a curse. They are a blessing because they allow representing more complex functions. They are a curse because in general optimising sophisticated nonlinear models is more difficult. As a methodological example consider hard-sigmoid and hard-tanh which were experimentally verified to improve results on a set of problems. However, differentiating hard-sigmoid and hard-tanh when the input argument is either very small or very large provides no useful information to an optimiser and the unit gets ”stuck” at its current value. While it is not impossible for a neural network to recover from this state, it tends to take many iterations over the data and it imposes a significant computational cost.

Such problems are ordinary when training deep neural networks. Various strategies have been proposed to mitigate those obstacles. Especially the curriculum learning approach seems to stand out, probably due to its excellent performance. Additionally, it

should be emphasised that the intuition behind it is very convincing to humans, because we often learn difficult things in a similar way.

Noisy activation functions are highly motivated by prior work on softened nonlinearities and on the concept of annealing. It is a general framework for injecting noise into nonlinear functions minimised with stochastic gradient descent. The amount of noise is related to the extent of exploration performed during learning. Noisy activation functions provide a clean and elegant way of parameterising the exploration and adapting it together with the rest of the model. Furthermore, their formulation allows backpropagating through otherwise saturating activation functions.

The injection can take place at the output but alternatively also at the input to the function. Although results seem to be slightly better with the former, the experiments generally suggest that solving challenging tasks becomes easier regardless of the choice. The improvements pertain to the training and the test error. One plausible explanation for this behaviour is the increased regularisation.

Furthermore, it was found that the proposed noisy activations outperform their sigmoid and tanh counterparts even if the noise distribution is not adaptable and remains fixed.

Finally, it is worth noticing that the noise annealing benchmarks similarly to handcrafted curriculum learning. With a potential to eliminate human expert dependent interventions to a learning algorithm, annealed noisy activation functions are an intriguing direction for future research.

4.4 Mollifying Networks

4.4.1 Introduction

Deep neural networks i.e. convolutional networks [LeCun et al., 1989], LSTMs [Hochreiter and Schmidhuber, 1997c] and GRUs [Cho et al., 2014] achieve state of the art results on a range of challenging tasks such as object classification, object detection [Szegedy et al., 2014], semantic segmentation [Visin et al., 2015], speech recognition [Hinton et al., 2012], statistical machine translation [Sutskever et al., 2014, Bahdanau et al., 2014], playing Atari [Mnih et al., 2013] and Go [Silver et al., 2016]. However, deep models can be hard to optimise when trained with variants of SGD [Bottou, 1998] due to a highly nonlinear and a nonconvex nature of their loss surface [Choromanska et al., 2014, Dauphin et al., 2014].

Many approaches were proposed to alleviate the difficulty of optimisation such as batch normalisation [Ioffe and Szegedy, 2015] intended to address the problem of an internal covariate shift, learning with a curriculum [Bengio et al., 2009b] and training with continuation methods like diffusion [Mobahi, 2016]. The impact of noise injection on the

behaviour of modern deep models has been explored in [Neelakantan et al., 2015b] and it has been shown that noisy activation functions improve performance on a wide variety of tasks [Gulcehre et al., 2016].

This chapter connects the ideas of curriculum learning and continuation methods with those arising from models with skip connections and with layers that compute near-identity transformations. Skip connections allow training very deep residual and highway architectures [He et al., 2015, Srivastava et al., 2015] by skipping layers or blocks of layers. Moreover, it is now known that it is possible to stochastically change the depth of a network during training [Huang et al., 2016b] and converge to a very good solution.

I focus on ideas inspired by mollification which is a form of differentiable smoothing of the loss function. It is related to noisy activation functions which can be interpreted as a form of adaptive noise injection with a single hyperparameter. Influenced by neural networks with stochastic depth [Huang et al., 2016b] I exploit a similar mechanism to stochastically control the depth of a network. It allows starting the optimisation in an easier setting. In the limit the initial objective function is convex as long as the optimised criterion is convex (e.g. linear regression). This behaviour is gradually annealed making the network deeper and increasingly different from the initial form so that the model can compute more complex functions.

4.4.2 Mollifying objective functions

This section describes continuation methods and annealing in more detail. Then the main idea is presented followed by an illustration of how gradually unleashing the full potential of the model can simplify the optimisation.

4.4.2.1 Continuation and annealing methods

Continuation methods (specifically *simulated annealing*) provide a general strategy to reduce the impact of local minima and deal with nonconvex objective functions.

Continuation methods [Allgower and Georg, 1980] address a complex optimisation problem by smoothing the original function and effectively turning it into another function that is easier to optimise. If the smoothing is gradually reduced then the training procedure can be interpreted as solving a sequence of optimisation problems which eventually converges to the optimisation problem of interest (see Fig. 4.12).

These methods are very successful in tackling optimisation problems involving nonconvex objective functions with multiple local minima and potentially with non-differentiable points in their domains.

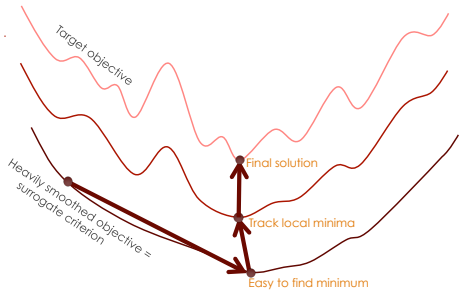


Figure 4.12: A sequence of optimisation problems of increasing complexity. The early ones (at the bottom) are significantly easier to solve, but it is the late ones (at the top) that correspond to the actual problem. One can imagine approaching these problems in order: each time starting at a reasonably good solution to the previous problem and optimising for the best local minimum on the current one. Then the process repeats.

The principle of defining a sequence of gradually more difficult training tasks (or training distributions) that converges to the task at hand had an impact on the machine learning community. In particular, approaches like curriculum learning [Bengio et al., 2009b] are an example. Gradient-based optimisation over a sequence of mollified objective functions has been shown to converge in [Chen, 2012].

Although typically there is no access to a closed form of the smoothed objective function, it is often possible to obtain an estimator. This key observation allows efficient computation of an approximation of the derivatives of the smoothed function under mild assumptions.

How such a sequence of smoothed loss functions can be obtained in practice? The aforementioned mollification can be an inspiration.

Definition 4.4.1. Mollifiers A function K is a mollifier if it is infinitely differentiable and for any integrable function $\mathcal{L}$ it satisfies

$$\mathcal{L}(\boldsymbol{\theta}) = \lim_{\epsilon \to 0} \int \epsilon^{-1} K(\mathbf{x}/\epsilon) \mathcal{L}(\boldsymbol{\theta} - \mathbf{x}) d\mathbf{x}$$

When rescaled appropriately a mollifier converges to the Dirac delta function. The high-level idea here is to construct a *sequence of mollifiers* indexed by eventually decreasing ϵ which in the above integral corresponds to $\epsilon^{-1} K(\mathbf{x}/\epsilon)$. It produces a sequence of gradually closer approximations to $\mathcal{L}(\theta)$. The next section introduces inspired generalisations of the concept of a mollifier.

4.4.2.2 Generalised and Noisy Mollifiers

In the context of optimisation via a continuation method, it is useful to introduce a concept of a generalised mollifier and of a noisy mollifier.

Definition 4.4.2. (Generalised Mollifier). A generalised mollifier is a transformation $T_\sigma(f)$ of a function f such that

$$\lim_{\sigma \to 0} T_\sigma(f) = f \tag{4.20}$$

$$\lim_{\sigma \to \infty} T_\sigma(f) = f^0 \text{ is convex} \tag{4.21}$$

$$\frac{\partial(T_\sigma(f))(x)}{\partial x} \text{ exists almost everywhere } \forall \sigma > 0 \tag{4.22}$$

Let us define a noisy mollifier which is based on the generalised mollifier and can be described as the expected value of a stochastic function $\phi(x, \xi)$ under a noise source producing ξ with variance σ^2.

Definition 4.4.3. (Noisy Mollifier). A stochastic function $\phi(x, \xi)$ with an input x and a noise ξ is called a noisy mollifier if its expected value corresponds to the application of a generalised mollifier T_σ

$$(T_\sigma(f))(x) = E_\xi[\phi(x, \xi)], \text{ where } \mathrm{Var}(\xi) = \sigma^2 \tag{4.23}$$

The composition of two noisy mollifiers sharing the same σ is a noisy mollifier since the three properties in the definition (Eq. 4.20, Eq. 4.21, Eq. 4.22) are satisfied. When $\sigma \to 0$ there is no noise and therefore it is equivalent to the original objective function. On the other hand when $\sigma \to \infty$ the function becomes convex.

Consequently, for example separately corrupting all activation functions of a deep neural network by noise injection with a shared and annealed noise level σ yields a noisy mollifier for the objective function. It is related to the recent work of [Mobahi, 2016] which introduced analytic smoothing of neural network's nonlinearities to improve training of recurrent models.

[Mobahi, 2016] also establishes a link between noise injection and continuation methods although an earlier form of that observation is also present in [Bottou, 1991b] in the context of gradually decreasing the learning rate when using stochastic gradient descent. The idea of injecting noise into a hard-saturating nonlinearity was previously used in [Bengio, 2013] to help backpropagate signals through semi-hard decisions with the "noisy rectifier" stochastic nonlinearity.

The noisy mollifier used in this work is not limited to functions that allow for an analytic approach. A well-performing, practical form of the noisy mollifier has been discovered experimentally.

4.4.3 The method

The focus of this chapter is improving the optimisation of neural networks with popular activation functions such as ReLU$(\cdot)$, tanh$(\cdot)$ or sigmoid$(\cdot)$. Some of these nonlinearities play a crucial role in models involving gating (e.g. LSTM, GRU) and are known to be particularly challenging to optimise. However general principles presented here can be extended to other functions.

Existing noisy training procedures rely on principled ways of injecting noise to the activations or to the gradient of a network. However, such noise injection has drawbacks. As the noise gets larger it can dominate the learning process and lead the algorithm to perform a long random walk on the landscape of the objective function. Moreover, a single drastically noisy update to the parameters of the model can potentially significantly damage or even ruin the whole training. A novel algorithm proposed here does not suffer from these issues. As the amount of noise gets large SGD minimises a simpler, but still meaningful objective. The definition of noisy mollifiers describes the desired behaviour in the limit cases where the noise is infinitely large or infinitely small.

The main innovation is that instead of directly modifying the loss function by using noise, one can adjust the complexity of a neural network with a similar motivation in mind. One way of controlling the complexity is to reduce some of the nonlinearities to specific linear transformations or completely skip preselected computational blocks. As a result the depth of the model is decreased and the count of parameters is shrinked. The model is simplified. Importantly, it also changes the landscape of the loss function. For example the number of local minima depends on the number of parameters and the depth of the network. Therefore architectural changes replace the mechanism of noise injection in simplifying the landscape of optimisation.

During training the algorithm minimises a sequence of noisy objectives

$$\mathrm{L} = (\mathcal{L}_1(\boldsymbol{\theta}_1; \xi_{\sigma_1}), \mathcal{L}_2(\boldsymbol{\theta}_2; \xi_{\sigma_2}), ..., \mathcal{L}_k(\boldsymbol{\theta}_k; \xi_{\sigma_k}))$$

where the scale (standard deviation) of the noise σ_i is reduced and as a result the complexity of the network is increased.

Typically the mollification of the objective function is achieved by performing a convolution with a well-chosen kernel. Instead, this work proposes to reformulate the training procedure.

Each layer of a mollifying network is indexed by l and has a scalar hyperparameter $p^l \in [0, 1]$ which controls its amount of noise. Start by optimising a highly simplified objective which is obtained by setting the noise level for all layers between the input and

the last non-cost layer to the maximum possible value of 1. A layer can be activated or completely skipped depending on the noise level p^l. At a high level of noise the layers of the network are skipped with high probability. What consitutes a single layer of a mollifying network is a decision characteristic to a given model. It is a computational block which sometimes can contain more than a single layer of a regular deep neural network. Furthermore, nonlinearities are constructed in a way that produces a linear behaviour when the amount of noise is high. Both properties are controlled by the scalar p^l per layer.

As the noise level p^l is annealed the skipping of the layers happens less often. The individual layers are more active now and they escalate model's complexity. Additionally, decreasing the level of noise allows the nonlinearities, which usually are element-wise activation functions, to change their behaviour from linear to increasingly nonlinear.

These improvements over a regular deep network allow one to control the model's capabilities. The annealing of the noise level p^l transforms the model from a very simple form to its full potential.

Note that this kind of noisy training potentially can improve generalisation because the noise in the mollified model results in the noise in backpropagation. Typically with more noise introduced to backpropagation through the noisy units [Hochreiter and Schmidhuber, 1997a], SGD is more likely to converge to a flatter minimum, because the chance of getting trapped in a local minimum is smaller.

Linearizing the nonlinearities Section 4.4.2 shows that convolving the objective function with a particular kernel can be approximated by adding a specific noise to the activation function. However, injecting an unbounded noise may result in an excessive random exploration and may introduce problems with the stability of optimisation. Occasionally the sampled noise can be extreme, which is likely to harm the training procedure.

At this point it is helpful to make sure that the activation function takes value 0 at $x = 0$ for simplicity. Usually it requires only a straightforward translation. For example in the case of sigmoid it is sufficient to translate the function by -0.5 along the y-axis. This slightly modified nonlinearity is denoted by $f(x)$.

To address the issue of the excessive noise the activation function needs to be modified. One possible solution is to make sure that when the standard deviation of the noise approaches infinity ($\sigma \to \infty$), the activation function becomes a linear function.

Which linear function should it be? In this work it is the first-order approximation of the original activation function at $x = 0$ and is referred to as $u(x)$ for the rest of the chapter.

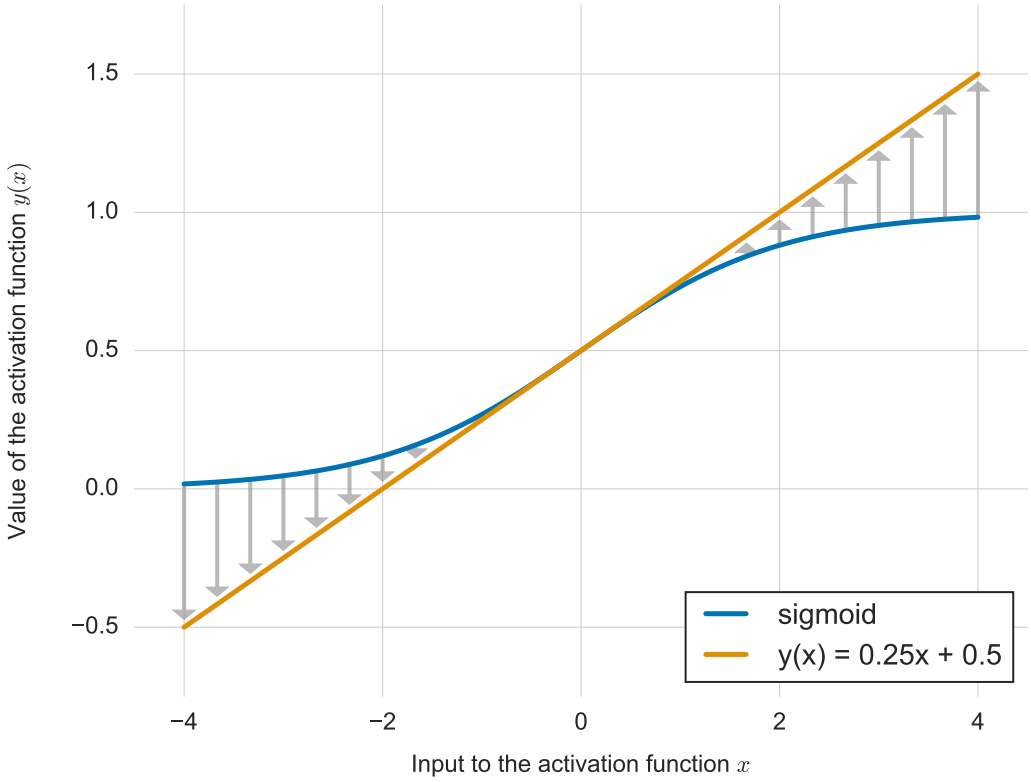


Figure 4.13: A depiction of the conceptual result of noise injection to the sigmoid function. Arrows denote the direction of the noise. It pushes the behaviour of the activation function towards a linear function.

The resulting function $\psi(x)$ takes value 0 at $x = 0$ and exhibits a linear behaviour when the amount of noise is maximised. The concept is illustrated in Figure 4.13 for the sigmoid nonlinearity.

Let h_i^{l-1} be the ith dimension of the vector of activations of the previous layer $\mathbf{h}^{l-1}$. It is the input to the nonlinearity. Then

$$\begin{aligned} h_i^l &= \psi(h_i^{l-1}, \xi_i^l) \\ &= \text{sgn}(u(h_i^{l-1}))\ \text{min}[\ |u(h_i^{l-1})|, |f(h_i^{l-1}) + \text{sgn}(u(h_i^{l-1}))|s_i^l||\ ] \end{aligned} \tag{4.24}$$

where

$$s_i^l = 0 + c\, p^l \sigma(h_i^{l-1})\ |\xi_i^l| \tag{4.25}$$

$$\xi_i^l \sim \mathcal{N}(0,\ 1) \tag{4.26}$$

where c is a hyperparameter, p^l is an annealed parameter and $\sigma(h_i^{l-1})$ is a function controlling the variance of the noise depending on the input h_i^{l-1} in a similar fashion to Eqn 4.10.

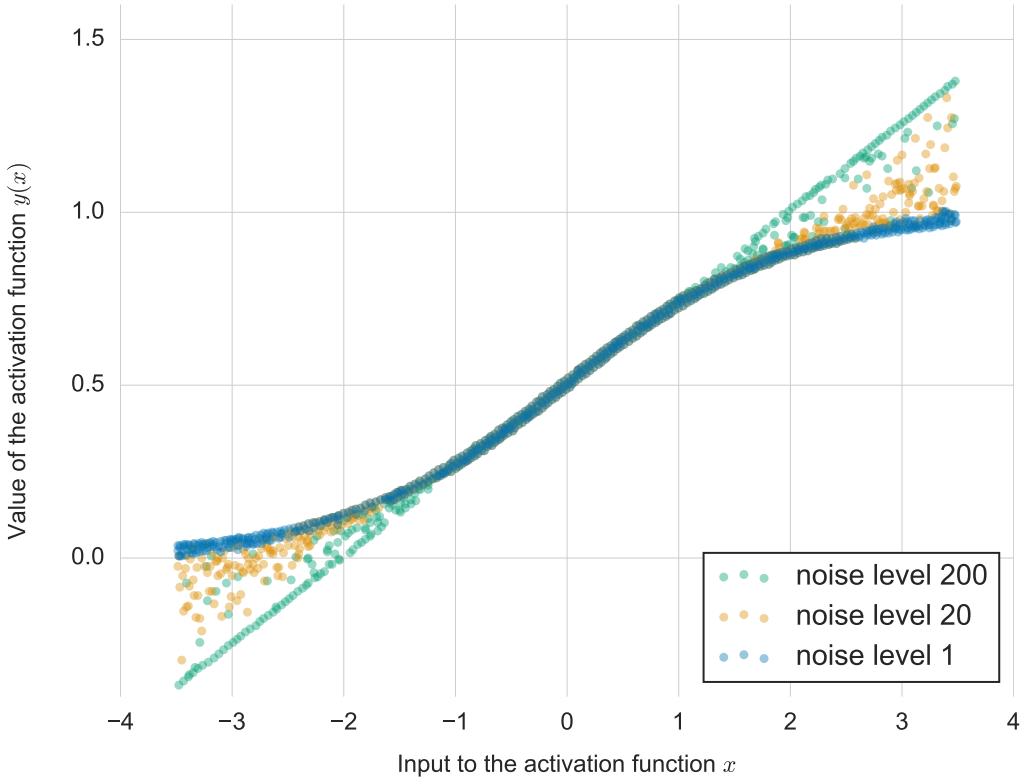


Figure 4.14: Empirical evaluation of the behaviour of a noisy sigmoid. Adding more noise makes sigmoid(x) behave closer to a linear function. The noise level is the product $p^l * c$.

When p^l is large the activation function $\psi(x)$ with high probability becomes a linear function of the input. Figure 4.14 depicts an empirical evaluation of the behaviour of a noisy sigmoid with different amounts of injected noise.

The pseudo-code for the mollified activation function is provided in Algorithm 2.

Algorithm 2 Mollified activation function applied to unit i at layer l

1: $\Delta_i^l \leftarrow u(h_i^{l-1}) - f(h_i^{l-1})$ # Δ_i^l is a measure of saturation of the unit
2: $\sigma(h_i^{l-1}) \leftarrow (\text{sigmoid}(a_i \Delta_i^l) - 0.5)^2$ # std. dev. of the injected noise depends on Δ_i
3: $\xi_i^l \sim \mathcal{N}(0,\ 1)$ # sampling the noise from the standard normal distribution
4: $s_i^l \leftarrow\ p^l\ c\ \sigma(x_i^l)|\xi_i^l|$ # half-normal noise controlled by $\sigma(h_i^{l-1})$, constant c and p-ty p^l
5: $\psi(h_i^{l-1}, \xi_i^l) \leftarrow \text{sgn}(u(h_i^{l-1}))\ \text{min}[\ |u(h_i^{l-1})|,\ |f(h_i^{l-1}) + \text{sgn}(u(h_i^{l-1}))|s_i^l|\,|]$
6: $\tilde{h}_i^l = \psi(h_i^{l-1}, \xi_i^l)$ # $\tilde{h}_i^l$ is the mollified activation

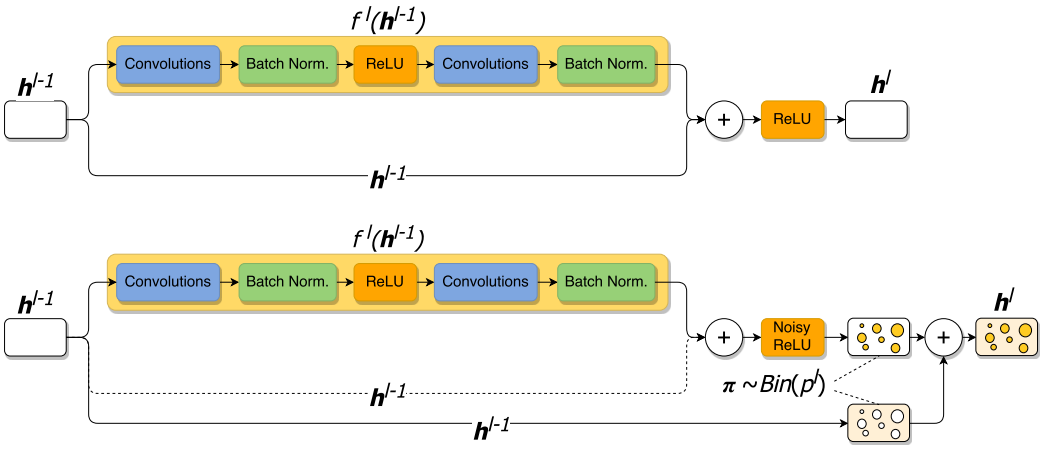


Figure 4.15: **Top:** Stochastic Depth neural network. **Bottom:** Mollifying network. In the top path of the Mollifying Network the input is processed with a convolutional block followed by a noisy activation function. In this example the activation function is noisy ReLU. In the bottom path the original activation of the layer $l-1$ is propagated untouched. For each unit one of the two paths is picked according to the binary decision vector $\boldsymbol{\pi}$. The dashed line represents the optional residual connection.

4.4.3.1 Mollifying feedforward networks

Consider a feedforward neural network. Figure 4.15) is an example of a computational block of a mollifying network built using a residual connection, convolutional layers and a mollified activation function. During the forward pass for each unit either copy the activations from the previous layer or use the output computed by the block. The binary decision between the two options described above is performed independently for each unit at layer l and is expressed as a binary vector $\boldsymbol{\pi}^l$. The noise $\boldsymbol{\xi}^l$ is injected into all mollified activation functions participating in the computation.

$$\tilde{\mathbf{h}}^l = \psi(\mathbf{h}^{l-1}, \boldsymbol{\xi}^l) \tag{4.27}$$

$$\phi(\mathbf{h}^{l-1}, \boldsymbol{\xi}^l, \boldsymbol{\pi}^l) = \boldsymbol{\pi}^l \odot \mathbf{h}^{l-1} + (1 - \boldsymbol{\pi}^l) \odot \tilde{\mathbf{h}}^l \tag{4.28}$$

$$\mathbf{h}^l = \phi(\mathbf{h}^{l-1}, \boldsymbol{\xi}^l, \boldsymbol{\pi}^l) \tag{4.29}$$

The binary decision vector $\boldsymbol{\pi}^l$ is sampled from Bernoulli distribution with the probability parameterised by the value of p^l

$$\boldsymbol{\pi}^l \sim \text{Bin}(p^l) \tag{4.30}$$

If the number of hidden units of the layer $l-1$ and the number of hidden units of the layer l are not the same, one can apply a zero padding or a linear projection to the layer $l-1$ to obtain the right dimensionality. When $p^l = 1$ the layer computes the identity function.

When $p^l = 0$ the layer computes the nonlinear transformation unfolding the full capacity of the model.

4.4.3.2 Mollifying LSTMs and GRUs

In a similar way it is possible to modify an objective function of LSTM or GRU networks by controlling the capacity of the model. For example at the beginning of training one can be optimising a word2vec, BoW-LM or CRF objective and gradually raise the difficulty by increasing the capacity of the network.

To perform an equivalent procedure for GRUs one can use Algorithm 2 for the mollification of the activation function and set the update gate to $\frac{1}{t}$ where t is the annealing timestep and reset the gate to 1 when the noise is very large. For LSTMs one can set the output gate to 1 or close to 1, set the input gate to $\frac{1}{t}$ and set the forget gate to $1 - \frac{1}{t}$ when the noise is very large. This way the LSTM will behave like a BoW model.

To achieve this behaviour the model has to be able to compute a function $\psi(x_t^l, \xi_t^l)$ such that $\mathbb{E}_{\xi_t^l}[\psi(x_t^l, \xi_t^l)] = t$ and $\psi(x_t^l, \xi_t^l)$ is differentiable with respect to x_t^l. Consider

$$\psi(x_t^l, \xi_t^l) = f(x_t^l + p_t^l \sigma(x_t^l)|\xi_t^l|)$$

where $f(x)$ represents the hard-sigmoid activation function. Assume $f(x)$ can be approximated well enough by a linear function. Let $f_{lin}(x)$ be the first order Taylor expansion of $f(x)$ around $x = 0$

$$f(x)_{x=0} \approx f_{lin}(x) = \frac{1}{4}x + \frac{1}{2}$$

Then let us assume

$$\mathbb{E}_{\xi_t^l}[f(x_t^l + p_t^l \sigma(x_t^l)|\xi_t^l|)] \approx \mathbb{E}_{\xi_t^l}[f_{lin}(x_t^l + p_t^l \sigma(x_t^l)|\xi_t^l|)]$$

Note that

$$\begin{aligned}\mathbb{E}_{\xi_t^l}[f_{lin}(x_t^l + p_t^l \sigma(x_t^l)|\xi_t^l|)] &= f_{lin}(\mathbb{E}_{\xi_t^l}[x_t^l + p_t^l \sigma(x_t^l)|\xi_t^l|]) \\ &= f_{lin}(x_t^l + p_t^l \sigma(x_t^l)\mathbb{E}_{\xi_t^l}[|\xi_t^l|])\end{aligned}$$

What should be the form of $\sigma(x_t^l)$ so $f_{lin}(x_t^l + p_t^l \sigma(x_t^l)\mathbb{E}_{\xi_t^l}[|\xi_t^l|]) = t$ is satisfied?

$$\begin{aligned}f_{lin}^{-1}(t) &= x_t^l + p_t^l \sigma(x_t^l)\mathbb{E}_{\xi_t^l}[|\xi_t^l|] \\ \sigma(x_t^l) &= \frac{f_{lin}^{-1}(t) - x_t^l}{\mathbb{E}_{\xi_t^l}[|\xi_t^l|]} \text{ when } p^l = 1\end{aligned}$$

Where

$$f_{lin}^{-1}(x) = 4(x - \frac{1}{2})$$

With the proposed $\sigma(x_t^l)$ the scalar $\mathbb{E}_{\xi_t^l}[\psi(x_t^l, \xi_t^l)]$ approximately takes discrete values from the domain of t (up to the approximation) while $\psi(x_t^l, \xi_t^l)$ remains differentiable with respect to x_t^l.

4.4.3.3 Annealing schedule for p

Each layer a mollifying network uses a separate annealing schedule. The motivation behind it is to allow lower layers to anneal faster. This is similar to the linearly decaying probability of layers in [Huang et al., 2016b]. The experiments use an annealing schedule similar to the inverse sigmoid rule found in [Bengio et al., 2015] with

$$p_t^l = 1 - e^{-\frac{kv_t l}{tL}} \tag{4.31}$$

where t is the index of the update, $k \geq 0$ is a hyperparameter, L is the number of layers of the model and v_t is the moving average of the loss*. The annealing stops when the expected depth $p_t = \sum_{l=1}^{L} p_t^l$ reaches a threshold δ.

Thanks to v_t the optimisation can directly influence the annealing schedule through the value of the loss. Note that

$$\lim_{v_t \to \infty} p_t^l = 1 \text{ and } \lim_{v_t \to 0} p_t^l = 0. \tag{4.32}$$

When the loss is high the noise injected into the system is large and the model is encouraged to explore more. As the training progressed the loss is decreasing and the amount of noise is reduced. Naturally, the number of steps t also drives the annealing.

4.4.4 Experiments

The experiments were performed mainly on deep MLPs with sigmoid or tanh activation functions that are difficult to train.

4.4.4.1 Deep feedforward networks

Deep Parity Training neural networks on a high-dimensional parity problem can be challenging [Graves, 2016, Kalchbrenner et al., 2015]. The 40-dimensional parity problem is approached with a 6-layer MLP using sigmoid activation function. All the models are initialised with Glorot method [Glorot et al., 2011] and trained with SGD with momentum equal to 0.92, which is only a slight change of the default value of 0.9. The learning rate is 0.001. Parameters a_i present in the mollified activation function are sampled from $\mathcal{U}[-2, 2]$.

* This can be the moving average of training or of validation loss depending on whether the model tends to overfit.

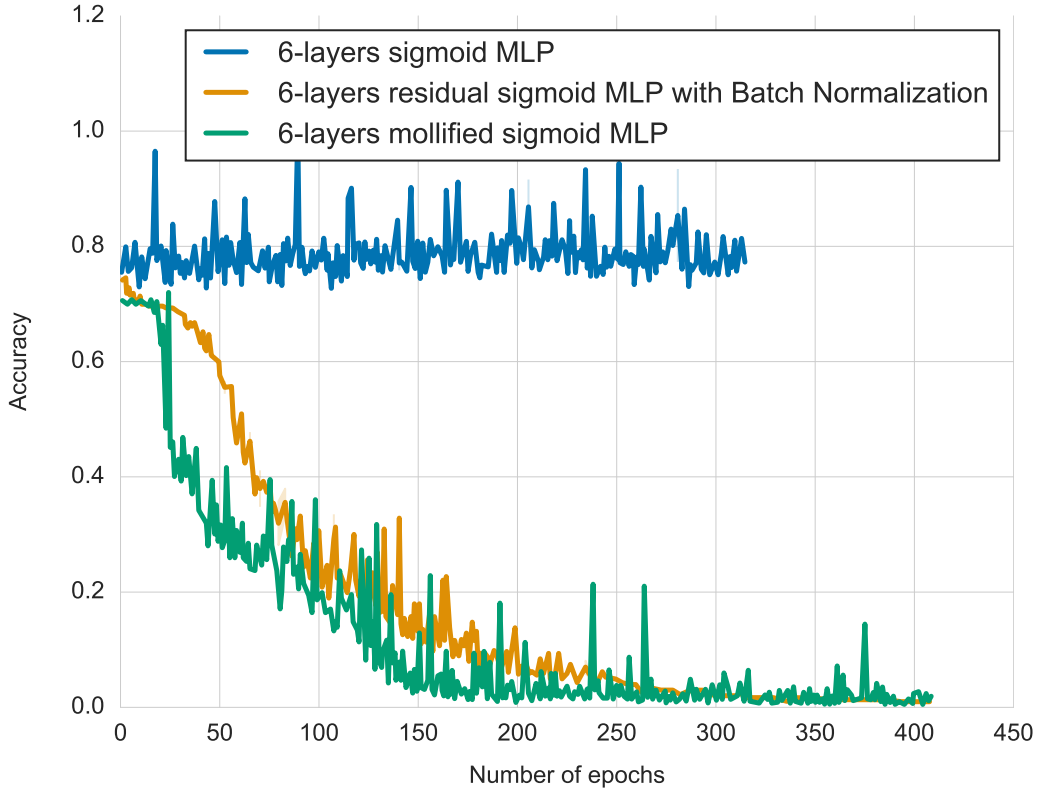


Figure 4.16: The learning curves of the standard MLP, the mollified MLP and the residual MLP on the 40-bit parity task.

That model is compared with an MLP with residual connections and batch normalisation. Figure 4.16 shows that while a regular MLP with the standard sigmoid activation function does not learn at all, the mollified network converges faster than the residual network.

Deep Pentomino Pentomino is a toy image dataset. Each image has three Pentomino blocks. The task is to decide if all the blocks on the image are the same [Gülçehre and Bengio, 2013]. The best result reported on this task with an MLP in the literature is 68.15% accuracy [Gulcehre et al., 2014].

It turns out that an MLP with 6 layers, 200 units per layer and sigmoid nonlinearities trained with SGD with momentum using learning rate 0.001 achieves 69.5% accuracy. The same model with mollified activation functions and residual connections performs at 75.15% accuracy after 100 epochs of training on the $80k$ dataset.

CIFAR10 All models in the experiment are deep convolutional neural networks with 110 layers. The proposed architecture is equipped with residual blocks and residual connec-

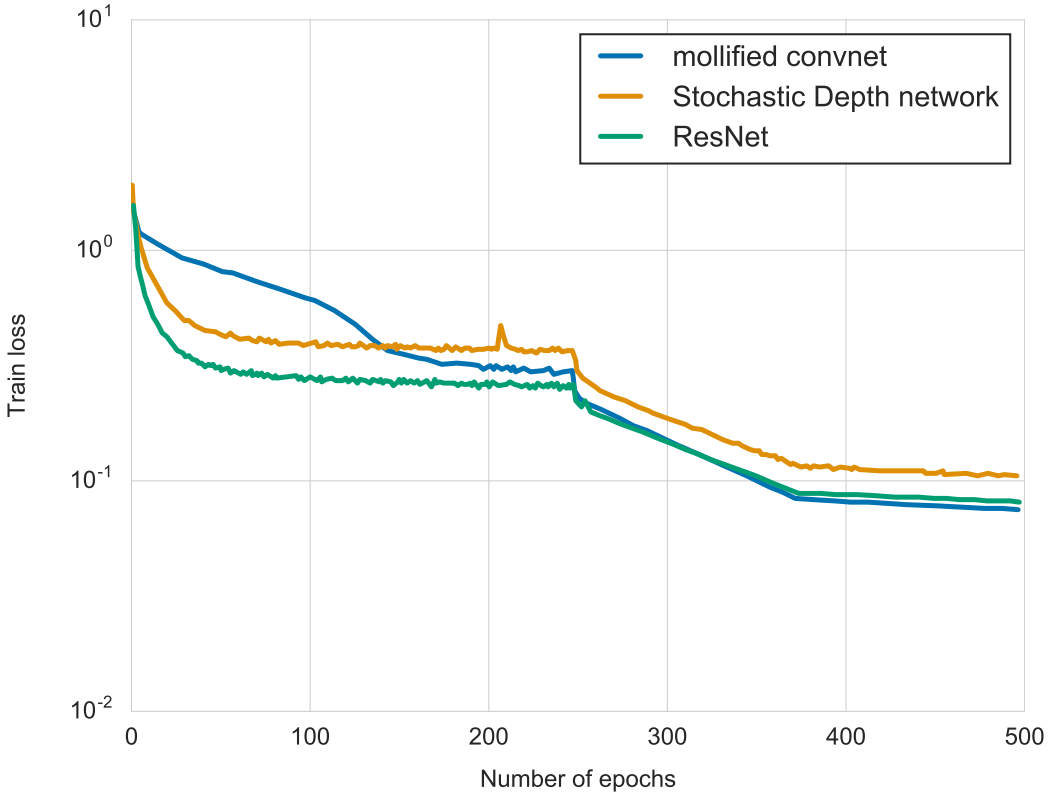


Figure 4.17: Training loss of the proposed model, Stochastic Depth and ResNet over 500 epochs on CIFAR10 dataset.

tions. It is evaluated against ResNet and Stochastic Depth [Huang et al., 2016a]. The hyperparameters are directly imported from the Stochastic Depth network.

Training and validation curves of the three variants are reported in Figure 4.17 and Figure 4.18 respectively. Furthermore, Table 4.6 lists the best test set accuracy for models obtained by early stopping on the validation set.

Stochastic Depth achieves the best generalisation of all three algorithms. Nevertheless mollified network performs better than ResNet and it might be possible to combine it with Stochastic Depth architecture.

4.4.4.2 LSTM experiments

Language Modelling This task is concerned with word-level language modelling on PenTree Bank dataset. The proposed network and the reference network are stacked LSTMs with 2 layers. Both models have the same hyperparameters identical to those found in Learning to Execute [Zaremba and Sutskever, 2014b]. Please note that the hard-sigmoid

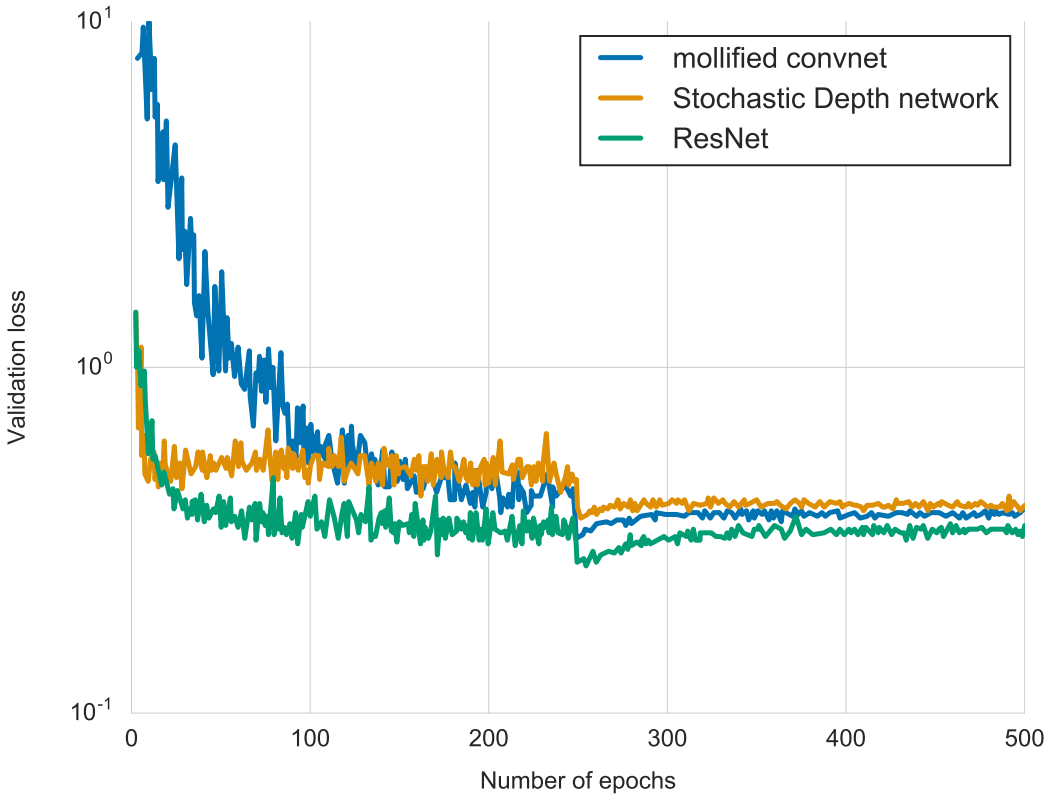


Figure 4.18: Validation loss of the proposed model, ResNet and Stochastic Depth over 500 epochs on CIFAR10 dataset.

Model	Test Accuracy
Stochastic Depth	93.25 %
Mollified Convnet	92.45 %
ResNet	91.78 %

Table 4.6: Test accuracy of the proposed model, ResNet and Stochastic Depth on CIFAR10 over 500 epochs.

Model	Test PPL
LSTM	128.4
Mollified LSTM	123.6

Table 4.7: Evaluation of mollification on word-level language modelling on PenTree Bank dataset. Both models are 2 layers stacked LSTMs with the same hyperparameters.

activation function was used in the mollified variant as well as in the baseline. It is a suitable setup for evaluation of benefits of mollification on this problem.

Please refer to Table 4.7 for results. The mollified model achieves better results. It was also observed that it converges faster.

Predicting Embeddings from Characters Learning the mapping from sequences of characters to word embeddings is a difficult problem. Generally, it requires a highly expressive network which typically is not easy to optimise. In this particular case a word2vec [Mikolov et al., 2014] model with 500-dimensional embeddings is trained on Wikipedia with vocabulary size 374557. 10000 of these word embeddings are used as a validation set and another 10000 of them are used as a test set.

Each word is broken down into a sequence of characters that a bi-directional LSTM is trained on. An MLP with five layers and the tanh activation function takes the representation from the LSTM and predicts the word embedding. The optimisation algorithm is RMSProp with momentum equal to 0.92. Learning rate is 0.0006 and the size of the mini-batch is 64. The training curve visualised in Figure 4.19 shows that the mollified LSTM network converges faster.

4.4.5 Conclusion

Mollification is a novel approach to training neural networks inspired by ideas such as curriculum learning, continuation and recent advances in nonconvex optimisation. The method makes the learning easier by starting from a simpler model solving a well-behaved problem and gradually transitioning to a more complicated setting.

In this chapter it was compared with neural networks benefiting from powerful techniques such as batch normalisation and residual connections. The experiments show improvements on very deep models and tasks that are difficult to optimise.

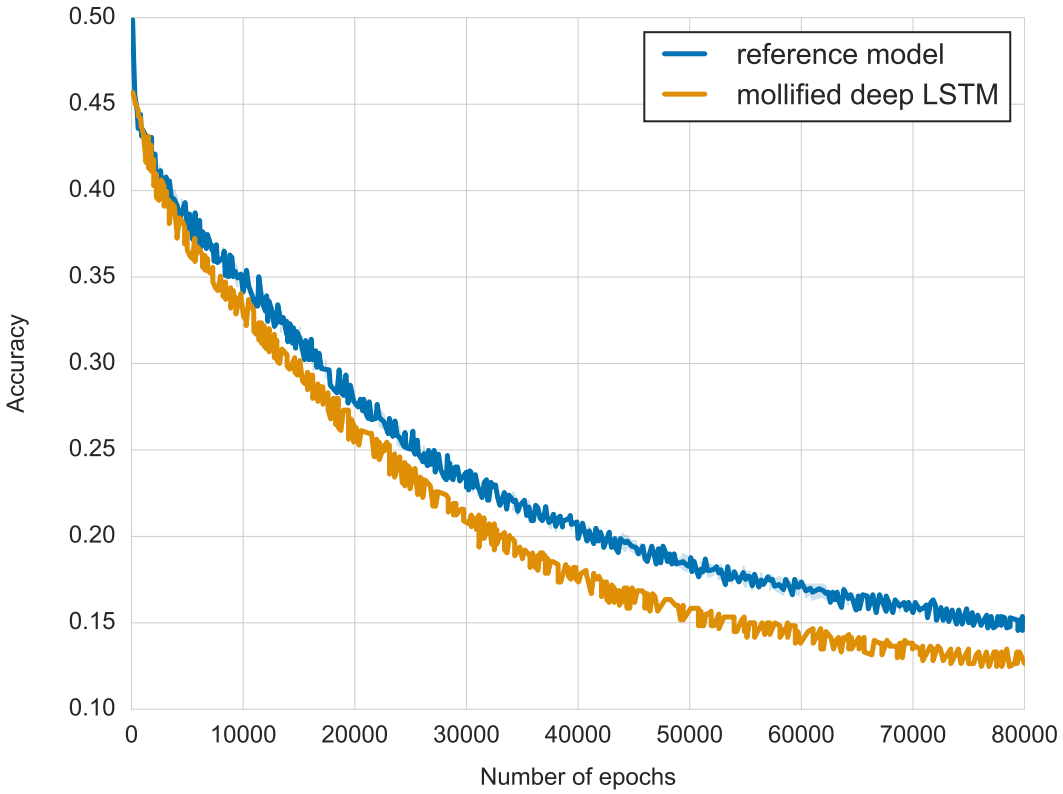


Figure 4.19: Training curve of a bi-directional LSTM that predicts the embedding corresponding to a sequence of characters.

Chapter 5

Conclusions

Reduction of parameters is an important direction of research on deep neural networks. This thesis is focused on training and evaluating networks in constrained environments. The results show that the parameters of a classical deep convolutional neural network can be compressed up to 6 times without a significant drop in accuracy. If a slight loss of performance is acceptable, the compression ratio can be even higher.

Moreover, it turns out that structured layers can replace dense matrices which are a popular building block of modern architectures. Interestingly this work and the relevant referenced literature suggest that reducing the number of parameters in the last dense layer is substantially more difficult as compared to other layers.

Those findings inspire further theoretical and practical insights and questions. How important is the size of a network in terms of the number of parameters for optimisation? Is it possible that a larger number of parameters results in redundancy and additional symmetries but at the same time it makes learning easier? Since we know that neural networks can be trained and evaluated with a radically smaller number of parameters, why is it challenging to arrive at such models? Should we look for more sophisticated optimisation algorithms?

Another important research topic undertaken in the thesis is noise injection. It can be approached in many possible ways. However, here I concentrate on applying it to activation functions. Injecting an adaptable noise to the activation function can improve training. Especially if the nonlinearity has subsets of the domain producing minimal or zero gradient. What is more, one may desire for a nonlinearity to assume discretised values for a subset of input arguments in order to make hard decisions.

Such circumstances essentially require appropriately increased exploration triggered by getting ”stuck” during optimisation. The idea of learning the distribution of the injected noise and conditioning it on the properties of the current input makes the approach flexible and more universal.

Furthermore, it is possible to control the expressiveness of a deep neural network with a noise. Assuming that a large amount of noise reduces the complexity of the model, one can use this property to adjust the difficulty of the optimisation problem that is being solved.

Mollifying networks implement this form of noise injection. This concept is intimately related to the ideas such as curriculum learning and continuation methods. Traditionally they rely on hand-crafted schedules or a hand-specified ordering of training examples, requiring a significant amount of expert knowledge.

When working with mollifying networks, it is enough to specify a noise annealing schedule to replace the need for an expert-derived intervention. The schedule can be discovered experimentally without any domain-specific knowledge. The model is optimised against gradually more and more difficult objective until the algorithm arrives at the original problem. This approach allows for training deep neural networks on tasks that are exceptionally hard to optimise.

Bibliography

[Achlioptas, 2003] Achlioptas, D. (2003). Database-friendly random projections: Johnson-Lindenstrauss with binary coins. *J. Comput. Syst. Sci.*, 66(4):671–687.

[Ailon and Chazelle, 2009] Ailon, N. and Chazelle, B. (2009). The Fast Johnson Lindenstrauss Transform and approximate nearest neighbors. *SIAM Journal on Computing*, 39(1):302–322.

[Allgower and Georg, 1980] Allgower, E. L. and Georg, K. (1980). *Numerical Continuation Methods. An Introduction*. Springer-Verlag.

[An, 1996] An, G. (1996). The effects of adding noise during backpropagation training on a generalization performance. *Neural Comput.*, 8(3):643–674.

[Bahdanau et al., 2014] Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

[Bahdanau et al., 2015] Bahdanau, D., Cho, K., and Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations*.

[Bakhtiary et al., 2015] Bakhtiary, A. H., Lapedriza, À., and Masip, D. (2015). Speeding up neural networks for large scale classification using WTA hashing. *arXiv preprint arXiv:1504.07488*.

[Bengio et al., 2015] Bengio, S., Vinyals, O., Jaitly, N., and Shazeer, N. (2015). Scheduled sampling for sequence prediction with recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 1171–1179.

[Bengio, 2013] Bengio, Y. (2013). Estimating or propagating gradients through stochastic neurons. Technical Report arXiv:1305.2982, Universite de Montreal.

[Bengio et al., 2013] Bengio, Y., Léonard, N., and Courville, A. (2013). Estimating or propagating gradients through stochastic neurons for conditional computation. arXiv:1308.3432.

[Bengio et al., 2009a] Bengio, Y., Louradour, J., Collobert, R., and Weston, J. (2009a). Curriculum learning. In *ICML'09*.

[Bengio et al., 2009b] Bengio, Y., Louradour, J., Collobert, R., and Weston, J. (2009b). Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM.

[Blundell et al., 2015] Blundell, C., Cornebise, J., Kavukcuoglu, K., and Wierstra, D. (2015). Weight uncertainty in neural networks. In *ICML*.

[Bottou, 1991a] Bottou, L. (1991a). Stochastic gradient learning in neural networks. In *Proceedings of Neuro-Nîmes 91*, Nimes, France. EC2.

[Bottou, 1991b] Bottou, L. (1991b). *Une approche théorique de l'apprentissage connexioniste; applications à la reconnaissance de la parole*. PhD thesis, Université de Paris XI.

[Bottou, 1998] Bottou, L. (1998). Online algorithms and stochastic approximations. In Saad, D., editor, *Online Learning in Neural Networks*. Cambridge University Press, Cambridge, UK.

[Charikar et al., 2004] Charikar, M., Chen, K., and Farach-Colton, M. (2004). Finding frequent items in data streams. *Theoretical Computer Science*, 312(1):3–15.

[Chen et al., 2015] Chen, W., Wilson, J. T., Tyree, S., Weinberger, K. Q., and Chen, Y. (2015). Compressing neural networks with the hashing trick. In *ICML*.

[Chen, 2012] Chen, X. (2012). Smoothing methods for nonsmooth, nonconvex minimization. *Math. Program. Ser. B*, 134:71–99.

[Cheng et al., 2015] Cheng, Y., Yu, F. X., Feris, R., Kumar, S., Choudhary, A., and Chang, S.-F. (2015). An exploration of parameter redundancy in deep networks with circulant projections. In *ICCV*.

[Cho et al., 2014] Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *Empiricial Methods in Natural Language Processing*.

[Cho and Saul, 2009a] Cho, Y. and Saul, L. K. (2009a). Kernel methods for deep learning. In Bengio, Y., Schuurmans, D., Lafferty, J. D., Williams, C. K. I., and Culotta, A., editors, *Advances in Neural Information Processing Systems 22*, pages 342–350. Curran Associates, Inc.

[Cho and Saul, 2009b] Cho, Y. and Saul, L. K. (2009b). Kernel methods for deep learning. In *Advances in neural information processing systems*, pages 342–350.

[Choromanska et al., 2014] Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., and LeCun, Y. (2014). The loss surface of multilayer networks.

[Collins and Kohli, 2014a] Collins, M. D. and Kohli, P. (2014a). Memory bounded deep convolutional networks. Technical report, University of Wisconsin-Madison.

[Collins and Kohli, 2014b] Collins, M. D. and Kohli, P. (2014b). Memory bounded deep convolutional networks. *CoRR*, abs/1412.1442.

[Cormode et al., 2012] Cormode, G., Garofalakis, M., Haas, P. J., and Jermaine, C. (2012). *Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches*. Foundations and Trends on databases. Now Publishers.

[Dai et al., 2014a] Dai, B., Xie, B., He, N., Liang, Y., Raj, A., Balcan, M., and Song, L. (2014a). Scalable kernel methods via doubly stochastic gradients. In *NIPS*.

[Dai et al., 2014b] Dai, B., Xie, B., He, N., Liang, Y., Raj, A., Balcan, M.-F. F., and Song, L. (2014b). Scalable kernel methods via doubly stochastic gradients. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. D., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 27*, pages 3041–3049. Curran Associates, Inc.

[Dauphin et al., 2014] Dauphin, Y., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., and Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *NIPS'2014*.

[Deng et al., 2009] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR*.

[Denil et al., 2013a] Denil, M., Shakibi, B., Dinh, L., Ranzato, M., and de Freitas, N. (2013a). Predicting parameters in deep learning. *CoRR*, abs/1306.0543.

[Denil et al., 2013b] Denil, M., Shakibi, B., Dinh, L., Ranzato, M., and de Freitas, N. (2013b). Predicting parameters in deep learning. In *NIPS*, pages 2148–2156.

[Denton et al., 2014] Denton, E., Zaremba, W., Bruna, J., LeCun, Y., and Fergus, R. (2014). Exploiting linear structure within convolutional networks for efficient evaluation. *CoRR*, abs/1404.0736.

[Evans, 1998] Evans, L. C. (1998). Partial differential equations. *Graduate Studies in Mathematics*, 19:251–258.

[Farabet et al., 2010] Farabet, C., Martini, B., Akselrod, P., Talay, S., LeCun, Y., and Culurciello, E. (2010). Hardware accelerated convolutional neural networks for synthetic vision systems. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 257–260.

[Fukushima, 1980] Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36:193–202.

[Ge et al., 2015] Ge, R., Huang, F., Jin, C., and Yuan, Y. (2015). Escaping from saddle points—online stochastic gradient for tensor decomposition. *arXiv preprint arXiv:1503.02101*.

[Glorot et al., 2011] Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 315–323.

[Golub and Van Loan, 1996] Golub, G. H. and Van Loan, C. F. (1996). *Matrix Computations*. Johns Hopkins University Press.

[Gong et al., 2014] Gong, Y., Liu, L., Yang, M., and Bourdev, L. (2014). Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*.

[Goodfellow et al., 2013] Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., and Bengio, Y. (2013). Maxout networks. *arXiv preprint arXiv:1302.4389*.

[Graves, 2016] Graves, A. (2016). Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*.

[Graves et al., 2014] Graves, A., Wayne, G., and Danihelka, I. (2014). Neural turing machines. *arXiv preprint arXiv:1410.5401*.

[Graves et al., 2015] Graves, A., Wayne, G., and Danihelka, I. (2015). Neural Turing machines. Technical report, Google DeepMind.

[Gülçehre and Bengio, 2013] Gülçehre, Ç. and Bengio, Y. (2013). Knowledge matters: Importance of prior information for optimization. *arXiv preprint arXiv:1301.4083*.

[Gulcehre et al., 2014] Gulcehre, C., Cho, K., Pascanu, R., and Bengio, Y. (2014). Learned-norm pooling for deep feedforward and recurrent neural networks. In *Machine Learning and Knowledge Discovery in Databases*, pages 530–546. Springer.

[Gulcehre et al., 2016] Gulcehre, C., Moczulski, M., Denil, M., and Bengio, Y. (2016). Noisy activation functions.

[Han et al., 2015a] Han, S., Mao, H., and Dally, W. J. (2015a). Deep compression: Compressing deep neural network with pruning, trained quantization and Huffman coding. *arXiv preprint arXiv:1510.00149*.

[Han et al., 2015b] Han, S., Pool, J., Tran, J., and Dally, W. J. (2015b). Learning both weights and connections for efficient neural networks. In *NIPS*.

[Hannun et al., 2014] Hannun, A., Case, C., Casper, J., Catanzaro, B., Diamos, G., Elsen, E., Prenger, R., Satheesh, S., Sengupta, S., Coates, A., et al. (2014). Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*.

[He et al., 2015] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*.

[Hermann et al., 2015] Hermann, K. M., Kocisky, T., Grefenstette, E., Espeholt, L., Kay, W., Suleyman, M., and Blunsom, P. (2015). Teaching machines to read and comprehend. In *Advances in Neural Information Processing Systems*, pages 1684–1692.

[Hermans and Vaerenbergh, 2015] Hermans, M. and Vaerenbergh, T. V. (2015). Towards trainable media: Using waves for neural network-style training. *arXiv preprint arXiv:1510.03776*.

[Hinton et al., 2012] Hinton, G., Deng, L., Yu, D., Dahl, G., rahman Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T., and Kingsbury, B. (2012). Deep neural networks for acoustic modeling in speech recognition. *Signal Processing Magazine*.

[Hinton et al., 2015] Hinton, G. E., Vinyals, O., and Dean, J. (2015). Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*.

[Hochreiter and Schmidhuber, 1997a] Hochreiter, S. and Schmidhuber, J. (1997a). Flat minima. *Neural Computation*, 9(1):1–42.

[Hochreiter and Schmidhuber, 1997b] Hochreiter, S. and Schmidhuber, J. (1997b). Long short-term memory. *Neural computation*, 9(8):1735–1780.

[Hochreiter and Schmidhuber, 1997c] Hochreiter, S. and Schmidhuber, J. (1997c). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

[Huang et al., 2016a] Huang, G., Sun, Y., Liu, Z., Sedra, D., and Weinberger, K. (2016a). Deep networks with stochastic depth. *arXiv preprint arXiv:1603.09382*.

[Huang et al., 2016b] Huang, G., Sun, Y., Liu, Z., Sedra, D., and Weinberger, K. Q. (2016b). Deep networks with stochastic depth. *CoRR*, abs/1603.09382.

[Huang et al., 2014] Huang, P.-S., Avron, H., Sainath, T., Sindhwani, V., and Ramabhadran, B. (2014). Kernel methods match deep neural networks on timit. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 205–209.

[Huhtanen, 2008] Huhtanen, M. (2008). Approximating ideal diffractive optical systems. *Journal of Mathematical Analysis and Applications*, 345:53–62.

[Huhtanen and Perämäki, 2015] Huhtanen, M. and Perämäki, A. (2015). Factoring matrices into the product of circulant and diagonal matrices. *Journal of Fourier Analysis and Applications*.

[Indyk and Motwani, 1998] Indyk, P. and Motwani, R. (1998). Approximate nearest neighbors: Towards removing the curse of dimensionality. pages 604–613.

[Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167.

[Jaderberg et al., 2014] Jaderberg, M., Vedaldi, A., and Zisserman, A. (2014). Speeding up convolutional neural networks with low rank expansions. *CoRR*, abs/1405.3866.

[Jaeger and Haas, 2004] Jaeger, H. and Haas, H. (2004). Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, 304(5667):78–80.

[Jia et al., 2014] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*.

[Kaiser and Sutskever, 2015] Kaiser, L. and Sutskever, I. (2015). Neural gpus learn algorithms. *CoRR*, abs/1511.08228.

[Kalchbrenner et al., 2015] Kalchbrenner, N., Danihelka, I., and Graves, A. (2015). Grid long short-term memory. *arXiv preprint arXiv:1507.01526*.

[Karpathy et al., 2015] Karpathy, A., Johnson, J., and Fei-Fei, L. (2015). Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*.

[Kirkpatrick et al., 1983] Kirkpatrick, S., Jr., C. D. G., , and Vecchi, M. P. (1983). Optimization by simulated annealing. 220:671–680.

[Krizhevsky, 2014] Krizhevsky, A. (2014). One weird trick for parallelizing convolutional neural networks. Technical report, Google.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.*, pages 1106–1114.

[Kurach et al., 2015] Kurach, K., Andrychowicz, M., and Sutskever, I. (2015). Neural random-access machines. *CoRR*, abs/1511.06392.

[Le et al., 2013] Le, Q., Sarlós, T., and Smola, A. (2013). Fastfood – approximating kernel expansions in loglinear time. In *ICML*.

[Le et al., 2015] Le, Q. V., Jaitly, N., and Hinton, G. E. (2015). A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arXiv:1504.00941*.

[LeCun et al., 2015] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

[LeCun et al., 1989] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Comput.*, 1(4):541–551.

[Lecun et al., 1998] Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324.

[Li et al., 2014] Li, H., Zhao, R., and Wang, X. (2014). Highly efficient forward and backward propagation of convolutional neural networks for pixelwise classification. *CoRR*, abs/1412.4526.

[Lin et al., 2014] Lin, M., Chen, Q., and Yan, S. (2014). Network in Network. In *ICLR*.

[Liu et al., 2015] Liu, B., Wang, M., Foroosh, H., Tappen, M., and Pensky, M. (2015). Sparse convolutional neural networks. In *CVPR*.

[Mairal et al., 2014a] Mairal, J., Koniusz, P., Harchaoui, Z., and Schmid, C. (2014a). Convolutional kernel networks. *CoRR*, abs/1406.3332.

[Mairal et al., 2014b] Mairal, J., Koniusz, P., Harchaoui, Z., and Schmid, C. (2014b). Convolutional kernel networks. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N., and Weinberger, K., editors, *Advances in Neural Information Processing Systems 27*, pages 2627–2635. Curran Associates, Inc.

[Makhoul, 1980] Makhoul, J. (1980). A fast cosine transform in one and two dimensions. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 28(1):27–34.

[Mikolov et al., 2014] Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2014). word2vec.

[Mitzenmacher and Upfal, 2005] Mitzenmacher, M. and Upfal, E. (2005). *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press.

[Mnih et al., 2013] Mnih, V., Kavukcuoglo, K., Silver, D., Graves, A., Antonoglou, I., and Wierstra, D. (2013). Playing atari with deep reinforcement learning. Technical report, arXiv:1312.5602.

[Mobahi, 2016] Mobahi, H. (2016). Training recurrent neural networks by diffusion. *arXiv preprint arXiv:1601.04114*.

[Müller-Quade et al., 1998] Müller-Quade, J., Aagedal, H., Beth, T., and Schmid, M. (1998). Algorithmic design of diffractive optical systems for information processing. *Physica D: Nonlinear Phenomena*, 120(1):196–205.

[Nair and Hinton, 2010a] Nair, V. and Hinton, G. E. (2010a). Rectified linear units improve restricted boltzmann machines. In Fürnkranz, J. and Joachims, T., editors, *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, pages 807–814. Omnipress.

[Nair and Hinton, 2010b] Nair, V. and Hinton, G. E. (2010b). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814.

[Neelakantan et al., 2015a] Neelakantan, A., Le, Q. V., and Sutskever, I. (2015a). Neural programmer: Inducing latent programs with gradient descent. *CoRR*, abs/1511.04834.

[Neelakantan et al., 2015b] Neelakantan, A., Vilnis, L., Le, Q. V., Sutskever, I., Kaiser, L., Kurach, K., and Martens, J. (2015b). Adding gradient noise improves learning for very deep networks. *CoRR*, abs/1511.06807.

[Neelakantan et al., 2015c] Neelakantan, A., Vilnis, L., Le, Q. V., Sutskever, I., Kaiser, L., Kurach, K., and Martens, J. (2015c). Adding gradient noise improves learning for very deep networks. *arXiv preprint arXiv:1511.06807*.

[Novikov et al., 2015] Novikov, A., Podoprikhin, D., Osokin, A., and Vetrov, D. (2015). Tensorizing neural networks. In *NIPS*.

[Pandey and Dukkipati, 2014] Pandey, G. and Dukkipati, A. (2014). Learning by stretching deep networks. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, pages 1719–1727.

[Rahimi and Recht, 2008] Rahimi, A. and Recht, B. (2008). Random features for large-scale kernel machines. In Platt, J. C., Koller, D., Singer, Y., and Roweis, S. T., editors, *Advances in Neural Information Processing Systems 20*, pages 1177–1184. Curran Associates, Inc.

[Reed and de Freitas, 2015] Reed, S. E. and de Freitas, N. (2015). Neural programmer-interpreters. *CoRR*, abs/1511.06279.

[Reif and Tyagi, 1997] Reif, J. and Tyagi, A. (1997). Efficient parallel algorithms for optical computing with the DFT primitive. *Applied Optics*, 36(29):7327–7340.

[Romero et al., 2015] Romero, A., Ballas, N., Kahou, S. E., Chassang, A., Gatta, C., and Bengio, Y. (2015). FitNets: Hints for thin deep nets. In *ICLR*.

[Russakovsky et al., 2014] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2014). ImageNet Large Scale Visual Recognition Challenge.

[Saade et al., 2015] Saade, A., Caltagirone, F., Carron, I., Daudet, L., Dremeau, A., Gigan, S., and Krzakala, F. (2015). Random projections through multiple optical scattering: Approximating kernels at the speed of light. *arXiv preprint arXiv:1510.06664*.

[Sainath et al., 2013] Sainath, T. N., Kingsbury, B., Sindhwani, V., Arisoy, E., and Ramabhadran, B. (2013). Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In *ICASSP*, pages 6655–6659.

[Saxe et al., 2011] Saxe, A., Koh, P. W., Chen, Z., Bhand, M., Suresh, B., and Ng, A. (2011). On random weights and unsupervised feature learning. In *ICML*, pages 1089–1096.

[Schmid et al., 2000] Schmid, M., Steinwandt, R., Mller-Quade, J., Rtteler, M., and Beth, T. (2000). Decomposing a matrix into circulant and diagonal factors. *Linear Algebra and its Applications*, 306(1–3):131–143.

[Silver et al., 2016] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

[Sindhwani et al., 2015] Sindhwani, V., Sainath, T. N., and Kumar, S. (2015). Structured transforms for small-footprint deep learning. In *NIPS*.

[Srivastava et al., 2015] Srivastava, R. K., Greff, K., and Schmidhuber, J. (2015). Training very deep networks. In *Advances in Neural Information Processing Systems*, pages 2368–2376.

[Sukhbaatar et al., 2015] Sukhbaatar, S., Szlam, A., Weston, J., and Fergus, R. (2015). End-to-end memory networks. In *NIPS*.

[Sutskever et al., 2014] Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.

[Szegedy et al., 2014] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2014). Going deeper with convolutions. Technical report, Google.

[Visin et al., 2015] Visin, F., Kastner, K., Courville, A., Bengio, Y., Matteucci, M., and Cho, K. (2015). Reseg: A recurrent neural network for object segmentation. *arXiv preprint arXiv:1511.07053*.

[Weinberger et al., 2009] Weinberger, K., Dasgupta, A., Langford, J., Smola, A., and Attenberg, J. (2009). Feature hashing for large scale multitask learning. In *ICML*, pages 1113–1120.

[Weston et al., 2014] Weston, J., Chopra, S., and Bordes, A. (2014). Memory networks. *arXiv preprint arXiv:1410.3916*.

[Xu et al., 2015a] Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A., Salakhutdinov, R., Zemel, R., and Bengio, Y. (2015a). Show, attend and tell: Neural image caption generation with visual attention. In *ICML*.

[Xu et al., 2015b] Xu, K., Ba, J., Kiros, R., Courville, A., Salakhutdinov, R., Zemel, R., and Bengio, Y. (2015b). Show, attend and tell: Neural image caption generation with visual attention. *arXiv preprint arXiv:1502.03044*.

[Xue et al., 2013] Xue, J., Li, J., and Gong, Y. (2013). Restructuring of deep neural network acoustic models with singular value decomposition. In *INTERSPEECH*, pages 2365–2369.

[Yang et al., 2015] Yang, Z., Moczulski, M., Denil, M., de Freitas, N., Smola, A., Song, L., and Wang, Z. (2015). Deep fried convnets. In *ICCV*.

[Yang et al., 2014] Yang, Z., Moczulski, M., Denil, M., de Freitas, N., Smola, A. J., Song, L., and Wang, Z. (2014). Deep fried convnets. *CoRR*, abs/1412.7149.

[Yao et al., 2015] Yao, L., Torabi, A., Cho, K., Ballas, N., Pal, C., Larochelle, H., and Courville, A. (2015). Describing videos by exploiting temporal structure. In *Computer Vision (ICCV), 2015 IEEE International Conference on*. IEEE.

[Yosinski et al., 2014] Yosinski, J., Clune, J., Bengio, Y., and Lipson, H. (2014). How transferable are features in deep neural networks? In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N., and Weinberger, K., editors, *Advances in Neural Information Processing Systems 27*, pages 3320–3328. Curran Associates, Inc.

[Zaremba and Sutskever, 2014a] Zaremba, W. and Sutskever, I. (2014a). Learning to execute. *arXiv preprint arXiv:1410.4615*.

[Zaremba and Sutskever, 2014b] Zaremba, W. and Sutskever, I. (2014b). Learning to execute. *CoRR*, abs/1410.4615.

[Zaremba et al., 2014] Zaremba, W., Sutskever, I., and Vinyals, O. (2014). Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*.