R.R.L.J. van Asseldonk

Online Learning of Sparse Network Architectures

MASTER'S THESIS

SUPERVISED BY

dr. Maarten Löffler and dr. Jyrki Alakuijala

11 APRIL 2017

Utrecht University

Department of Information
and Computing Sciences

Google Research Europe

ICA-5797780

ABSTRACT

Modern neural network architectures can have as many as hundreds of millions of parameters. This makes them power and memory-hungry, and impedes running networks on resource-constrained devices such as phones. Sparse networks can achieve performance similar to that of dense networks, with a fraction of the parameters. However, sparsification is usually done as an afterthought, without benefits in the learning phase. In this thesis, we propose to simultaneously optimise network architecture and parameters. Apart from sparsity benefits, this eliminates the need to choose a particular network architecture in advance.

Contents

1 Introduction **1**

2 Related work **3**

2.1 Metalearning 3
2.2 Sparse approximation and quantisation 4
2.3 Constrained optimisation and regularisation 6
2.4 Unsupervised learning 7
2.5 Online topological ordering 9
2.6 Maximal inner product and nearest neighbour search 10

3 Networks **13**

3.1 Supervised learning 13
3.2 Parameter optimisation 15
3.3 The space of networks 18
3.4 Networks as functions 20
3.5 Symmetries 23

4 Rewiring **29**

4.1 Edge utility 30
4.2 Applications in quantisation 34
4.3 Selecting candidate edges 35
4.4 Optimal weight updates 40
4.5 Putting it together 41
4.6 Results 45

5 Conclusion **49**

5.1 Discussion and future work 49

Bibliography **53**

CHAPTER 1

Introduction

Within the past decade, artificial neural networks have taken the world by storm. As opposed to traditional programs that compute by executing instructions, an artifical neural network computes by performing simple arithmetic operations at vertices in a large graph, where the graph defines how values flow through the network. No single vertex is essential to the computation in the way that an instruction in a program is. This makes neural networks successful at tasks that are hard to express as an algorithm. Starting from the more basic tasks such as digit classification, neural networks now power many of the services that we interact with on a daily basis, such as an autocorrecting keyboard, or text to speech in a navigation system. Networks for these applications have grown larger and larger, and with this came an increase in power consumption and bandwidth usage. Neural networks have mainly been used for applications that run inside a data center due to their power-hungry nature. Out of privacy considerations, and in order to service users who might not have a stable internet connection, there is a desire to move away from datacenters to "on-device" networks. Operating a network in a resource-constrained environment such as a mobile phone amplifies the problems of power consumption — which translates directly into battery life, bandwidth usage — which for many users is metered and limited, and storage space — which on many phones is scarce. We need to make networks smaller and more power-efficient.

Apart from the challenges in operating neural networks, there is the issue of designing them in the first place. Machine learning experts currently choose network architectures based on experience and trial and error. This is a time consuming process, and there is very little theory to guide the choice. Automating the process of designing a network would allow neural networks to be used in more contexts, and reduce iteration times.

In this thesis, we will approach both problems with a single solution. We will move away from the traditional dense networks and consider sparse networks instead, which have less parameters, and can be less expensive to evaluate if they are sufficiently sparse. A sparse network can have the expressive power of a large network, at the computational

cost of a small network. For a given amount of edges, a sparse network can be wider and deeper than a fully-connected dense network, and so the network can express a more complex model. Furthermore, rather than fixing the architecture of such a network up front, we will *learn* its architecture, and optimise architecture and parameters simultaneously. To this end, we will seek to answer the following question:

> Given a fixed number of vertices and directed edges, and possibly additional constraints such as layering constraints — what is the graph for which the corresponding artificial neural network performs optimally on a given task?

In other words, given a fixed amount of compute power, what is network architecture that uses that compute power to its full extent? Yet another way to state the question is: given a directed graph where edges indicate *potential* connections, which connections should we *instantiate*?

In this thesis we will answer those questions by developing a framework for learning sparse network architectures. The proposed method is online, in the sense that we modify the network architecture while learning parameters. Learning architecture and learning parameters are integrated in a single process, in contrast to previous work where architecture selection is something external. Unfortunately we were unable to make our method outperform a static network architecture that is fixed up front. However, we made a few fruitful discoveries along the way that have applications beyond the use case of this thesis, and we have a clear idea of how the method could be improved in future work. Optimising network architecture and parameters simultaneously is something that has received little attention in the past, and in this thesis we make a step in that direction.

In Chapter 2, we will give an overview of previous attempts to approach similar problems, as well as an overview of known solutions to subproblems that we will encounter. Chapter 3 will then define networks as they are used in this thesis, along with some of their properties, and the methods for learning network parameters. The definitions in Chapter 3 are unusual in the sense that they can describe networks with a general architecture, including non-layered networks. However, the real contribution of this thesis is in Chapter 4, where we will describe an algorithm that optimises network architecture and parameters simultaneously, a process we call *rewiring*. The development of this technique sparked many questions that we cannot hope to answer in this thesis, and it uncovered interesting paths for future work, such as verifying our method for deleting edges on more networks, exploring extensions to quantisation, and improving our method for adding edges. Chapter 5 outlines the opportunities for future work, and gives a summary of the present work.

CHAPTER 2

Related work

The issues of producing a sparse network, and learning network architecture, have traditionally been approached as two different problems. Making a network sparse is done as a one-time post-processing step, and learning architecture is an external process that does not exploit sparsity. In both cases, network architecture is seen as something rigid, and fixed. To the best of our knowledge, continuously adapting the edges in a network as it learns, is something that has not been explored before. But several lines of research can be considered a step in that direction.

An approach sometimes called *metalearning* is concerned with designing algorithms that design networks for a given task. These metalearning algorithms can themselves be based on a neural network.

A different line of research takes fully trained networks, and tries to reduce their size in various ways, in order to reduce memory footprint, or to minimise wasted compute power. As a side effect, the resulting networks can be sparser. Techniques here include *sparse approximation* and *quantisation*.

Finally, an area of research focuses on constrained optimisation, which can include sparsity constraints. This can be used to optimise network architecture and parameters simultaneously, but it is often as expensive as training a fully-connected network. The sparsity of the resulting network is only beneficial after training.

2.1 Metalearning

The problem of learning the parameters of a given neural network has been studied extensively, but coming up with a good network architecture in the first place is still more of an art than a science. Experts use experience and intuition to come up with a

network, and much of current research goes into investigating different network architectures. For a given task that the network should perform, the goal is to determine the optimal number of layers, number of neurons per layer, which layers to connect, etc. Occasionally, research focuses on explaining why a given architecture performs better than some other architecture, but the process of selecting an architecture remains largely a black box.

The field of *metalearning* is concerned with algorithmically determining a good network architecture for a given task. Recently several directions have been explored:

- [Zoph and Le 2016] train a neural network (the *controller* network) to construct a child network. The child network is consequently trained to perform the original task. The controller network is trained unsupervised by reinforcement learning. This approach achieved state of the art performance on predicting symbols of the Penn Treebank dataset.

- [Xie and Yuille 2017] use a genetic algorithm to optimise the architecture of a child convolutional network, used for an image classification task.

Both of these approaches produce intermediate child networks, which have to be fully trained to evaluate their performance. Because these methods have to train many child networks, they are expensive.

To mitigate the cost of fully training a child network, [Brock et al. 2017] note that the relative performance increase during early training can be used to estimate optimum performance. They train a network to provide initial weights for the child network, and rank the child networks after a single training batch. This allows exploring more of the vast space of network architectures.

To conclude, various approaches for learning network architecture have been proposed. In all of the approaches mentioned here, the process that optimises the network architecture is isolated from the learning process of the child network. In this thesis, we will instead optimise the network architecture and parameters simultaneously.

2.2 Sparse approximation and quantisation

State of the art networks tend to have a large memory footprint: they are deep, and have hundreds of millions of parameters. One way to reduce the memory footprint of such networks, is to approximate the weight matrices with something of lower rank. An other option is to quantise parameters, usually from 32-bit floating point numbers

down to a set of values that can be encoded in a few bits. As a side effect of quantisation, some parameters may become zero, effectively eliminating edges. If a sparse matrix representation is used for the weight matrices, edges can also be pruned directly. Many techniques to reduce the network size have been proposed.

- [Denton et al. 2014] replace convolution tensors in a convolutional neural network with sparse approximations that are sums of outer products. The approach is limited to convolutional nets, and takes a trained fully-connected network as input. The advantages of the sparse approximation are limited to the evaluation phase.

- [Han, Mao, and Dally 2015] prune a network to reduce the number of edges by 9 to 13 times. (They also quantise weights and compress the network further.) Their pruning procedure is to first train a fully-connected network, discard all edges for which the weight is below a certain threshold, and then train the remaining network again.

- [Hubara et al. 2016] go a step further and make weights binary, effectively deciding whether to keep or drop an edge. The main reason for doing this is to avoid the need for floating-point hardware. Quantisation is applied at every step during the training process; not as a post-processing step. However, apart from the benefits of being able to fit a bigger model in memory, sparsity is not exploited.

- [LeCun, Denker, and Solla 1990] prune edges using a more advanced method, dubbed "optimal brain damage". Rather than removing edges with the smallest absolute weight, they estimate the change in loss using the second derivative of the loss with respect to edge weight. This is a particular element of the Hessian of the loss with respect to the weights, which is assumed to be diagonal. The authors show that this procedure results in lower classification errors than simply deleting edges with smallest absolute weight, when used on a handwritten digit classification task. The number of parameters can be reduced by a factor 8 without significantly affecting the loss. [Han, Mao, and Dally 2015] do reference this work, but choose to prune based on absolute weight nonetheless.

- [Hassibi and Stork 1993] extend the optimal brain damage algorithm by dropping the assumption of a diagonal Hessian, which they discovered to be an unrealistic assumption. Their algorithm, dubbed "optimal brain surgeon", furthermore updates weights of the non-deleted edges to keep the loss low. The method assumes a mean squared error loss function, and it does require storing the entire Hessian, the size of which is quadratic in the number of parameters. At the time this might have been feasible — the authors mention networks of 10^4 parameters. However, for modern networks that can have as much as 10^8 parameters, computing or stor-

See [Zoph, Vasudevan, et al. 2017, Figure 5] for an overview of state of the art network sizes.

ing the entire Hessian no longer seems feasible. [Dong, Chen, and Pan 2017] note that the high cost can be somewhat mitigated by applying the optimal brain surgeon algorithm to one layer at a time, rather than to the entire network.

Although all approaches mentioned above produce a sparse network from a dense network, this is done as a final step. Sparsity is not integrated into the learning process — perhaps apart from [Hubara et al. 2016] in a limited way — and consequently the major benefits are in the evaluation phase only. In this thesis, we will instead optimise the architecture of a sparse network simultaneously with learning its parameters.

2.3 Constrained optimisation and regularisation

To limit the number of edges in a network, one approach is to enforce a sparsity constraint on a fully-connected network. We can view this approach as minimising a loss function not over $\mathbb{R}^n$ (where n is the number of parameters of the network), but over a subset $X \subseteq \mathbb{R}^n$, called the *feasible set*.

Gradient descent can be adapted to this constrained setting: when the descent step produces new parameters $w \in \mathbb{R}^n$, replace them with the nearest point $w' \in X$ (for a given metric on $\mathbb{R}^n$). This procedure is called *projected gradient descent* or *iterative hard thresholding*: the parameters are projected onto the feasible set. Without additional assumptions on f or X, projected gradient descent may get stuck in local minima.

In order for a network to be sparsely connected, many of the edge weights should be zero, to eliminate the edge. To be able to describe the feasible sets for such parameter vectors, we introduce the notion of *s-sparsity*:

Definition 2.1 · A vector $v = (v_1, v_2, \ldots, v_n) \in \mathbb{R}^n$ is called *s-sparse* if $v_i \neq 0$ for at most s indices $i \in \{1, \ldots, n\}$. Note that this definition depends on the choice of basis.

Under this definition, the set of 1-sparse vectors in $\mathbb{R}^n$ is the union of all axes. In general, the set of s-sparse vectors in $\mathbb{R}^n$ is the union of all s-dimensional hyperplanes spanned by s basis vectors. This set is not convex if $s < n$.

Projected gradient descent can still work well under certain assumptions, in particular when the feasible set X is convex. Unfortunately, sparsity constraints lead to feasible sets that are not convex, as shown above. [Jain, Tewari, and Kar 2014] analyse convergence of projected gradient descent in the high-dimensional case (where n is large), where the feasible set is the set of s-sparse vectors, for some $s < n$. The projection step in that case sets the $n - s$ smallest components of the parameter vector $w \in \mathbb{R}^n$ to zero, and this

works reasonably well in the high-dimensional case.

A slightly different approach, without hard constraints on the parameters, is to minimise the sum of a differentiable loss function $f : \mathbb{R}^n \to \mathbb{R}$ and a non-differentiable regularisation penalty $g : \mathbb{R}^n \to \mathbb{R}$:

$$\min_{w \in \mathbb{R}^n} f(w) + g(w)$$

Note that this problem is closely related to the previous problem, because we can choose g to prefer solutions close to the feasible set X. For instance, we might take $g(w) = d(w, X)$, the distance to the feasible set.

Gradient descent can be adapted to deal with the regularisation penalty in several ways, collectively called *proximal gradient descent* methods. One particular example of this problem is ℓ_1-regularisation, where $g(w) = \lambda \|w\|_1$. [Tibshirani 1996] proposed the *lasso* (least absolute shrinkage and selection operator) algorithm to solve this problem for particular classes of f, and noted that it tends to produce solutions where many of the parameters are zero. In [Tibshirani 2011, p. 274], the author hints at applications in graph selection. The sparsity can be controlled with λ, but it is not fixed up front. To achieve a desired sparsity, we have to tune λ.

In [Mazumder, Friedman, and Hastie 2011], the authors approach the problem of fitting a linear model with sparse coefficients. They present an optimisation algorithm based on *coordinate descent*, that can deal with non-convex regularisation penalties. In coordinate descent, rather than updating the entire parameter vector w at every step, only one coordinate w_i is updated per step. The idea of their algorithm is to parametrise the regularisation loss g with some $\gamma \in \mathbb{R}$, where for large values of γ, $g_\gamma : \mathbb{R}^n \to \mathbb{R}$ is convex, and as γ goes to 0, g_γ goes to the actual desired penalty function, which might not be convex. The algorithm then iteratively optimises w for a given value of γ, and uses the optimum as a starting point for optimising with a smaller γ in the next iteration.

The techniques in this section train a fully-connected network, in which some of the parameters happen to be zero. Sparse approximation algorithms can explore the full space of possible network architectures, but this means that the feasible problem size is limited. Training a sparse network using these techniques generally is as expensive as training a fully-connected network.

2.4 Unsupervised learning

So far, we have discussed supervised learning methods, where a model is made to fit a training set of labelled input-output pairs. In unsupervised learning, such a training set

is not available. We cannot use the loss over a training set to quantify performance of a model. Unsupervised methods optimise a different metric, that depends only on the input data.

One particular example of unsupervised learning is *autoencoding*. Autoencoders are used to encode input data into a format that is more suitable for subsequent processing, by removing redundancies or noise. Autoencoders can be defined in various ways. The following definition is very general:

Definition 2.2 · Let X be a metric space. An *autoencoder* consists of two functions, the encoder $E : X \to Y$, and the decoder $D : Y \to X$, such that $D \circ E$ is close to the identity, in the sense that for all $x \in X$, the *reconstruction error* $d(x, D(E(x)))$ is small.

The meaning of "small" depends on the kind of autoencoder. One choice is to limit E and D to a particular class of functions, and to minimise $\sum_{x \in X} d(x, D(E(x)))$. If we have a probability distribution over X, we might instead minimise $\mathbb{E}[d(x, D(E(x))]$ where x is drawn from X according to the given distribution.

Autoencoders can be used for dimensionality reduction, by taking $X = \mathbb{R}^n$ and $Y = \mathbb{R}^m$ for $m < n$. The hope is that the encoder removes redundancies and noise in the input data, such that coordinates in Y have a more meaningful interpretation. Autoencoders are often used as part of a larger network, but they can be trained separately: unlike in supervised learning, no training set of labelled input-output pairs is needed. The features in Y are entirely emergent.

In [Aroa et al. 2013], the authors give an unsupervised learning algorithm that learns a deep autoencoder network. The learning algorithm constructs a network layer by layer, by identifying correlations in the input data, and constructing a layer that removes these correlations. The edge weights are chosen from $\{-1, 1\}$, and the layers are *not* fully-connected. (The authors later extend the technique to real-valued weights.) Learning in this setting can be expressed as recovering a graph, and the authors use properties of random graphs to prove that the constructed network is a denoising autoencoder.

In the work by [Aroa et al. 2013], modifying edges of the network is an integral part of the training algorithm. However, the process cannot be guided: the training algorithm is unsupervised. While the resulting autoencoder can be used as part of a larger network, and the network architecture is learned from the input data, the technique cannot learn a network architecture to solve a particular given task (such as symbol prediction or image classification). In this thesis, we instead want to explore supervised approaches to learning network architecture.

2.5 Online topological ordering

In this thesis we will construct networks with general architectures, not necessarily layered like a traditional multilayer perceptron network. One challenge in such architectures is to determine the right order of computation. In a layered network, the layers can be evaluated one by one, starting from the input layer. Within layers no specific order is imposed, which makes evaluating such networks very suitable for massively parallel hardware. In a general acyclic graph, we need to topologically sort neurons to determine a computation order. This ensures that neuron outputs are known, before these outputs are used in subsequent computations. In a network that evolves by making small local changes to the graph, an interesting question arises: given the changes to make, can we efficiently update the neuron ordering?

The problem at hand is called *online topological ordering*. It can be formulated as follows: given directed acyclic graph with n vertices, a total ordering of the n vertices that is compatible with the graph structure, and a sequence of m edges to insert, produce a total ordering of the new graph. The related problem of *offline* topological ordering — given a directed acyclic graph with n vertices and m edges, produce a total ordering of the vertices — has been well-studied. An $O(n + m)$ solution has been known since the 1960s, and can be found in many algorithms textbooks. (See for example [Knuth 1997, § 2.2.3].) The online problem appears to be harder than the offline problem, with new discoveries having been published in the last decade.

- [Haeupler et al. 2011] introduce two algorithms for online topological ordering. The first algorithm, applicable to sparse graphs where $m \propto n$ rather than n^2, has a time complexity of $O(m^{3/2})$. The second algorithm can handle an arbitrary number of edge additions in $O(n^{5/2})$ time, but this is of no interest to us, as reordering the entire graph using an offline algorithm would be more efficient. Unlike the use case envisioned in the paper, we do not need to observe the intermediate graph after each of the m edge insertions. The algorithm for sparse graphs is based on a traversal of the vertices between the two endpoints of an inserted edge, in the vertex order.

- [Bender et al. 2011] give an $O(\min\{m^{1/2}, n^{2/3}\}m)$ algorithm for sparse graphs, and an $O(n^2 \log n)$ algorithm for dense graphs. Their sparse-graph algorithm assigns to every vertex a *level* and *index*, such that the combination of the two is lexicographically a topological ordering. The algorithm is based on a two-way search, forward and backward from the endpoints of an inserted edge. By showing how insertions affect the level and index of vertices, the authors are able to show that their algorithm is efficient. This work was reprinted in 2016.

In short, we do not need to topologically sort a graph after every modification. For sparse graphs, inserting m edges can be done in $O(m^{3/2})$ time. We can maintain a topological ordering of the graph as the graph is being updated. This means that making small modifications to the graph during the learning process can be done efficiently.

2.6 Maximal inner product and nearest neighbour search

To select candidate edges to add to the network graph later in this thesis, we will need to do a *maximal inner product search* (abbreviated MIPS): given a set of vectors, find the vector that has the maximal inner product with a query vector. A related problem is the *nearest neighbour search* based on cosine similarity. When all vectors in the search set have the same norm, the two problems can be shown to be equivalent. With the rise of recommendation systems and the revived interest in data mining recently, both maximal inner product search and nearest neighbour search have been studied extensively.

- [Friedman, Bentley, and Finkel 1977] give an overview of early solutions to the nearest neighbour problem, culminating in the introduction of the k-d tree in [Bentley 1975]. The authors then go on to present an optimised algorighm to construct and query the k-d tree for a nearest neighbour search. The algorithm traverses the k-d tree as usual to find the leaf node that would contain the query point, and selects the closest point from that leaf. The algorithm then backtracks, from the leaf back to the root. If, at a split plane, there could exist points on the unvisited side that are closer to the query point than the current closest point, that side needs to be traversed as well. Construction of the k-d tree takes $O(kn \log n)$ time for n points. The best-case query time is $O(\log n)$, but in the worst case, backtracking needs to visit all nodes, causing a worst-case query time of $O(n)$. For nearest neighbour search, k-d trees tend to work well in low dimensions, but performance degrades for high k.

- [Meiser 1988] gives an $O(k^5 \log n)$ algorithm for point location query in an arrangement of n hyperplanes. The hyperplanes partition $\mathbb{R}^k$ into regions, and the goal of a point location query is to find the region that contains the query point. The main contribution of this work, is that it demonstrated that point location queries need not be exponential in k. This work was later reprinted as [Meiser 1993].

- [Panigrahy 2008] proposes to traverse a k-d tree multiple times with a small random pertubation applied to the query point each time. This yields an approximate nearest neighbour of the query point, without the cost of backtracking.

- [Clarkson 1999] introduces two data structures with corresponding query algorithms for nearest neighbour search. One data structure follows a divide and conquer approach, the other is based on skip lists. Both data structures appear efficient, with a construction time proportional to $n \log n$ and query time proportional to $\log n$. Unfortunately the bounds also include a factor that is exponential in the dimension k. Although interesting, these data structures do not cure the curse of dimensionality that plagued the k-d tree.

- [Indyk and Motwani 1988] introduce *locality-sensitive hashing* (abbreviated LSH) as a solution to the approximate nearest neighbour search problem. Locality-sensitive hashing addresses the curse of dimensionality using a familiy of hash functions that preserve locality. The hashes of points that are near *should* collide, so points that are near end up in the same bucket with high probability. The algorithm constructs $m \propto n^\rho$ hash tables, where $\rho \in (0, 1)$ is a constant determined by properties of the hash function family. Points that are closer than a certain threshold r collide in at least one of the m hash tables with high probability. To do a query, the algorithm traverses the bucket in which the query point would fall in all m hash tables one by one. When a bucket contains a point no further than r from the query point, it is returned. The algorithm may fail to return a point with some probability, but this probability can be made arbitrarily small at the cost of increasing m. The query time is $O(n^\rho \log n)$ in the number of distance computations. An updated version of this work was published as [Har-Peled, Indyk, and Motwani 2012].

- [Datar et al. 2004] give a concrete locality-sensitive hash function that works for all ℓ_p norms with $p \in (0, 2]$. Their hash function $h_{a,b} : \mathbb{R}^k \to \mathbb{Z}$ is to take

$$h_{a,b}(v) = \left\lfloor \frac{\langle a, v \rangle + b}{r} \right\rfloor$$

  for $a \in \mathbb{R}^k$ and $b, r \in \mathbb{R}$. The main contribution of the work is to show that for particular choices of a and b, $h_{a,b}$ is indeed a locality-sensitive hash function. [Li, Mitzenmacher, and Shrivastava 2014] later show that the term b is not necessary, and can in some cases even be harmful.

- Many incremental approvements have been made to to locality-sensitive hashing-based algorithms since they were first introduced. [Christiani 2017] gives a good overview.

The problems of maximal inner product search and nearest neighbour search, either exact or approximate, have been studied extensively. In this section we only scratched the surface of known results. Many more specialised results are known, for example nearest neighbour search algorithms in two dimensions based on Voronoi diagrams,

or general maximal inner product search algorithms that can be applied to search sets where vectors differ in norm. Such results are beyond the scope of this thesis. In this thesis we will only be searching through unit vectors in high dimensions, ranging from 10^1 to 10^3. The overview above focuses on this use case, and the history leading up to it. The key takeaway of this section is that it is possible to efficiently do an approximate nearest neighbour search in high dimensions.

CHAPTER 3

Networks

Many machine learning problems can be framed as the search for a certain function $f : \mathbb{R}^n \to \mathbb{R}^m$. In supervised learning this function f is unknown, but we do know $f(x)$ for a number of points $x \in \mathbb{R}^n$. We can use this knowledge to construct an approximation of f, and training a neural network is one way of doing this. In this chapter we will give one possible definition of "network", and we will show how the function that the network computes can be evaluated. The definitions given in this chapter are slightly more involved than those for a conventional multilayer perceptron network, in order to describe networks that are not necessarily layered.

3.1 Supervised learning

The goal in supervised learning is to construct an approximation of $f : \mathbb{R}^n \to \mathbb{R}^m$, given $f(x)$ for all $x \in T \subseteq \mathbb{R}^n$. T is called the *training set*. The goal of the learning process is to find the parameters for a function $g_w : \mathbb{R}^n \to \mathbb{R}^m$, parametrised by $w \in \mathbb{R}^k$ for some $k \in \mathbb{Z}_{>0}$, such that $g_w(x) \approx f(x)$ for all x in the training set. We will quantify this approximation in Section 3.2. The hope is that g_w will generalise — that $g_w(x) \approx f(x)$ even for $x \notin T$.

Example 3.1 · One instance of this problem is *linear regression*. In linear regression, g_w is a linear function, and the parameters are the matrix elements.

In linear regression the function g_w is simple, but a linear function might not be a good approximation of f. If we have prior knowledge about what kind of function f should be, we can use that knowledge to pick g. But as the problems we try to attack get more complicated, so does f. We need to consider different classes of functions. □

Artificial neural networks are one class of parametrised functions that are so flexible, that they can be made to approximate vitually any well-behaved function. Unlike poly-

nomials, neural networks can approximate a function well on a wide domain. Neural networks can compute bounded functions, whereas nonzero polynomials are unbounded. Moreover, algorithms are known that are able to effectively optimise the network parameters in practice. These two properties mean that we can "learn" a good approximation, even when we know little about f. Of course this does not mean that neural networks can magically mimick any black-box computation, but they have been succesfully applied to a wide variety of problems, including image classification, speech recognition, speech synthesis, facial recognition, translation between natural languages, image transcription, identification of cancer cells, playing board games, video games, and many more.

Example 3.2 · Perhaps the canonical problem where neural networks are applied, is classification of handwritten digits. The input here are luminance values of pixels in a scan of the digit, and the output is a vector of probabilities, one for each digit. The MNIST dataset consists of $7 \cdot 10^4$ labelled 28×28 greyscale scans of digits. In that case, we try to approximate $f : \mathbb{R}^n \to \mathbb{R}^m$ where $n = 28^2$ and $m = 10$. The MNIST dataset is often used to benchmark network architectures and optimisation algorithms, but is considered a small dataset by modern standards. Classification of handwritten digits has become a toy problem, as even relatively simple networks readily rival human performance on the task. [Cireşan, Meier, and Schmidhuber 2012] achieved a 0.23% test error rate with a network of $2.3 \cdot 10^5$ parameters, and a 0.39% test error rate using only $6.6 \cdot 10^3$ parameters. □

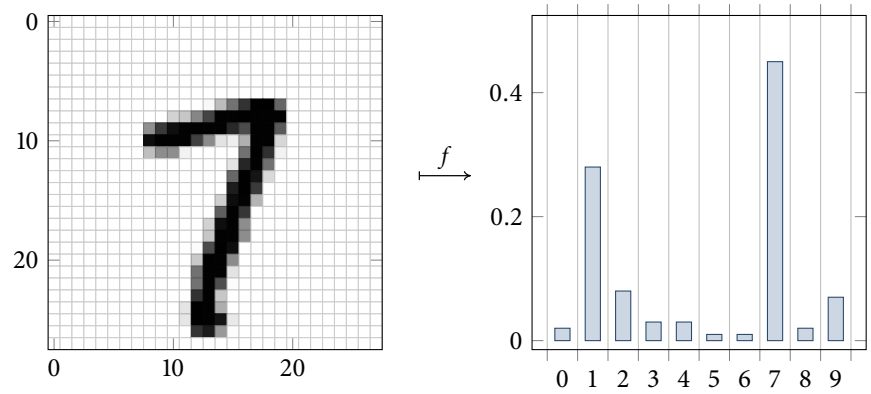


Figure 3.1 · The digit classification function f maps a vector in $\mathbb{R}^{28\times 28}$ to a vector in $\mathbb{R}^{10}$, by mapping pixel luminance to a probability distribution over digits.

Example 3.3 · Another problem where neural networks have been applied successfully, is natural language processing. One approach here is to convert a (prefix of) a sentence into a point in a vector space, such that sentences with a similar semantic meaning have a similar representation (natural language understanding). This representation can be used to e.g. predict the next word in the sentence, which is useful for compression, spell checking, autocorrection, etc. A commonly used dataset for language-related problems is the *Penn Treebank*, a collection of sentences taken from English newspapers, compiled

by the university of Pennsylvania. The sentences are annotated with trees that indicate their syntactic structure. Another standard dataset is *Enwik9*, the first 10^9 bytes of the English Wikipedia, compiled March 2006 by Matt Mahoney. □

3.2 Parameter optimisation

As we saw before, neural networks are a class of functions that are very flexible, and can be used to approximate virtually any well-behaved function $f : \mathbb{R}^n \to \mathbb{R}^m$. To quantify how good the approximation is, we need to introduce the notion of a *loss function*.

Definition 3.4 · Suppose the codomain of the function we wish to approximate is $\mathbb{R}^m$, and let $y, y' \in \mathbb{R}^m$. A *loss function* is a function $L : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}_{\geq 0}$, such that $L(y', y)$ is small when y' is a good approximation of y, and $L(y', y)$ is large when y' is a bad approximation of y. Metrics are often used as loss functions, but in general a loss function need not be symmetric, and $L(y, y) = 0$ need not hold for all y.

Example 3.5 · A loss function that was common in early machine learning research is the *squared error loss*:

$$L(y', y) = \|y - y'\|_2^2$$

This particular loss function is symmetric, and satisfies $L(y, y) = 0$ for all $y \in \mathbb{R}^m$. The derivative of the squared error loss with respect to y' is particularly simple, which makes it an attractive candidate when implementing a gradient-based optimisation method by hand, and for theoretical analysis. □

Example 3.6 · The *cross-entropy loss function* is appropriate when the output of the network is a probability distribution (all elements of y are in $[0, 1]$ and the elements of y sum to 1). It is defined as

$$L(y', y) = -\langle y, \log_2(y') \rangle$$

Here $\langle \cdot, \cdot \rangle$ denotes the standard inner product and $\log_2$ is applied elementwise. The cross-entropy indicates the expected number of bits required to identify one of m values drawn from the distribution y, when encoded in a scheme that is optimal for the distribution y'. One needs to be careful to avoid vectors y' that contain zero elements.

Unlike the squared error loss, the derivative of the cross-entropy loss does not vanish when y is close to y', when a softmax function is applied to the output layer (see Section 3.4). This makes the cross-entropy a popular loss function for classification problems. Natural language processing literature often uses a slightly different variation of this loss, the *set perplexity*. We will formulate set perplexity in Definition 3.9, after building up the loss over a batch. □

Now that we can quantify loss on a single sample, we can define the loss over a batch. To keep the equations manageable, we will assume that we have settled on a particular loss function L.

Definition 3.7 · For $f, g_w : \mathbb{R}^n \to \mathbb{R}^m$ and $x \in \mathbb{R}^n$, we can quantify how well g_w approximates f in x. Define the *loss of w in x* as

$$\begin{aligned} \mathcal{L}_{g,f} : \mathbb{R}^k \times \mathbb{R}^n &\longrightarrow \mathbb{R}_{\geq 0} \\ (w, x) &\longmapsto L(g_w(x), f(x)) \end{aligned}$$

When g and f are clear from the context, we will drop the subscript and write $\mathcal{L}(w, x)$.

Definition 3.8 · Let $T \subseteq \mathbb{R}^n$ be a set of points where $f(x)$ is known, and let f and g_w be as before. Define the *loss of w on T* as

$$\begin{aligned} \mathcal{L}_{T,g,f} : \mathbb{R}^k &\longrightarrow \mathbb{R}_{\geq 0} \\ w &\longmapsto \sum_{x \in T} \mathcal{L}_{g,f}(w, x) \end{aligned}$$

When g and f are clear from the context, we will drop them from the subscript and write $\mathcal{L}_T(w)$. Some authors average the loss over T, rather than summing it. This helps to make the loss comparable across sets of different cardinalities, but in this thesis we will omit the normalisation factor for clarity.

Related to the cross-entropy that we saw before is *set perplexity*, which is used often in natural language processing literature. In this case we briefly do need to introduce a normalisation factor.

Definition 3.9 · Let T, f, g_w be as before, and denote by $\mathcal{L}_T$ the cross-entropy loss on T. The *set perplexity* of g_w on T is given by

$$2^H \quad \text{where} \quad H = \frac{1}{|T|} \mathcal{L}_T(w)$$

Here $|T|$ denotes the cardinality of T. The set perplexity can be interpreted as the number of elements that g_w has to choose from.

Example 3.10 · Suppose we want to predict the second symbol of a pair based on the first symbol. First we need to embed symbols in a vector space. The free $\mathbb{R}$-vector space spanned by ♠, ♡, ♢, ♣ is isomorphic to $\mathbb{R}^4$. This is called a *one-hot encoding*. We might take

$$T = \{(♢, ♠), (♣, ♢), (♡, ♠), (♠, ♣)(♣, ♡)\}$$

For prediction, a naive g_w we can use is to ignore the first symbol, and always output the marginal probability distribution. In the basis $(♠, ♡, ♢, ♣)$ this yields $g_w(x) = (\frac{2}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5})$. With this we can compute the cross-entropy loss over T (which does not

depend on w in this case, for g does not):

$$\begin{aligned}
\mathcal{L}_T(w) &= \langle (1,0,0,0), -\log_2(\tfrac{2}{5},\tfrac{1}{5},\tfrac{1}{5},\tfrac{1}{5})\rangle + \langle (0,0,1,0), -\log_2(\tfrac{2}{5},\tfrac{1}{5},\tfrac{1}{5},\tfrac{1}{5})\rangle \\
&\quad + \langle (1,0,0,0), -\log_2(\tfrac{2}{5},\tfrac{1}{5},\tfrac{1}{5},\tfrac{1}{5})\rangle + \langle (0,0,0,1), -\log_2(\tfrac{2}{5},\tfrac{1}{5},\tfrac{1}{5},\tfrac{1}{5})\rangle \\
&\quad + \langle (0,1,0,0), -\log_2(\tfrac{2}{5},\tfrac{1}{5},\tfrac{1}{5},\tfrac{1}{5})\rangle \\
&= -\log_2(\tfrac{2}{5}) - \log_2(\tfrac{1}{5}) - \log_2(\tfrac{2}{5}) - \log_2(\tfrac{1}{5}) - \log_2(\tfrac{1}{5}) \\
&= 5\log_2(5) - 2\log_2(2) - 3\log_2(1) \\
&\approx 9.61
\end{aligned}$$

The set perplexity on T is given by $2^H \approx 3.79$, for $H = \frac{1}{5}\mathcal{L}_T(w) = \log_2(5) - \frac{2}{5}$. We find that our g_w performs better than random guessing (predicting a uniform distribution): the set perplexity of 3.79 is less than the 4 symbols in the set. □

Optimisation

The loss gives us a target for optimisation of the function parameters: the parameters for which g best approximates f on T are

$$w^\star = \arg\min_{w \in \mathbb{R}^k} \mathcal{L}_T(w)$$

Many optimisation algorithms exists to find $w^\star$, or at least a local minimum. The most common approaches are based on gradient descent, which assumes that $\mathcal{L}_T$ is differentiable.

- In traditional gradient descent, we iteratively improve the parameter vector $w^{(t)}$ at iteration $t \in \mathbb{Z}_{>0}$ as

  $$w^{(t+1)} = w^{(t)} - \frac{\alpha}{\sqrt{t}} \nabla \mathcal{L}_T(w^{(t)})$$

  for some step size $\alpha \in \mathbb{R}_{>0}$. Note that $w^{(t)}$ is not guaranteed to converge if we keep the step size α fixed for every update, hence the factor $t^{-1/2}$.

- *Stochastic gradient descent* evaluates the gradient on a random subset $U \subseteq T$, such that $\nabla \mathcal{L}_U$ is an unbiased estimate of $\nabla \mathcal{L}_T$.

- Many more advanced optimization algorithms operate on the same principle, including momentum-based methods such as Adam, introduced in [Kingma and Ba 2014].

- Second-order methods incorporate also the Hessian of $\mathcal{L}_T$. These methods are not feasible for optimising large networks, due to the sheer size of the Hessian.

Many more specialised optimisation algorithms exist. Note that often, achieving the lowest possible loss on a given set T is not the goal, due to overfitting. The role of the

optimisation algorithm is not to find a local minimum of the loss, but rather to find parameters for which the loss is low enough. By imposing extra constraints on $\mathcal{L}_T$, optimisation algorithms with better convergence properties can be constructed — or algorithms for which a convergence proof is known at all. For many optimisation algorithms, convergence to a local minimum has only been proven under assumptions that do not hold in the situations where these algorithms are typically used. For instance, most optimisation algorithms assume that $\mathcal{L}_T$ is convex and has a continuous derivative, and both of these assumptions are routinely violated. (See for instance Corollary 3.26 and Example 3.19.) In practice, optimisers reach parameters for which the loss is small enough nonetheless. Understanding why they do, is an area of active research. The field of optimisation is vast, and beyond the scope of this thesis.

3.3 The space of networks

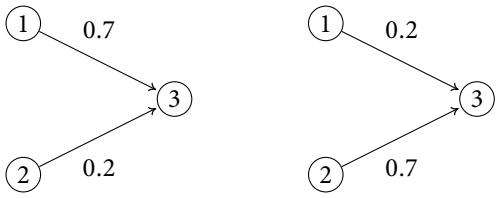
Later in this thesis we will rewire edges in a network, and for this it is useful to consider the set of all networks as a metric space. Adding and deleting edges in a directed graph may sound like discrete operations: an edge is either present or not. But because a neural network has *weighted* edges, we *can* partially delete an edge. A network with an edge that is present with weight zero, is equivalent to the same network where the edge is absent: the network computes the same function in both cases. Therefore we may consider a network of $k \in \mathbb{Z}_{>0}$ vertices to be a fully-connected graph, where some of the edges have weight zero. Because the network is defined by its edge weights, the network is a point in a space.

Definition 3.11 · Let $k \in \mathbb{Z}_{>0}$. A *weighted directed graph* with k vertices, is a function $w : [1, \ldots, k]^2 \to \mathbb{R}$, that assigns to every pair of vertices the weight of the directed edge between them. We say an edge between vertex i and j is *present* if $w(i, j) \neq 0$, and *absent* if $w(i, j) = 0$. Denote by $\mathrm{Digraph}(k)$ the set of all weighted directed graphs with k vertices.

Remark 3.12 · Note that under the above definition, a weighted directed graph has the following properties:

- There exists at most one edge between a pair of vertices. For the graphs that correspond to neural networks this is not a limitation, as we will see later. A network with multiple edges between vertices computes the same function as a network with the edges combined into a single edge by summing the weights.

- Vertices are labelled. The following two graphs are not equal:



Corollary 3.13 · For every $k \in \mathbb{Z}_{>0}$, there exists a bijection

$$\begin{aligned} \mathrm{Digraph}(k) &\longrightarrow \mathrm{Mat}(k \times k, \mathbb{R}) \\ w &\longmapsto \big(w(i,j)\big)_{ij} \end{aligned}$$

The matrix that a graph maps to, plays a role similar to that of the adjacency matrix. Because $\mathrm{Mat}(k \times k, \mathbb{R})$ is a metric space by taking the Euclidean distance on $\mathbb{R}^{k \times k}$, it follows that $\mathrm{Digraph}(k)$ is a metric space. □

Not every weighted directed graph corresponds to a neural network: the graph of a neural network is acyclic. (Recurrent neural networks are sometimes called cyclic in literature, but this is a misleading use of terminology. For the purpose of evaluation, such networks are not cyclic.)

Definition 3.14 · Let $k \in \mathbb{Z}_{>0}$. A *network* with k neurons is an acyclic weighted directed graph $w : [1, \ldots, k]^2 \to \mathbb{R}$ together with a function $b : [1, \ldots, k] \to \mathbb{R}$. The vertices of a network are also called *neurons*, and the function b gives the *bias* at every neuron. Denote by $\mathrm{Net}(k)$ the set of networks with k neurons.

As with weighted direct graphs, we can map networks to matrices:

$$\begin{aligned} &\mathrm{Net}(k) \longrightarrow \mathrm{Mat}(k \times k, \mathbb{R}) \\ &(w, b) \longmapsto \big(x_{ij}\big)_{ij} \quad \text{where} \quad x_{ij} = \begin{cases} w(i,j) & \text{if } i \neq j \\ b(i) & \text{if } i = j \end{cases} \end{aligned}$$

This function is a bijection onto its image, hence $\mathrm{Net}(k)$ is a metric space. The zero matrix is in the image, and it corresponds to the network where all edges are absent, and the bias associated with every neuron is zero. We put the biases on the diagonal in $\mathrm{Mat}(k \times k, \mathbb{R})$ for ease of notation. This is possible, because $w(i,i) = 0$ for all i, so no information is lost.

Definition 3.15 · A *layered* network with z layers is a network where the neurons can be partitioned into z sets, $A_1, \ldots, A_z$, such that for every present edge $v_1 \to v_2$ it holds that

$$v_1 \in A_i \implies v_2 \in A_{i+1}$$

In other words, edges only exist between consecutive layers. See also Figure 3.2.

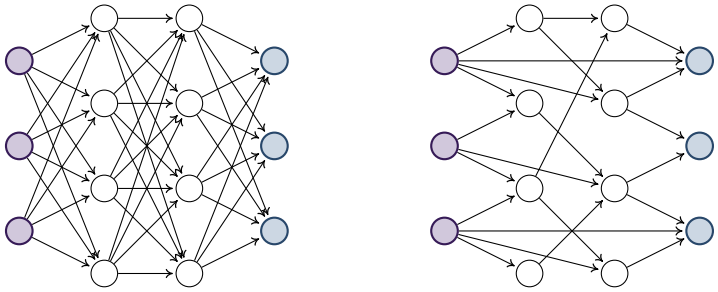


Figure 3.2 · Left: a fully-connected, layered network with 4 layers. Right: a non-layered network. Some of the neurons in the non-layered network can be reached from an input neuron (coloured •) with paths of different lengths.

Deep networks

In recent years, many advances in machine intelligence have been attributed to so-called "deep" neural networks. In a traditional multilayer perceptron network, the depth of the network is equal to its number of layers. There is no clear threshold for what constitutes a "deep" network; many authors consider a network with more than three layers to be deep already. In our case, where networks are not necessarily layered we can define a notion of depth in terms of paths through the graph.

Definition 3.16 · Let $(w, b) \in \mathrm{Net}(k)$ be a network. The *depth* of (w, b) is the maximum length d of a sequence $(v_1, \ldots, v_d)$ of distinct vertices, such that $\forall i \in [1, \ldots, d-1] : w(v_i, v_{i+1}) \neq 0$. In other words, the depth is the longest path along present (nonzero-weighted) edges of the graph.

Treating networks as point in a metric space is a powerful technique. It gives us a way to quantify similarity between networks, and optimisation algorithms can be understood as constructing a path through the space. And as we will see later, networks that are near in space, compute similar functions. In this framework, learning network parameters, and learning network architecture, are the same thing.

3.4 Networks as functions

Now that we have defined networks as a particular kind of weighted acyclic graph, we can define the function associated with a network. Before we can explain how networks compute though, we need to introduce *activation functions*. Activation functions give networks their power: without a nonlinear activation function, a network of any depth would reduce to a two-layer network, and the function it computes would be a linear map.

Definition 3.17 · An *activation function* can be any continuous function $\sigma : \mathbb{R} \to \mathbb{R}$,

although for most purposes (such as optimisation of the network parameters) stronger guarantees such as differentiability are required. Activation functions that are used in practice are typically monotonic.

Example 3.18 · One activation function that was popular in early machine learning research, is the *sigmoid activation function*, given by

$$x \longmapsto \frac{1}{1+\exp(-x)}$$

It has fallen out of favour because its derivative has the undesirable property that it goes to zero as $x \to \pm\infty$. □

Example 3.19 · An activation function that is commonly used in modern machine learning applications, is the function

$$x \longmapsto \max\{0, x\}$$

Many authors refer to this function as a *rectified linear unit*, or "ReLU" activation function. Its derivative does not vanish as $x \to \infty$, but the function is not differentiable at 0. Moreover, if due to an unfortunate combination of parameters the input to this function is always negative, then the gradient of the parameters will always be zero, and the situation will persist. Neurons with such parameters are called *dead neurons*. □

Given an activation function, we can now define the function that a network computes.

Definition 3.20 · Let $n, m, k \in \mathbb{Z}_{>0}$ be given, such that $k \geq n$ and $k \geq m$. Let (w, b) be a network with k neurons, and σ an activation function. We will define the *evaluation function* $f_{w,b,\sigma} : \mathbb{R}^n \to \mathbb{R}^m$ computed by this network, by constructing the image of $x \in \mathbb{R}^n$. Associate with every neuron $i \in [1, \ldots, k]$ its *activation* $a_i \in \mathbb{R}$:

$$a_i = b(i) + \left[\sum_j w(j,i)\sigma(a_j)\right] + \begin{cases} x_i & \text{if } i \leq n \\ 0 & \text{otherwise} \end{cases}$$

Note that because the network is an acyclic graph, the activations can be computed directly. If a_j occurs with nonzero weight in the expression for a_i, then a_j does not depend on a_i. Finally, define

$$f_{w,b,\sigma}(x) = (a_{k-m+1},\ a_{k-m+2},\ \ldots,\ a_k)$$

When the activation function σ is the sigmoid function, the value a_i is also called the *logit*. Some authors use this term even in combination with different activation functions. The value $\sigma(a_i)$ is called the *output* of neuron i. Neuron $1, \ldots, n$ are called *input neurons* and neuron $k - m + 1, \ldots, k$ are called *output neurons*.

For simplicity, we will only consider networks without edges between input neurons, and no input biases, i.e. $w(i,j) = 0$ and $b(i) = 0$ for all $i, j \in [1, \ldots, n]$. In that case, we have $a_i = x_i$ for $i \in [1, \ldots, n]$. Furthermore, we assume that $k \geq n + m$, so input and output neurons are disjoint. After all, an output neuron that is also an input is not useful.

Note that the output of the network consists of output neuron activations, without the activation function σ applied. The elements of the output vector are simply weighted sums of neuron outputs and biases. Often one more function is applied to the output vector. In classification problems for example, the output should be a probability distribution over all classes, so the elements of the output vector should sum to one, which is generally not the case for $f_{w,b,\sigma}(x)$. One trick to turn the output vector into a probability distribution, is to use a construction akin to the Boltzmann distribution in statistical physics. We take the probability p_i of class i to be proportional to $\exp(a_i)$ by setting

$$p_i = \frac{\exp(a_i)}{\sum_{j=k-m+1}^{k} \exp(a_j)}$$

The function $\mathbb{R}^m \to [0,1]^m$ that turns an output vector into a probability distribution as above, is often called the *softmax function* in machine learning literature. For the purpose of this thesis, such a function is not important, because it has no learnable parameters. The function can be folded into the loss function for the purpose of optimising the network.

Multilayer perceptrons

The definition of network in this chapter is deliberately very general. Traditionally, neurons in a network have been organised into layers, where consecutive layers form a complete bipartite graph. In this case, (w, b) can be represented by a weight matrix and bias vector for every layer, and the output of the network can be computed with a few matrix multiplications. For every layer z, we can define

$$\begin{aligned} f_z : \mathbb{R}^{n_z} &\longrightarrow \mathbb{R}^{n_{z+1}} \\ x &\longmapsto b_z + W_z \sigma(x) \end{aligned}$$

and the evaluation function

$$f = f_{Z-1} \circ f_{Z-2} \circ \cdots \circ f_2 \circ f_1$$

Here $Z \in \mathbb{Z}_{>1}$ denotes the number of layers, $n_z \in \mathbb{Z}_{>0}$ is the number of neurons at layer z, and for every $z \in [1, \ldots, Z-1]$ we have a weight matrix $W_z \in \mathrm{Mat}(n_{z+1} \times n_z, \mathbb{R})$ and a bias vector $b_z \in \mathbb{R}^{n_{z+1}}$. The activation function σ is applied elementwise. One challenge of this approach is that Z and the layer sizes n_z have to be chosen up front, and there is very little theory on how to guide this choice. Even for a given budget of neurons or edges, it is not clear how to distribute these over layers. In this thesis we will try to automate

this process by incrementally improving the graph, which *may* result in a layered graph, where consecutive layers form complete bipartite graphs, although these graphs are only a very small subset of all possible networks. While networks as defined in this chapter do encompass traditional multilayer perceptron networks, they can also express more general network architectures.

The definition of network given in this chapter is very general in the structure of the underlying graph. It is however very regular with respect to the neurons. Networks that are used in modern machine learning applications are often less regular, and composed of more elaborate building blocks. For instance, convolutional networks often include *max-pooling* layers, where instead of taking a weighted sum of incoming activation, neurons take the maximum over incoming activation. Moreover, constructions such as LSTMs combine multiple vectors with elementwise operations at every *cell* (a building block similar to a layer of neurons). Some networks have parameters other than edge weights and biases. In most cases though, at least some part of the network satisfies the definition given in this chapter, and the techniques in this thesis still apply to subgraphs. If a network includes more elaborate building blocks, we will only consider the part of the network with traditional, homogeneous neurons.

3.5 Symmetries

In the previous section we introduced the space of networks as a subspace of $\mathbb{R}^{k\times k}$, but not every point in this space corresponds to a distinct evaluation function. In the end we are interested in evaluation functions, and not necessarily their parameters. Therefore we should look at $\mathrm{Net}(k)$ modulo symmetries that do not affect the evaluation function. In this section we will briefly examine a few symmetries, and their effect on optimisation.

Definition 3.21 · Let $(w_1, b_1), (w_2, b_2) \in \mathrm{Net}(k)$ be networks with n inputs and m outputs, and let σ be an activation function. We say (w_1, b_1) and (w_2, b_2) are *equivalent*, written $(w_1, b_1) \sim (w_2, b_2)$, if they have same evaluation function. That is, if

$$\forall x \in \mathbb{R}^n : f_{w_1,b_1,\sigma}(x) = f_{w_2,b_2,\sigma}(x)$$

One symmetry under which evaluation functions are invariant, is permuting inner neurons. Recall from Remark 3.12 that neurons in a network are labelled; they are distinguished by index. For input and output neurons this is important, because we attach meaning to the different coordinates. But as far as the evaluation function is concerned, only the structure of the graph (including edge weights) matters, and we can assign labels freely to the internal neurons. In other words, we can permute the labels of internal neurons without changing the evaluation function.

Lemma 3.22 · The symmetric group on k elements, S_k, acts on $\mathrm{Net}(k)$ in the category of vector spaces.

Proof: Let $\tau \in S_k$ be a permutation of k elements. Then we can define

$$\begin{aligned} \tau^* : \mathrm{Net}(k) &\longrightarrow \mathrm{Net}(k) \\ (w, b) &\longmapsto (w \circ (\tau, \tau), b \circ \tau) \end{aligned}$$

Here (τ, τ) denotes τ applied elementwise to both coordinates. To show that τ^* is a linear automorphism, we may show that it is a linear map with kernel 0. Because τ^* is a permutation of coordinates, it is a linear map. Furthermore, because composing with τ only permutes coordinates,

$$(w \circ (\tau, \tau), b \circ \tau) = ((i, j) \mapsto 0, i \mapsto 0) \implies (w, b) = ((i, j) \mapsto 0, i \mapsto 0)$$

hence the kernel of τ^* is 0. We have $(\tau^*)^{-1} = (\tau^{-1})^*$. □

Proposition 3.23 · Let $(w, b) \in \mathrm{Net}(k)$ be a network that has n inputs and m outputs. Let $\tau \in S_k$ be a permutation of k elements that is the identity on $[1, \ldots, n]$ and $[k-m+1, \ldots, k]$. Then $\tau^*((w, b)) \sim (w, b)$.

Proof: Because τ can be written as a product of transpositions, it is sufficient to prove the claim for a transposition ρ. Suppose that ρ transposes neuron i and j. Recall from Definition 3.20 that the neuron activations a_i and a_j are weighted sums over the outputs of incoming neurons, and a bias. By swapping incoming edges and the biases of neuron i and j, the transposition relabels a_i such that a_i has the value of a_j before the transposition, and vice versa. Because outgoing edges are also swapped, the activations of neurons that have neuron i or j as input, remain unchanged (unless the neuron is i or j itself). This holds even when neuron i is an input to j or when j is an input to i. This shows that $\rho^*((w, b)) \sim (w, b)$. By repeatedly applying this result for the transpositions that τ is a product of, we find $\tau^*((w, b)) \sim (w, b)$. □

Networks that use the "ReLU" activation function have another symmetry. We can rescale the output of a neuron, by scaling all its inputs and its bias by a factor $\lambda \in \mathbb{R}_{>0}$. If the neuron is not an output neuron, we can compensate for the change by scaling the weights of outgoing edges by a factor λ^{-1}. Unlike a network with, say, a sigmoid activation function, a network with "ReLU" activation has no inherent scale. We can rescale all weights freely — apart from those feeding into output neurons.

Proposition 3.24 · Let $(w, b) \in \mathrm{Net}(k)$ be a network in that has n inputs and m outputs. Let $\lambda \in \mathbb{R}_{>0}$ and $p \in [1, \ldots, k - m]$. Define (w', b') as follows:

$$w'(i,j) = \begin{cases} \lambda w(i,j) & \text{if } j = p \\ \lambda^{-1} w(i,j) & \text{if } i = p \\ w(i,j) & \text{otherwise} \end{cases}$$

$$b'(i) = \begin{cases} \lambda b(i) & \text{if } i = p \\ b(i) & \text{otherwise} \end{cases}$$

Note that $w'(p,p)$ is well-defined because $w(p,p) = 0$. Then, if $\sigma(x) = \max\{0, x\}$ is used as activation function, it holds that $(w', b') \sim (w, b)$.

Proof: Recall from Definition 3.20 that the activation of neuron p is given by

$$a_p = b(p) + \sum_j w(j,p)\sigma(a_j)$$

If we replace (w, b) with (w', b'), then a_p would gain a factor λ. For neurons that depend on neuron p, the factor is cancelled by λ^{-1} in the weights, because $\sigma(\lambda a_p) = \lambda\sigma(a_p)$. As a result, the activation of internal neuron a_p changes with a factor λ, but the activations of output neurons remain unchanged, hence $(w', b') \sim (w, b)$. □

The above proposition sheds some light on what the "loss landscape" of a network looks like. Any network in $\mathrm{Net}(k)$ with "ReLU" activation function and at least one internal neuron is a point on a branch of a hyperbola of constant loss, with one such hyperbola for every internal neuron. Every internal neuron *almost* removes one dimension from the parameter space, because we can choose the weight of its first incoming edge to be in $\{\pm 1\}$. In particular, this means that local minima of the loss are not isolated points in the parameter space, but hypersurfaces.

In the end, we are interested in an evaluation function, and not necessarily its parameters. Therefore, we should really consider $\mathrm{Net}(k)$ modulo the symmetries mentioned above (and possibly more). However, the quotient space $\mathrm{Net}(k)/\sim$ is not easily parametrised, and optimisation algorithms rely on $\mathrm{Net}(k)$ being a vector space. Generally, we simply disregard the symmetries. After all, once we have *some* parameters (w, b) that have a good evaluation function, we have a good evaluation function. Still, these symmetries have implications for optimisation. Symmetries of $\mathrm{Net}(k)$ can shed light on the shape of the loss function $\mathcal{L}_T$.

Theorem 3.25 · Let $(w, b)^\star \in \mathrm{Net}(k)$ be a local minimum of the loss $\mathcal{L}_T$ on some $T \subseteq \mathbb{R}^n$. For the sake of readability we will write $wb^\star$ rather than $(w, b)^\star$ in this theorem. Say the network has $z = k - m - n$ inner neurons, and suppose that $\mathcal{L}_T$ is continuous and not constant. Let $U \subseteq S_k$ be the subgroup of S_k that is the identity on $[1, \ldots, n]$ and $[k - m + 1, \ldots, k]$. Then the elements of the orbit of $wb^\star$ under U,

$$U \cdot wb^\star = \{\tau^*(wb^\star) \mid \tau \in U\}$$

are all local minima of $\mathcal{L}_T$.

Proof: Let $Q \subseteq \mathrm{Net}(k)$ be an open neighbourhood of $wb^\star$, such that for all $wb \in Q$, $\mathcal{L}_T(wb) \leq \mathcal{L}_T(wb^\star)$. Let $\tau \in U$. From Lemma 3.22, we know that τ^* is a linear automorphism, and therefore it is a continuous open map. It follows that $\tau^*(Q)$ is an open neighbourhood of $\tau^*(wb^\star)$. From Proposition 3.23, it follows that

$$\forall wb \in Q : \mathcal{L}_T(\tau^*(wb)) = \mathcal{L}_T(wb) \geq \mathcal{L}_T(wb^\star) = \mathcal{L}_T(\tau^*(wb^\star))$$

Hence $\tau^*(wb^\star)$ is a local minimum in $\tau^*(Q)$. □

Corollary 3.26 · In general, $U \cdot wb^\star$ is not a singleton, hence $\mathcal{L}_T$ is not strictly convex, because strictly convex functions have a single global minimum. Moreover, if $\mathcal{L}_T$ is strictly convex on an open neighbourhood of $wb^\star$, then $\mathcal{L}_T$ is not convex on its entire domain. □

Theorem 3.27 · The multiplicative group $\mathbb{R}^k_{>0}$ acts on $\mathrm{Net}(k)$ in the category of topological spaces by applying the construction from Proposition 3.24 to each of the k neurons. Let $(w, b) \in \mathrm{Net}(k)$ be a network with n inputs, m outputs, and $q = k - n - m$ internal neurons. $\mathbb{R}^q_{>0}$ is a subgroup of $\mathbb{R}^k_{>0}$ by embedding it as follows:

$$\begin{aligned} \mathbb{R}^q_{>0} &\longrightarrow \mathbb{R}^k_{>0} \\ (\lambda_1, \ldots, \lambda_q) &\longmapsto (\underbrace{1, \ldots, 1}_{n}, \lambda_1, \ldots, \lambda_q, \underbrace{1, \ldots, 1}_{m}) \end{aligned}$$

Then the elements of the orbit of (w, b) under $\mathbb{R}^q_{>0}$ are all equivalent, and hence the quotient map $\mathrm{Net}(k) \twoheadrightarrow \mathrm{Net}(k)/\!\sim$ factors as $\mathrm{Net}(k) \twoheadrightarrow \mathrm{Net}(k)/\mathbb{R}^q_{>0} \twoheadrightarrow \mathrm{Net}(k)/\!\sim$.

Proof: Equivalence of the elements of the orbit follows from repeatedly applying Proposition 3.24. Factorisation of the quotient map then follows from the universal property of the quotient topology. □

The theorems in this section give a theoretical justification for a few of the empirical observations in deep learning. In particular, we have a partial explanation of the many local minima of the loss function. There have been a few attempts to formally explain various other empirical observations in literature. One notable result is the work by [Choromanska et al. 2014], which further characterises the "loss landscape", and claims that for sufficiently deep networks, the loss at all local minima is close to the globally minimal loss. Although interesting, the formalism is beyond the scope of this thesis.

Without knowing a lot about a network, or the particular loss function, the theorems in this section shed some light on the nature of $\mathcal{L}_T$. In particular, it is not strongly convex, and for networks with a "ReLU" activation function, local minima are surfaces rather than single points. It might be possible for optimisers to take advantage of symmetries to reduce the size of the parameter space to explore, but this appears to be an area that is

unexplored in the literature. In fact, some literature suggests that overparametrisation is essential for effective optimisation. The main takeaway for this thesis, is that one particular evaluation function can be represented by many different points in $\mathrm{Net}(k)$.

CHAPTER 4

Rewiring

To unify optimisation of the network parameters and optimisation of the network architecture in one process, we will *rewire* edges during learning. The number of edges is kept fixed, but some edges may be deleted, and others added. Because deleting one edge and adding another are small local changes to the graph, many of the parameters (the edge weights) can be carried over. The learning process does not need to start from scratch.

Although it is useful to think of a network as a full graph conceptually, an implementation of a full graph, or a network with fully-connected layers, does not benefit from sparsity. As shown in sections 2.2 and 2.3, methods that do include absent edges in their representation of the network, have no advantage over dense networks during training. We would like to avoid storing zeros, or spending power to multiply and add zeros.

Instead of representing the network as a matrix of weights, we can explicitly store the indices of incoming edges to a neuron. Fixing the number of edges in the representation means that we can no longer represent networks that are less sparse than the chosen sparsity. Hence, optimisation methods that construct paths through the space of networks which may deviate from the desired sparsity (such as the methods discussed in Section 2.3) are no longer applicable. We can still use gradient descent methods to optimise the weights and biases of a given network, but if an edge is absent, it can never become present as a result of parameter optimisation alone.

To optimise the architecture of a network while fixing the representation at a given sparsity, we must replace edges. Deleting an edge in one place frees up space to insert an edge elsewhere. This raises two questions: **how to determine which edges to delete**, and **how to determine which neurons to connect?**

4.1 Edge utility

To determine which edges to delete, we must determine the *utility* of every edge: a measure of how useful or essential an edge is to the network. If removing an edge does not affect the output of the network by much, then perhaps that edge would be better spent elsewhere. Furthermore, the utility should be cheap to evaluate; it should not dominate the learning process.

Definition 4.1 · Let $k \in \mathbb{Z}_{>0}$ and $v_1, v_2 \in [1, \ldots, k]$. Let $(w, b) \in \mathrm{Net}(k)$ be a network with n input neurons and m output neurons, and σ an activation function. Let $T \subseteq \mathbb{R}^n$ be a set to evaluate the utility on. The evaluation function $f_{w,b,\sigma}$ as defined in Definition 3.20 is parametrised by w, so we can compute $\mathcal{L}_T(w)$, the loss of $f_{w,b,\sigma}$ on T with respect to the parameters w, as defined in Definition 3.7. Define the *utility of edge* $v_1 \to v_2$ *on* T as:

$$\begin{aligned} \operatorname*{utility}_T : [1, \ldots, k]^2 &\longrightarrow \mathbb{R} \\ (v_1, v_2) &\longmapsto \mathcal{L}_T(w') - \mathcal{L}_T(w) \end{aligned}$$

where

$$w'(i, j) = \begin{cases} 0 & \text{if } (i, j) = (v_1, v_2) \\ w(i, j) & \text{otherwise} \end{cases}$$

In other words, the utility of edge $v_1 \to v_2$, is the change in loss that would occur if the edge would be deleted. (Usually an increase in loss, although some edges can be harmful, and the loss can decrease by deleting such edge.) A similar quantity is called *saliency* in [LeCun, Denker, and Solla 1990], although the authors do not give a precise definition.

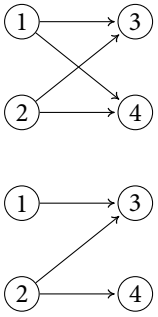


Figure 4.1 · The utility of edge $1 \to 4$ in the top network is given by the loss of the bottom network, minus the loss of the top network.

The utility as defined above is expensive to compute. We have to evaluate the loss over T twice for two networks that are similar, but not identical. Moreover, it is difficult to take advantage of the similarity: for changes close to the input neurons, there is little hope of reusing computed results, as eventually most neurons depend on the affected neuron. Even a small pertubation propagates to a large part of the network. Computing edge utility directly is infeasible, hence we need a heuristic.

A naive heuristic that is sometimes used, is to take the absolute weight of an edge as an approximation of its utility. The reasoning is that — because the activation of a neuron is a weighted sum — terms with a smaller absolute weight will contribute less. However, this ignores the scale of inputs, which can be very different. Recall that the neuron activation is defined in Definition 3.20 as:

$$a_i = \sum_j w(j, i)\sigma(a_j) + \cdots$$

Just because $|w(j,i)|$ is small does not mean that the contribution to the sum is small, because $|\sigma(a_j)|$ might be large. If σ is bounded (the sigmoid function, for example), absolute weight at least gives us an upper bound on utility. But when σ is unbounded (like the "ReLU" activation function), we cannot expect absolute weight to be a good heuristic.

A good starting point for a heuristic is the Taylor series of the loss. Let us fix for the moment all parameters apart from one, say the weight $w \in \mathbb{R}$, associated with edge $i \to j$. The network now has a single parameter of which we can determine the utility. Suppose the current parameter value is x. Then we have

$$\mathcal{L}_T(x+\Delta) = \mathcal{L}_T(x) + \Delta\frac{\partial \mathcal{L}_T}{\partial w}(x) + \Delta^2\frac{1}{2}\frac{\partial^2 \mathcal{L}_T}{\partial w^2}(x) + o(\Delta^2)$$

Plugging this into Definition 4.1, taking $\Delta = -x$, we get

$$\underset{T}{\text{utility}}(i,j) = -x\frac{\partial \mathcal{L}_T}{\partial w}(x) + x^2\frac{1}{2}\frac{\partial^2 \mathcal{L}_T}{\partial w^2}(x) + o(x^2) \tag{4.2}$$

For the purpose of approximation, we will discard the $o(x^2)$ term.

The first term is cheap to compute, as it contains $\partial \mathcal{L}_T/\partial w$, which is already computed for the gradient descent parameter update anyway. The second term is more expensive, because it involves a second derivative (the diagonal of the Hessian) which is not normally computed. [LeCun 1987, § 3.12.2] derives an approximation of the Hessian that can be computed using backpropagation, under the assumption that the Hessian is diagonal, and under the assumption that

$$i \neq j \implies \frac{\partial^2 a_k}{\partial a_i \partial a_j} = 0$$

That is, the change in activation of neuron k, when the activation of neuron i is modified a bit, is independent of the activation of neuron $j \neq i$. The cost of approximating the second term in this way is manageable, but still expensive in comparison to the first term.

In a local optimum of the parameters, the first term in Equation 4.2 will vanish, because $\partial \mathcal{L}_T/\partial w = 0$. What is left is a heuristic for the utility based on the second derivative:

$$\underset{T}{\text{utility}}(i,j) \approx x^2\frac{1}{2}\frac{\partial^2 \mathcal{L}_T}{\partial w^2}(x)$$

This heuristic is dubbed *optimal brain damage* by [LeCun, Denker, and Solla 1990], and the authors show that it is a better utility heuristic than simply taking the absolute weight $|x|$. In our setting however, we are not yet in a local optimum: we want to rewire edges during the learning process, *before* the parameters have converged. Furthermore, even for networks that are fully trained, it is not clear that $\partial \mathcal{L}_T/\partial w = 0$. Techniques such as

early stopping and weight decaying (ℓ_2 regularisation) are designed to avoid reaching an optimum, to prevent overfitting on the training data. In those cases we cannot ignore the first term. On the contrary, $\partial \mathcal{L}_T / \partial w$ will dominate outside of an optimum.

Because we expect parameters to be rarely close enough to an optimum for the second term to matter, we propose the following utility heuristic instead:

$$\underset{T}{\text{utility}}(i, j) \approx -x \frac{\partial \mathcal{L}_T}{\partial w}(x) \tag{4.3}$$

The core insight here, is that **parameters are almost never close to an optimum** in practice. Even though optimisation algorithms find parameters with low loss, these parameters are not an optimum, in the sense that $\partial \mathcal{L}_T / \partial w \neq 0$. Empirical evidence supports this observation. When optimising with stochastic gradient descent or a variation such as Adam, $\|\partial \mathcal{L}_T / \partial w\|_2$ converges to a nonzero value in practice. This is to be expected: even if $\partial \mathcal{L}_{U_1} / \partial w = 0$ for some batch $U_1 \subseteq T$, it is unlikely that $\partial \mathcal{L}_{U_2} / \partial w = 0$ for a different batch $U_2 \subseteq T$. As a result, a linear approximation of the loss function is a good approximation.

To verify the effectiveness of the heuristic, we can compute the true edge utility for edges in a fully-connected network, and compare it against various heuristics. We will evaluate heuristics on the following network:

> The *Tensorflow deep* MNIST network, an MNIST digit classification network described in the Tensorflow documentation. This network consists of two convolutional layers, followed by two fully-connected layers. The first fully-connected layer reduces the number of features from $7 \cdot 7 \cdot 64$ to 1024 features, and the second layer from 1024 features to the final 10 classes. The activation function $\sigma(x) = \max\{0, x\}$ is used after the first fully-connected layer. The network is trained on the training set until the loss on the test set no longer decreases. We then evaluate utility on the test set.

A comparison of the heuristics is shown in Figure 4.2. We can observe a few things from the plots:

- The two fully-connected layers of the *Tensorflow deep* MNIST network differ notably. This is explained by the first fully-connected layer being somewhat redundant. The distribution of weights after learning is not very different from the initial distribution. The learning process might have stopped too early for the layer to learn significantly, indicating that the network is overparametrised.

- Absolute weight provides an upper bound for utility. This makes sense: after all,

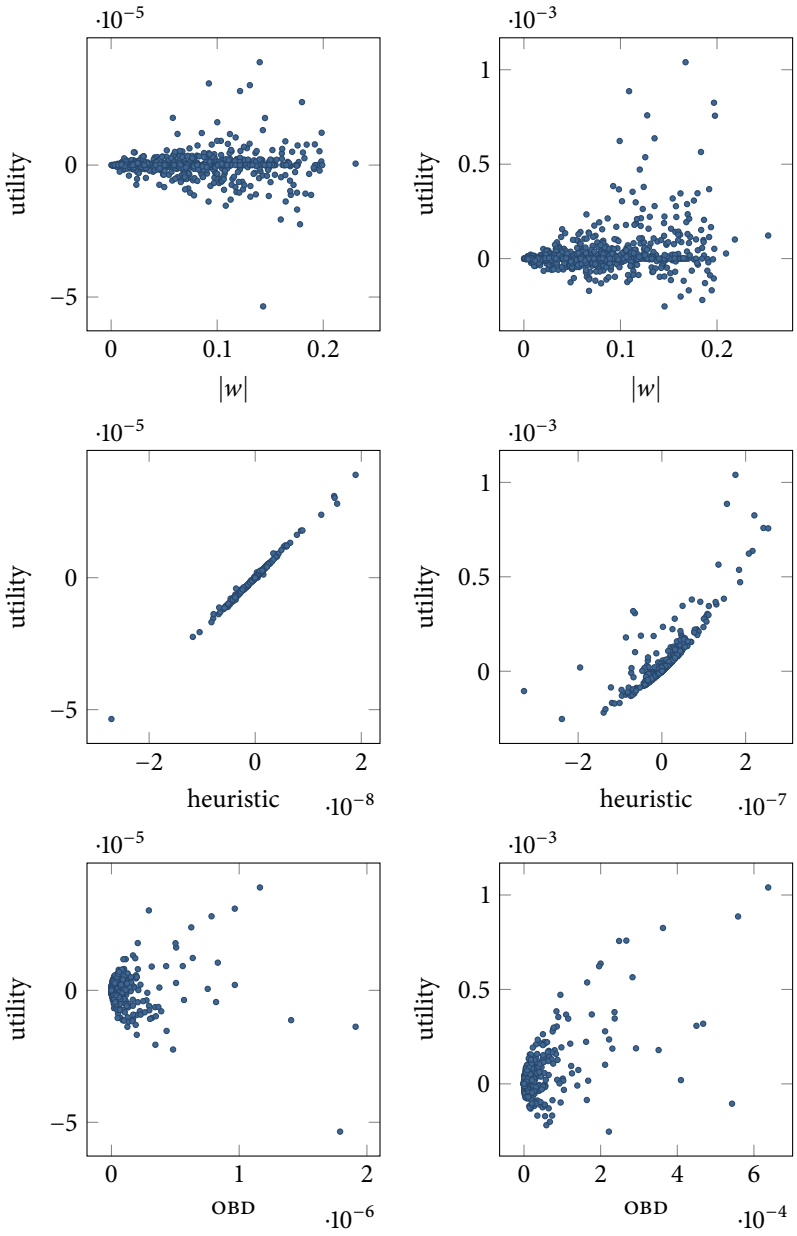


Figure 4.2 · Comparison of different utility heuristics applied to the weights of the *Tensorflow deep* MNIST network. Plots show the true utility versus a heuristic for 750 weights drawn uniformly from all weights in a layer. The left and right columns show weights in the first and second fully-connected layer respectively. Heuristics from top to bottom: absolute weight, our proposed heuristic, and optimal brain damage.

as a weight goes to zero, the utility of the edge goes to zero as well. However, absolute weight is a poor heuristic for edge utility. Absolute weight classifies a large portion of low-utility edges as useful, and many of the more useful edges as low-utility.

- Our proposed heuristic performs extremely well on the first fully-connected layer, and it still performs well on the second fully-connected layer. This agrees with the

hypothesis that the second fully-connected layer is closer to a parameter optimum than the first layer: we expect our heuristic to deteriorate close to an optimum.

- Optimal brain damage performs poorly on the first fully-connected layer, and somewhat better on the second fully-connected layer. Again, this agrees with the hypothesis that the second fully-connected layer is closer to a parameter optimum than the first fully-connected layer: optimal brain damage works best in a parameter optimum.

We now have a partial answer to the first question of this chapter: how to determine which edges to delete? The utility heuristic proposed in this section is a good proxy for how useful an edge is, and it is cheap to compute. We can estimate the utility of every edge, rank edges by utility, and delete the n least useful ones. This cannot be the full story though — even if an edge between two neurons is not useful, deleting it only makes sense if we have a *better* candidate edge to replace it with. This will be the topic of Section 4.3. Moreover, at a certain point we should expect the network to converge to an architecture that is optimal for the task. In a network that uses the available edges optimally, *all* edges should be useful. This means that replacing e.g. the 5% least useful edges is not a good strategy. The estimated utility of an edge, and the merit of a potential replacement edge, will together determine whether to delete the edge.

4.2 Applications in quantisation

As an aside, the heuristic proposed in the previous section can be used to guide quantisation in addition to guiding edge pruning. Quantisation is the process of replacing a set of weights $w \in \mathbb{R}^k$ with an approximation qw' for some $q \in \mathbb{Q} \setminus \{0\}$ and $w' \in \mathbb{Z}^k$. Quantised weights can be stored in less memory than their unquantised counterparts, and evaluation of a quantised network might use fixed-point arithmetic rather than floating-point arithmetic for increased performance. The job of a quantiser is to choose w' and q in such a way that the loss of the network does not degrade too much, while also not spending too much memory on large integers. For small values of q the approximation can be very precise, at the cost of large elements of w', which require many bits to store. For large values of q, elements of w' can be small integers that can be stored in a few bits (e.g. 8 bits rather than the 32 bits used for floating-point weights), however the approximation will be coarse. A quantiser makes a trade-off between increase in loss, and reduction in storage space. To estimate the loss increase due to quantisation, an adaptation of the utility heuristic may be used.

Example 4.4 · Suppose a network has weight vector $(0.2, -0.1, 0.3, 88.7) \in \mathbb{R}^4$, and we want to quantise to 3-bit signed integers, integers in $\{-4, -3, \ldots, 3\}$. A quantiser may

propose the following two quantisations:

$$q = 10^{-1} \qquad w' = (2, -1, 3, 3)$$
$$q = 30 \qquad w' = (0, 0, 0, 3)$$

Assuming that the storage for q is negligible (because it is shared among all weights, hence the space can be amortised), and weights are stored as fixed-width integers, both proposals require the same amount of storage space. The decision of which quantisation to apply should depend solely on loss increase. The first proposal reproduces the first three elements perfectly, but the last element is off by a large margin. The second proposal gets the orders of magnitude right, at the cost of losing precision in the first three elements. Which of the two proposed schemes is preferred, depends on the utilities of the associated weights. Perhaps the fourth weight was not very useful, so rounding it will not affect the loss by much. If on the other hand the fourth weight was important, the second proposed quantisation scheme might be better. □

The utility heuristic proposed in the previous section is a special case of a more general problem: estimating how a change in network parameters will affect the loss. Removing an edge is the extreme case of changing the associated weight to zero. In general, if the value of parameter w is changed from x_1 to x_2, we can expect a change in loss proportional to

$$(x_2 - x_1)\frac{\partial \mathcal{L}_T}{\partial w}(x_1) + o(x_2 - x_1) \tag{4.5}$$

This is the more general form of Equation 4.3. Like before, we can discard the $o(x_2 - x_1)$ term under the assumption that it small, to obtain an approximation of the change in loss. To estimate the change in loss due to quantisation, we can sum the estimated change in loss due to changing each of the parameters.

To conclude, the observation that parameters are almost never at an optimum, is useful beyond pruning edges. Not being in a parameter optimum means that a linear approximation of the loss as function of the parameters is a good approximation, and this can be used to guide many decisions, be it which edge to prune, or which quantisation scheme to apply.

4.3 Selecting candidate edges

Now that we know which edges in a network are least useful, we need to find candidate edges on which resources would be better spent. When sparsifying an existing fully-connected network, we know exactly which edges are useful: the ones that remain after deleting the least useful edges. But if we wish to evolve the architecture of a network with

a fixed number of edges, we have to determine in advance whether an edge is going to be useful. This is difficult in at least one sense — if it were possible in general to predict which edges are useful, we would not need to learn network parameters at all. By Theorem 3.25, the optimal architecture of a network is not unique, and so the best candidate edges depend on the current parameter values — values that change during training. Nonetheless, in some cases we can say that a candidate edge would be beneficial. In this section we will identify such beneficial candidate edges.

Before we can identify candidate edges, we need to take a step back to look at batch learning. When we do gradient descent, we typically do not compute $\nabla \mathcal{L}_T$, the gradient of the loss over the full training set T. Instead, we compute $\nabla \mathcal{L}_U$ over $U \subseteq T$, a *batch*, where U is a random subset of T. Then $\nabla \mathcal{L}_U$ is an unbiased estimate of $\nabla \mathcal{L}_T$. Recall that $\mathcal{L}_U$ is defined as a sum over U in Definition 3.8, and consequently its gradient with respect to the parameters can be written as a sum over U:

$$\nabla \mathcal{L}_U = \nabla \left[\sum_{x \in U} \mathcal{L}(\cdot, x) \right] = \sum_{x \in U} \nabla \mathcal{L}(\cdot, x)$$

Suppose U has p elements, and the loss function is for a network with k parameters. Then we get kp partial derivatives. For every parameter w, we get a vector $\delta \in \mathbb{R}^p$ of derivatives, and we have

$$\frac{\partial \mathcal{L}_U}{\partial w} = \sum_{i=1}^{p} \delta_i = \langle \delta, 1 \rangle$$

where $1 \in \mathbb{R}^p$ is the vector of ones. If w is at an optimum, $\partial \mathcal{L}_U / \partial w = 0$, and hence $\langle \delta, 1 \rangle = 0$. But that does not mean that the elements of δ are zero! In a sense, $\|\delta\|$ indicates how much samples in the batch disagree about the parameter value. The key point is that in batch learning we get the partial derivative of the loss with respect to the parameter for every sample in the batch.

To understand where adding edges might make sense, we first need to understand which neurons are responsible for errors in the network. Rather than studying the derivative of $\mathcal{L}_U$ with respect to one of the network parameters, we can take the derivative with respect to one of the neuron activations a_i. In a network with biases, this derivative happens to be equal to the derivative of $\mathcal{L}_U$ with respect to the bias $b(i)$, but consider just the neuron activation for now. As before, we get such a derivative for every sample in the batch U, so we again get a vector $\delta \in \mathbb{R}^p$ of derivatives. Here the elements of δ can be interpreted as the error of the neuron, for the given sample. Again, in an optimum the total neuron error over the test set is zero, $\langle \delta, 1 \rangle = 0$ — but the individual errors for every sample need not be.

Now that we have a measure of neuron error, we can use that to determine whether to connect two neurons. For every sample in the batch, neuron j has a particular output $\sigma(a_j)$. As a slight abuse of notation, we will denote by $\sigma(a_j) \in \mathbb{R}^p$ the vector of outputs,

one element for every sample in the batch. We now have neuron outputs $\sigma(a_j)$ of neuron j, and errors $\delta \in \mathbb{R}^p$ at neuron i. If they are somehow correlated — if neuron j is active only when neuron i is wrong — then it might make sense to add an edge $j \to i$. More formally, if $\langle \sigma(a_j), \delta \rangle \neq 0$, then it might be worthwhile to add the edge, and the greater the absolute value of the inner product, the more lucrative it would be to add the edge.

Neuron activation and error need not have an inherent length scale, so the inner product in itself is not a good measure of correlation. (See Proposition 3.24.) Fortunately, because we can choose the weight for the new edge, the length scales are not important. Therefore the cosine similarity is a good measure of correlation.

Definition 4.6 · Let $(w, b) \in \mathrm{Net}(k)$ be a network with n imputs and m outputs, and let $T \subseteq \mathbb{R}^n$ be a test set of p elements. Let $i, j \in [1, \ldots, k]$ be two neurons in the network. Denote by $\delta \in \mathbb{R}^p$ the vector of derivatives of the loss with respect to a_j, for all samples in T. Define the *correlation* of neuron i and j on T as

$$\underset{T}{\mathrm{corr}}(i, j) = \frac{\langle \sigma(a_i), \delta \rangle}{\|\sigma(a_i)\| \cdot \|\delta\|}$$

The norm used here is the Euclidean norm.

For neurons connected by an edge, we expect their correlation to be zero at a parameter optimum. If not, then by changing the weight of the edge and bias of the neuron, we can reduce the norm of the error. Therefore, correlation is a good way to rank *absent* edges, but correlation does not say anything about how useful *present* edges are. This makes correlation a useful heuristic for deciding which edges to add, while we can use the utility heuristic from the previous section to decide which edges to remove.

Armed with neuron correlations, we can answer the second question of this chapter: how to determine which neurons to connect? We should connect neurons that have a large absolute correlation. The sign of correlation is not relevant — it can be cancelled by the sign of the edge weight. What remains is a matter of acting on the knowledge: given utility estimates of all present edges, and correlations for all absent edges, which particular edges should we delete, and which particular edges should we add? In other words: given the current graph of the network, propose a better graph based on utility and correlation.

Proposing a new graph can be done in many ways. Supposing the number of incoming edges per neuron is fixed, the following rewiring strategies come to mind:

- Delete x % of the edges with the lowest utility. For a neuron j that now has an available incoming edge, connect the edge to the neuron i for which $|\mathrm{corr}(i, j)|$ is maximal. If neuron j has more than one available edges, connect them to the next most correlated neurons.

- For x % of the neuron pairs (i, j) with largest absolute correlation, delete the incoming edge from neuron j with the lowest utility, and replace it with an edge $i \to j$. This does affect neuron correlations and the utilities of the remaining neurons, but a single local change should be small enough for the estimates to still be valid. On the other hand, we cannot expect the estimates to still be valid after rewiring half the network. [LeCun, Denker, and Solla 1990, Figure 1] shows how their optimal brain damage estimate holds up as more and more edges are removed, and we expect to see a similar curve. The number of edges that can be rewired before estimates need to be updated would have to be determined empirically.

- In both cases, rather than taking a fixed percentage, we could take a number that decreases as training progresses, or we could set a threshold on the utility or correlation.

In general, an algorithm would have to balance the expected utility of a new edge against the utility of the edge that will be removed. A rewiring strategy is a particular choice of making this trade-off. Furthermore, algorithmic complexity may play a role here. Estimating edge utility can be done at no additional algorithmic cost, but for instance deleting a percentage of least useful edges would require $O(n \log n)$ comparisons for n edges, as we would need to identify $O(n)$ least useful edges. (For a fixed number of least useful edges we could do it in $O(n)$ time instead, but for large n this is ineffective, because the fraction of edges rewired is negligible.) Moreover, instantiating a percentage of absent edges would require computing the $O(k^2)$ correlations for k neurons, and selecting the most correlated pairs would require $O(k^2 \log k)$ comparisons. This may seem worse than deleting a percentage of edges, but recall that a fully-connected network of k neurons has $O(k^2)$ edges. Taking a fixed fraction of edges does not eliminate the k^2 scaling, nor does dividing the network into fully-connected layers. It seems that rewiring would come at significant algorithmic cost.

Despite the apparent computational cost, in practice the constants matter: this is why we try to make networks sparser in the first place. For 10^5 edges, performing a few steps of gradient descent is typically more expensive than sorting edges by utility, even though gradient descent performs a number of multiplications that is linear in the number of edges. Furthermore, some of the cost can be amortised over multiple rewiring runs. For example, we may decide to rewire the 2% least useful edges, but implement that using a threshold, and compute the new threshold (the 2nd percentile of edge utility) only every fifth rewiring round, assuming that the distribution of edge utility does not change by a lot. Finally, if the number of edges to rewire is small, then we can use the maximal inner product search techniques from Section 2.6 to do better. Suppose we rewire only $O(k)$ edges rather than $O(k^2)$ edges per round. A few edges per neuron seems reasonable — rewiring should be a *small* change to the network after all. Suppose furthermore that the

decision of which edges to remove was not based on correlations, so there was no need to compute correlations so far. Then we can efficiently find a highly correlated neuron using an approximate nearest neighbour search. By using the geometric structure of the problem, rewiring is not a bottleneck in practice.

Inner product search and nearest neighbour search

In the case of a maximal inner product search through a set of unit-norm vectors, inner product search and nearest neighbour search turn out to be equivalent.

Proposition 4.7 · Let $u \in \mathbb{R}^n$ be a unit norm query vector (the needle), and $S \subseteq \mathbb{R}^n$ a set of unit vectors (the haystack). Then it holds that

$$\arg\max_{v \in S} \langle u, v \rangle = \arg\min_{v \in S} \|u - v\|$$

Here $\| \cdot \|$ denotes the Euclidean norm.

Proof: Note that instead of minimising $\|u - v\|$, we may also minimise $\|u - v\|^2$. By expanding the square, we get

$$\begin{aligned} \arg\min_{v \in S} \|u - v\|^2 &= \arg\min_{v \in S} \sum_{i=1}^{n} (u_i - v_i)^2 \\ &= \arg\min_{v \in S} \sum_{i=1}^{n} u_i^2 - 2\, u_i v_i + v_i^2 \end{aligned}$$

Because u and v are unit vectors, we have

$$\begin{aligned} &= \arg\min_{v \in S} 2 - \sum_{i=1}^{n} 2\, u_i v_i \\ &= \arg\max_{v \in S} \sum_{i=1}^{n} u_i v_i \\ &= \arg\max_{v \in S} \langle u, v \rangle \end{aligned}$$

□

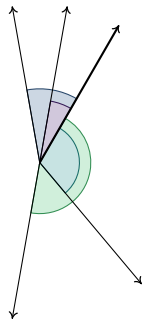


Figure 4.3 · For a set of unit vectors, the vector with the smallest angle to a given vector, is the nearest neighbour.

This equivalence is useful, because it means that we can find correlated neurons using a nearest neighbour search, in addition to using an inner product search. This opens op the opportunity of using a more specialised algorithm. Algorithms to find the nearest neighbour and approximate nearest neigbour have been outlined in Section 2.6. In our case, we want to find the maximal *absolute* correlation. To solve the absolute maximal inner product problem, we can perform the query twice, once with query vector u, and once with $-u$. To conclude, nearest neighbour search can be used to implement rewiring efficiently.

4.4 Optimal weight updates

When we rewire, we want the changes to be small, local changes, so most of the parameters can be carried over, and learning does not have to start from scratch. Still, by changing the graph, we make a "jump" in the parameter space $\mathrm{Net}(k)$. An optimiser constructs a path through this space, and by changing the graph, we interrupt this path. The jump can be much larger than the typical step taken by the optimiser. For momentum-based optimisers such as Adam [Kingma and Ba 2014], this can be an issue. We have to restart the optimisation process, albeit with a warm start. The goal of rewiring is not to immediately reduce the loss. On the contrary, the goal is to find an architecture for which an optimiser is able to *later* reach a lower loss. Therefore, it is important to choose edge weights for the new edges carefully, to minimise the effects of changes.

Especially for edges close to the inputs, new edge weights matter. If due to a change in the graph, the output of the neuron changes, that change cascades, and the output of the network may be vastly different. Denote by $a_i \in \mathbb{R}^p$ the activation of neuron i before rewiring, for all p samples in a batch. Denote by $a_i' \in \mathbb{R}^p$ the activation of neuron i after rewiring. Then we have a difference $\Delta_i = a_i' - a_i$. It seems reasonable to demand that the expected value of neuron i does not change: $\langle \Delta_i, 1 \rangle = 0$. Here $1 \in \mathbb{R}^p$ denotes the vector of ones. This in itself is not a strong guarantee though: whatever the new weight, we can achieve $\langle \Delta_i, 1 \rangle = 0$ by adjusting the bias of neuron i. This leaves us with one more degree of freedom (the weight of a new edge) that we can use to minimise e.g. $\|\Delta_i\|_1$, $\|\Delta_i\|_2$, or $\|\Delta_i\|_\infty$. In other words, we get to minimise the change in neuron activation for each individual sample in the batch.

Recall from Definition 3.20 that for a given network $(w, b) \in \mathrm{Net}(k)$, the neuron activation for a non-input neuron i is given by

$$a_i = b(i) + \sum_j w(j, i)\sigma(a_j)$$

We know which edges have been removed, and we know which edges have been added. Suppose for the moment that no more than one edge has been added. Then we get

$$\Delta_i = \Delta b(i) + w(r, i)\sigma(a_r) - \sum_s w(s, i)\sigma(a_s)$$

where r is the neuron that the new edge connects to, s ranges over the neurons s for which the edge $s \to i$ has been deleted, and $\Delta b(i)$ is the change in bias of neuron i. This expression extends to vector form where the elements of a_r and a_s represent the values for different samples in the batch, σ is applied elementwise, and $\Delta b(i)$ is replaced with $\Delta b(i) \cdot 1$, where $1 \in \mathbb{R}^p$ is the vector of ones. More abstractly, we have a problem of the following form:

$$\text{minimise} \quad \|\Delta\| = \|\beta 1 + \alpha u - v\| \quad \text{given} \quad \langle \Delta, 1 \rangle = 0$$

where $\alpha, \beta \in \mathbb{R}$ and $1, u, v \in \mathbb{R}^p$. Here $\| \cdot \|$ denotes the ℓ_2-norm. Solving the constraint for β, we find

$$\beta = \frac{\langle v - \alpha u, 1 \rangle}{p}$$

Define

$$\bar{u} = p^{-1}\langle u, 1 \rangle 1 \qquad \text{and} \qquad \bar{v} = p^{-1}\langle v, 1 \rangle 1$$

In other words, the elements of $\bar{u}$ and $\bar{v}$ are the means of the elements of u and v respectively. Then we can express $\|\Delta\|$ as

$$\|\Delta\| = \|\bar{v} - \alpha\bar{u} + \alpha u - v\| = \|\alpha(u - \bar{u}) - (v - \bar{v})\|$$

The part $\alpha(u - \bar{u})$ defines a line, and to minimise $\|\Delta\|$, we want to find the point on this line closest to $v - \bar{v}$. It follows that

$$\alpha = \frac{\langle u - \bar{u}, v - \bar{v} \rangle}{\|u - \bar{u}\|^2} \tag{4.8}$$

In the case of multiple new edges, we have multiple new weights to choose. Here the problem generalises to finding the closest point on the hyperplane spanned by the vectors

$$V = \{u - \bar{u} \mid u = \sigma(a_r), r \in A\}$$

where A is the set of neuron indices of new incoming edges. For a single new incoming edge, the hyperplane is a line, and the new weight is given by Equation 4.8. In general, the point on the hyperplane closest to $v - \bar{v}$ is a linear combination of vectors in V, and the coefficients are the optimal new weights.

By choosing weights of new edges as described in this section, we can ensure that neuron outputs changes as little as possible after rewiring. This avoids cascading of changes, which ensures that the weights of dependent neurons remain relevant. This in turn means that the new, modified network, starts the learning process from reasonable parameters. This means that optimisation has a warm start, it does not need to start from scratch.

4.5 Putting it together

In the previous sections we have outlined how to identify the least useful edges in a network, how to propose new candidate edges, and how to pick a weight for new edges to disturb the network as little as possible. The combination of these enables optimising the network architecture during the learning process. When putting all of this together in an implementation, a few interesting issues arise.

Firstly, we need a way to evaluate a non-layered network. In a layered network evaluation order is determined by the layer ordering. In a network that is not strictly layered, we can still order neurons into layers, in such a way that edges only go from a layer to the layers after it, with no internal edges within a layer, and no edges to earlier layers. An example is shown in Figure 4.4. The layers define a partial order on the neurons: we have $i < j$ if neuron i is in a layer before j. Neurons in the same layer are unordered among eachother. By picking *any* order for the neurons within a layer, we can extend the partial order to a total order. This is shown at the bottom of Figure 4.4, with $i < j$ if i is left of j. The total order can be maintained as an array containing neuron indices, and this defines an evaluation order. To exploit parallelism, it is possible to instead store the layer structure (top right in Figure 4.4), and evaluate all neurons within a layer in parallel. An initial evaluation order can be obtained by topologically sorting all neurons.

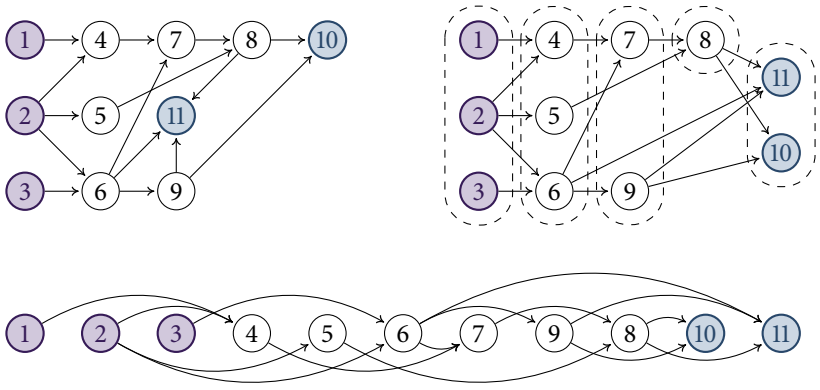
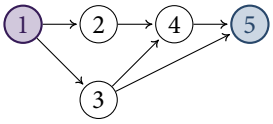


Figure 4.4 · A network, not layered (top left). Input and output neurons coloured • and • respectively. We can arrange neurons into layers, where neurons can only have edges to neurons in a later layer (top right). This defines a partial order. We can extend the partial order to a total order by enumerating neurons in layer order. One possible total order is shown at the bottom.

Now that we have a way to evaluate the network, we need a way to optimise its parameters. Gradient descent or a more sophisticated variant (as outlined in Section 3.2) is suitable for this; this is no different from traditional multilayer perceptron networks. The derivative of the loss with respect to the network parameters can be computed using backpropagation in the reverse evaluation order.

Example 4.9 · Consider the following network $(w, b) \in \mathrm{Net}(5)$, with one input neuron (1) and one output neuron (5).



A total order on the neurons compatible with the network architecture, would be $(1, 2, 3, 4, 5)$, so we use this as evaluation order. Suppose the activation a_1 of neuron 1 is given. Then the other neuron activations are given by

$$
\begin{aligned}
a_2 &= b(2) + w(1,2)\,\sigma(a_1)\\
a_3 &= b(3) + w(1,3)\,\sigma(a_1)\\
a_4 &= b(4) + w(2,4)\,\sigma(a_2) + w(3,4)\,\sigma(a_3)\\
a_5 &= b(5) + w(3,5)\,\sigma(a_3) + w(4,5)\,\sigma(a_4)
\end{aligned}
$$

If we store neuron activations in an array, we could fill it in the evaluation order, without ever reading uninitialised values. For derivatives, the order is backwards. Recall from Definition 3.7 that $\mathcal{L}(w,x) = L(g_w(x), f(x))$ for some loss function L. In this case $x \in \mathbb{R}$, $a_1 = x$, $g_w(x) = a_5$, and $f(x)$ is the target output of the network. If we fix x and $f(x)$, we can compute derivatives of L using the chain rule, in the reverse evaluation order. Denote by σ' the derivative of the activation function σ. Then we have:

$$
\begin{aligned}
\frac{\partial L}{\partial a_4} &= \frac{\partial L}{\partial a_5}\frac{\partial a_5}{\partial a_4} = \frac{\partial L}{\partial a_5} w(4,5)\,\sigma'(a_4)\\
\frac{\partial L}{\partial a_3} &= \frac{\partial L}{\partial a_5}\frac{\partial a_5}{\partial a_3} + \frac{\partial L}{\partial a_4}\frac{\partial a_4}{\partial a_3} = \left(\frac{\partial L}{\partial a_5} w(3,5) + \frac{\partial L}{\partial a_4} w(3,4)\right)\sigma'(a_3)\\
\frac{\partial L}{\partial a_2} &= \frac{\partial L}{\partial a_4}\frac{\partial a_4}{\partial a_2} = \frac{\partial L}{\partial a_4} w(2,4)\,\sigma'(a_2)\\
\frac{\partial L}{\partial a_1} &= \frac{\partial L}{\partial a_3}\frac{\partial a_3}{\partial a_1} + \frac{\partial L}{\partial a_2}\frac{\partial a_2}{\partial a_1} = \left(\frac{\partial L}{\partial a_3} w(1,3) + \frac{\partial L}{\partial a_2} w(1,2)\right)\sigma'(a_1)
\end{aligned}
$$

Note again: if we store the derivatives in an array, we could fill it in the reverse evaluation order, without ever reading uninitialised values. Derivatives of the loss with respect to the weights and biases can be expressed in terms of the above derivatives. □

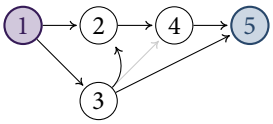
When deciding which edges to rewire, we must take care not to introduce cycles in the graph of the network. We can use any of the strategies proposed in Section 4.3, but edges that would introduce a cycle should not be considered as candidates for rewiring, even if such an edge would connect highly correlated neurons. There are two ways of avoiding cycles:

- We can discard edges that would introduce a cycle in advance. If for a fixed neuron j we want to consider candidate edges $i \to j$, then we can mark all neurons that depend on neuron j, and exclude marked neurons from the candidates. Such a process needs to visit $O(k)$ neurons, where k is the number of neurons in the graph. The evaluation order can be used to reduce the number of neurons to visit: neurons that come before j in the evaluation order cannot depend on j.

- We can insert an edge, and detect cycles later, when we restore the evaluation order. If it turns out that no such order exists, the edge should not have been inserted. This approach can be more efficient than the former, because cycle detection is a side effect of updating the evaluation order, which needs to to be done anyway.

Once we have a candidate edge to rewire, the next task is to update the evaluation order. We could of course topologically sort all neurons again, but this seems wasteful: we already had an ordering of neurons before rewiring, and we know which edges were removed and added. Can this knowledge be used to update the evaluation order more efficiently? Removing edges does not invalidate the evaluation order, therefore we only need to update the evaluation order for *inserted* edges. This problem is called *online topological ordering*, and solutions have been discussed in Section 2.5. Accordingly, it is possible to efficiently update the evaluation order after rewiring, using online topological ordering algorithms.

Example 4.10 · Consider again the network from Example 4.9. Suppose that after a number of training steps, we find that $\text{corr}(3, 2)$ is high, and the estimated utility of edge $3 \to 4$ is the lowest of the network. Then we may want to delete edge $3 \to 4$ and insert edge $3 \to 2$, as shown below.



This invalidates the previous evaluation order $(1, 2, 3, 4, 5)$, for neuron 2 now depends on 3, but 2 comes before 3 in the evaluation order. We only need to reorder the part of the evaluation order between 2 and 3 (inclusive): all neurons before 2 can be considered as having edges into a subgraph composed of neuron 2 and all neurons after it, and an internal change to this subgraph does not affect the complement of the subgraph. Similarly, all neurons after neuron 3 can be considered as having edges from a subgraph composed of neuron 3 and all neurons before it, and an internal change to this subgraph does not affect the ordering of its complement. The new evaluation order would be $(1, 3, 2, 4, 5)$. □

In this section we have shown how the ideas from the previous sections come together. We can train a sparse non-layered network, and incrementally improve its architecture by rewiring edges. Doing so involves deciding which edges to delete, by estimating their utility, and determining which edges to insert, based on correlation between neurons. For training and evaluation we need an evaluation order, which we can keep up to date using online topological ordering. In the next section we will show the results of this approach.

4.6 Results

We implemented the framework outlined in this thesis, and applied it to predicting characters of the *Enwik9* dataset. Characters are lowercased and numbers and punctuation are removed, which leaves 27 symbols (the Latin alphabet and the space). The input to the network is a one-hot encoding of the past 20 symbols (so the network has 540 input neurons), and the output is a prediction of the next symbol, as a probability distribution over the 27 symbols (so the network has 27 output neurons). Performance of the network is quantified using the cross-entropy loss function. For a fully-connected network, 1700 neurons is where the returns of adding more neurons start to diminish, so we use a total of 1700 neurons for the sparse network as well. The number of incoming edges to each neuron is fixed, and the same for all neurons in the network. This has implementation advantages: edges can be stored in a fixed-size array, and there is no need for the extra indirection of a heap-allocated dynamically sized array. We examine various sparsities: either 58, 29, or 15 incoming edges per neuron, so the network is fairly sparse. The activation function used is $\sigma(x) = \max\{0, x\}$. To intialise the network, neurons are added to the graph one by one, and connected to neurons sampled uniformly from the graph so far. We train the network with the Adam optimiser [Kingma and Ba 2014] using a batch size of 512. Every 2048 training steps, the network is rewired. For new edges, we initialise the Adam parameters m_t and v_t to zero, and we also reset Adam's timestep t to zero after every rewiring round. The rewiring strategy is to rewire incoming edges of neuron i, if there exists another neuron j such that $\text{corr}(j, i)$ exceeds a threshold, which we vary to control how many edges get rewired. The incoming edge with the lowest utility is replaced. Although our approach eliminates the need to choose a network architecture in advance, it still has many hyperparameters, such as the number of neurons, the number of edges per neuron, and when to rewire.

Because we optimise using stochastic gradient descent, the training loss is noisy. To get a grip on how rewiring affects the loss in general, we train 12 instances of the network, all with a different random seed. Training losses are grouped per 64 training steps, and we show the median, 25th percentile, and 75th percentile, of the $64 \cdot 12$ loss values. The results are shown in Figure 4.5.

We make a few observations:

- Rewiring does not improve performance in any of our measurements.
- Even after many training steps, there are still plenty of correlated neurons that pass the rewiring threshold. Training does not in itself eliminate correlation between neurons.
- Rewiring too many edges too often can make the loss diverge: the loss does not

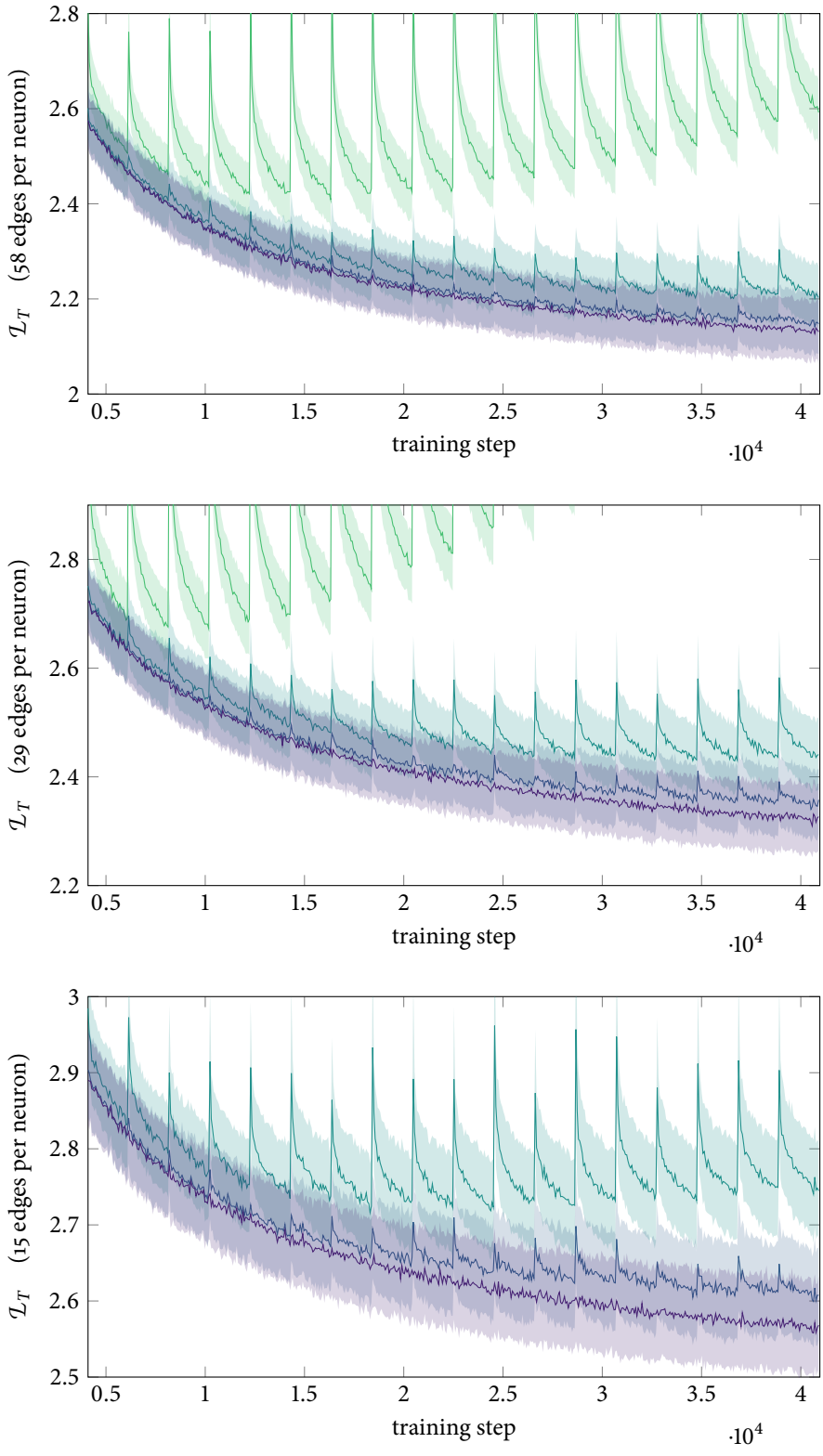


Figure 4.5 · Cross-entropy training loss of a character prediction network. From top to bottom, a network with 1160 non-input neurons and respectively 58, 29, and 15 incoming edges per neuron. The lines indicate the median loss over 64 training steps and 12 different seeds, the shaded areas indicate the 25th and 75th percentile.

- Rewiring disabled
- Rewiring 0.0089% of edges
- Rewiring 0.097% of edges
- Rewiring 0.93% of edges

With 15 edges per neuron, rewiring 0.93% of edges (•) diverged immediately, and as such is not shown in the graph.

recover before the next rewiring round. This happens to the • coloured graph, where about a percent of edges are rewired.

- The denser networks perform better than sparser networks.

One thing that stands out is that the network takes a while to recover after rewiring. This is not a problem in itself, as long as there are sufficient training steps to recover. As the loss converges to a minimum, the time to recover increases, and rewiring is no longer useful. This makes sense: at a certain point we should settle on a particular network architecture. We cannot keep on rewiring all the time. A threshold on correlation alone is not sufficient to reduce the number of rewired edges as training progresses.

Furthermore, there is a tension between rewiring few edges often, and making larger changes infrequently. Making frequent small changes to the network better integrates learning parameters and architecture into one process. However, it is not clear how to select the edges to rewire in this case: newly added edges will have a low utility, because their weights have not been learned yet, and the rest of the network has not adapted to the new edges. These edges would need to be excluded from rewiring for a while, but it is not clear for how long: waiting until convergence makes rewiring *less frequent*, but rewiring too early could mean that the same edges, or edges of the same neurons, get rewired all the time. Figuring out how to make frequent small local changes that do not interfere with one another, would be an interesting topic for future research.

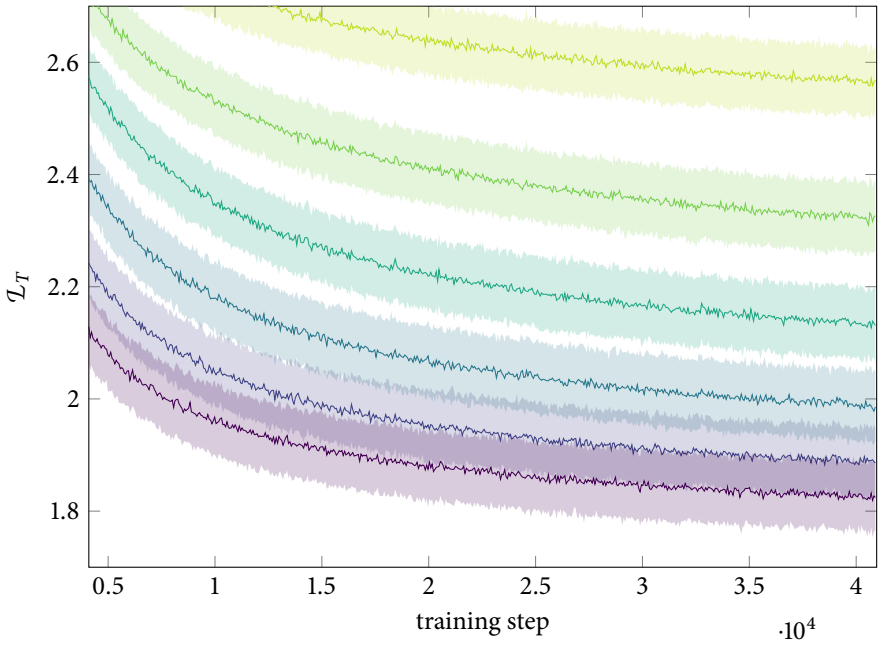


Figure 4.6 · Cross-entropy training loss of a sparse character prediction network with 1160 non-input neurons, rewiring disabled. The lines indicate the median loss over 64 training steps and 12 different seeds, the shaded areas indicate the 25th and 75th percentile.

- 15 edges per neuron
- 29 edges per neuron
- 58 edges per neuron
- 116 edges per neuron
- 232 edges per neuron
- 464 edges per neuron

The effect of sparsity is shown in Figure 4.6. This figure shows the training loss for networks of various sparsities while keeping the number of neurons fixed. The networks are constructed in the same way as earlier in this section, and no rewiring is applied. The graph shows that for a fixed network architecture, denser networks perform significantly better than sparser networks. However, sparsification of a dense network could still result in a better network than training the sparse network from scratch, especially by pruning using the utility heuristic from Section 4.1. This area of research has been explored in literature (see also Section 2.2) so we make no attempt to further investigate the effects here. Rather, these results form the baseline for Figure 4.5.

From the results in this section we can conclude that we have not found the optimal hyperparameters for our method yet. There are many directions to explore still, and our results give a direction of where to look: we should make a more conscious decision about when to rewire and how much. Moreover, the task we chose is perhaps not the kind of task where sparsity is most beneficial. A one-hot encoding of 27 symbols is still relatively dense. Predicting words rather than characters, with a vocabulary of e.g. 10,000 words, might be a better fit. What the results do confirm, is that making small changes to the graph of the network is possible without paying too much in loss, if the change is small enough.

CHAPTER 5

Conclusion

In this thesis, we have proposed a novel method of simultaneously optimising the architecture of a sparse neural network, and its parameters. Along the way we formalised networks based on general acyclic graphs that are not necessarily layered. We introduced a new heuristic to estimate the utility of edges in a network, which apart from being integral to our method, has applications in sparsification and quantisation of neural networks. We introduced a criterion for inserting edges in a sparse network, and we showed how combining all of these relates to the geometric problem of nearest neighbour search, and the graph problem of online topological ordering. Finally, we implemented the technique and evaluated its performance on a character prediction task.

5.1 Discussion and future work

In Section 4.1 we proposed a new heuristic for estimating edge utility. Our initial investigation shows that the heuristic outperforms simpler heuristics such as absolute weight, and more sophisticated heuristics such as *optimal brain damage*, when applied to the *Tensorflow deep* MNIST model. One of the key observations was that network parameters are sufficiently far from an optimum even after training. A first-order approximation of the loss is valid, and better than a second-order approximation with first term omitted. It remains to be seen how these results generalise to other networks, and what happens when techniques such as dropout are applied. On the one hand, dropout is supposed to reduce reliance on single neurons, thereby evening out the utility distribution. On the other hand, dropout is a regularisation technique that prevents overfitting, and so it explicitly avoids a minimum of the loss on the test set to achieve better generalisation. Investigating how our utility heuristic performs on other networks and in combination with regularisation techniques, would be an interesting step for future work.

As an aside to the utility heuristic in Section 4.1, we hinted at applications in quantisation

in Section 4.2. Doing a full analysis of how the method stacks up to other quantisation objectives (such as minimising the ℓ_2 norm of the difference in weights, rather than minimising the estimated change in loss) is beyond the scope of this thesis. Improving quantisation methods would be valuable in its own right.

In Section 4.3 we proposed a way to select candidate edges to insert based on correlation between neuron error and output. In Section 4.4 we determined how to select a weight for new edges. We offered an explanation of why adding the proposed edges makes sense in the short term, but in Section 4.6 we show that — in our current implementation — rewiring does not find a better network architecture than the initial architecture. The ineffectiveness of our method can have multiple reasons, and one of them could be that the proposed candidate edges are not the right edges to connect. Further investigation might be done to validate this hypothesis. For example, one could sparsify a dense network, delete a few edges, and see if our method proposes to insert the deleted edges again. If it does so, the method at least takes steps towards a local optimum. More generally, investigating different ways to propose candidate edges would be an interesting line of research for future work.

In Section 4.6 we presented the results of our method when applied to a character prediction task. In this case, our method to learn network architecture was unabled to improve upon a static network architecture. However, we only explored a small portion of the hyperparameter space, and perhaps the task we chose is not the kind of task where sparsity is most beneficial. Furthermore, more work is needed to decide when to rewire and when to *stop* rewiring, and to decide how many edges to rewire. In its current form our method of rewiring edges does not outperform static networks, but future research could change this. We confirmed that making small changes to the graph of the network enables optimisation with a warm start: the parameters are closer to an optimum than random parameters.

To conclude, in this thesis we presented a technique that — given a budget of neurons and edges — optimises a network to perform a given task. More work is required to make our approach competitive, but along the way we had some interesting results that have applications in sparsification and quantisation. In general, optimising network architecture and parameters simultaneously is an area that has received little attention, and it would be worth exploring further.

Acknowledgements

In addition to my supervisors who guided me throughout this project, Maarten Löffler at Utrecht University and Jyrki Alakuijala at Google, this project would not have been

possible without the help of many others. In particular, I would like to thank Jan Wassenberg for his C++ reviews; Krzysztof Potempa for his algorithms expertise, and for reviewing early drafts of this thesis; Robert Obryk for countless valuable discussions, his knowledge of just about any topic imaginable, and his experience with Google infrastructure; Thomas Fischbacher for his Python reviews and his experience with Google infrastructure; and Zoltan Szabadka for continuing to focus on quantifiable results.

Bibliography

Aroa, Sanjeev, Aditya Bhaskara, Rong Ge, and Tengyu Ma (Oct. 23, 2013). "Provable Bounds for Learning Some Deep Representations". In: arXiv: 1310.6343 [cs.LG].

Bender, Michael A., Jeremy T. Fineman, Seth Gilbert, and Robert Endre Tarjan (Dec. 4, 2011). "A New Approach to Incremental Cycle Detection and Related Problems". In: arXiv: 1112.0784 [cs.DS].

Bentley, Jon Louis (1975). "Multidimensional Binary Search Trees Used for Associative Searching". In: *Communications of the ACM* 18.9, pages 509–517. DOI: 10.1145/361002.361007.

Brock, Andrew, Theodore Lim, J.M. Ritchie, and Nick Weston (Aug. 17, 2017). "Smash: One-Shot Model Architecture Search through Hypernetworks". In: arXiv: 1708.05344 [cs.LG].

Choromanska, Anna, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, and Yann LeCun (Nov. 30, 2014). "The Loss Surfaces of Multilayer Networks". In: arXiv: 1412.0233 [cs.LG].

Christiani, Tobias (Aug. 25, 2017). "Fast Locality-Sensitive Hashing for Approximate Near Neighbor Search". In: arXiv: 1708.07586 [cs.DS].

Cireşan, Dan, Ueli Meier, and Jürgen Schmidhuber (Feb. 13, 2012). "Multi-column Deep Neural Networks for Image Classification". In: arXiv: 1202.2745 [cs.CV].

Clarkson, Kenneth Lee (1999). "Nearest Neighbour Queries in Metric Spaces". In: *Discrete & Computational Geometry* 22.1, pages 63–93. DOI: 10.1007/pl00009449.

Datar, Mayur, Nicoleand Immorlica, Piotr Indyk, and Vahab S. Mirrokni (2004).

"Locality-sensitive Hashing Scheme Based on p-Stable Distributions". In: *Proceedings of the Twentieth Annual Symposium on Computational Geometry*. ACM, pages 253–262. ISBN: 1-58113-885-7. DOI: 10.1145/997817.997857.

Denton, Emily, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus (Apr. 2, 2014). "Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation". In: arXiv: 1404.0736 [cs.CV].

Dong, Xin, Shangyu Chen, and Sinno Jialin Pan (May 22, 2017). "Learning to Prune Deep Neural Networks via Layer-wise Optimal Brain Surgeon". In: arXiv: 1705.07565 [cs.NE].

Friedman, Jerome H., Jon Louis Bentley, and Raphael Ari Finkel (1977). "An Algorithm for Finding Best Matches in Logarithmic Expected Time". In: *ACM Transactions on Mathematical Software* 3.3, pages 209–226. DOI: 10.1145/355744.355745.

Haeupler, Bernhard, Telikepalli Kavitha, Rogers Mathew, Siddhartha Sen, and Robert Endre Tarjan (May 12, 2011). "Incremental Cycle Detection, Topological Ordering, and Strong Component Maintenance". In: arXiv: 1105.2397 [cs.DS].

Han, Song, Huizi Mao, and William J. Dally (Oct. 1, 2015). "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding". In: arXiv: 1510.00149 [cs.NE].

Har-Peled, Sariel, Piotr Indyk, and Rajeev Motwani (2012). "Approximate Nearest Neigbors: Towards Removing the Curse of Dimensionality". In: *Theory of Computing* 8.14, pages 321–350. DOI: 10.4086/toc.2012.v008a014.

Hassibi, Babak and David G. Stork (1993). "Second Order Derivatives for Network Pruning: Optimal Brain Surgeon". In: *Advances in Neural Information Processing Systems 5*. Edited by S. J. Hanson, J. D. Cowan, and C. L. Giles. Morgan-Kaufmann, pages 164–171. ISBN: 1-55860-274-7.

Hubara, Itay, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio (Sept. 22, 2016). "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations". In: arXiv: 1609.07061 [cs.NE].

Indyk, Piotr and Rajeev Motwani (1988). "Approximate Nearest Neigbors: Towards Removing the Curse of Dimensionality". In: *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*. ACM, pages 604–613. ISBN: 0-89791-962-9. DOI: 10.1145/276698.276876.

Jain, Prateek, Ambuj Tewari, and Purushottam Kar (Oct. 20, 2014). "On Iterative Hard Thresholding Methods for High-Dimensional M-Estimation". In: arXiv: 1410.5137 [cs.LG].

Kingma, Diederik P. and Jimmy Ba (Dec. 22, 2014). "Adam: A Method for Stochastic Optimization". In: arXiv: 1412.6980 [cs.LG].

Knuth, Donald Ervin (1997). *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. 3rd edition. Addison Wesley. ISBN: 0-201-89683-4.

LeCun, Yann (1987). "Modèles Connexionnistes de l'Apprentissage". PhD thesis. Université Pierre-et-Marie-Curie (Paris 6).

LeCun, Yann, John S. Denker, and Sara A. Solla (1990). "Optimal Brain Damage". In: *Advances in Neural Information Processing Systems 2*. Edited by D. S. Touretzky. Morgan-Kaufmann, pages 598–605. ISBN: 1-55860-100-7.

Li, Ping, Michael Mitzenmacher, and Anshumali Shrivastava (Mar. 31, 2014). "Coding for Random Projections and Approximate Near Neighbor Search". In: arXiv: 1403.8144 [cs.LG].

Mazumder, Rahul, Jerome H. Friedman, and Trevor Hastie (2011). "SparseNet: Coordinate Descent with Non-Convex Penalties". In: *Journal of the American Statistical Association* 106.495, pages 1125–1138.

Meiser, Stefan (1988). "Point Location in Arrangements". In: *Computational Geometry and its Applicataions*. Edited by Hartmut Noltemeier. Springer, pages 71–84. ISBN: 978-3-540-45975-0. DOI: 10.1007/3-540-50335-8_25.

Meiser, Stefan (1993). "Point Location in Arrangements of Hyperplanes". In: *Information and Computation* 106.2, pages 286–303. DOI: 10.1006/inco.1993.1057.

Panigrahy, Rina (2008). "An Improved Algorithm Finding Nearest Neighbor Using Kd-trees". In: *Latin American Symposium on Theoretical Informatics*. Edited by Eduardo Sany Laber, Claudson Bornstein, Loana Tito Nogueira, and Luerbio Faria. Springer, pages 387–398. ISBN: 978-3-540-78772-3.

Tibshirani, Robert (1996). "Regression Shrinkage and Selection via the Lasso". In: *Journal of the Royal Statistical Society* 58.1, pages 267–288.

Tibshirani, Robert (2011). "Regression Shrinkage and Selection via the Lasso: A Retrospective". In: *Journal of the Royal Statistical Society* 73.3, pages 273–282.

Xie, Lingxi and Alan Yuille (Mar. 4, 2017). "Genetic CNN". In: arXiv: 1703.01513 [cs.cv].

Zoph, Barret and Quoc V. Le (Nov. 5, 2016). "Neural Architecture Search with Reinforcement Learning". In: arXiv: 1611.01578 [cs.lg].

Zoph, Barret, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le (July 21, 2017). "Learning Transferable Architectures for Scalable Image Recognition". In: arXiv: 1707.07012 [cs.cv].