DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Data Engineering and Analytics

An Empirical Investigation of the Failure Mode of Training in Mildly Over-parameterized Neural Networks

Theodor Stoican

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Data Engineering and Analytics

An Empirical Investigation of the Failure Mode of Training in Mildly Over-parameterized Neural Networks

Eine empirische Untersuchung der Versagensart des Trainings in leicht überparametrisierten neuronalen Netzen

Author:	Theodor Stoican
Supervisor:	Berfin Şimşek
Advisor:	Martin Bichler
Submission Date:	1 September 2022

I confirm that this master's thesis in data engineering and analytics is my own work and I have documented all sources and material used.

Munich, 1 September 2022 Theodor Stoican

Abstract

Despite the success of modern deep learning, not much is known about the magic behind the optimizers which are widely used to minimize their objective loss. Paradoxically, even though the landscape is highly non-convex, the optimizers still manage to find sensible minima which yield good results on validation/test sets. In this work, we attempt to shed some light on the landscape of a common loss function widely used in modern deep learning optimization. Some statistics are provided in terms of the nature of critical points in which gradient-descent methods are likely to end up. All these aspects are studied in a certain regime, in which a network is mildly over-parameterized, i.e. a network a little larger than the one that is capable of achieving zero-loss on a certain data set. Moreover, we show that the network can get stuck in symmetry-induced critical points (i.e. critical points that originate in a sub-parameterized network) and that there are other local minima close to such points. We validate these results theoretically and we analyze them on two setups: one in which we have a toy data set and one based on the MNIST data set.

Kurzfassung

Trotz des Erfolgs des modernen Deep Learning ist nicht viel über die Magie hinter den Optimierern bekannt, die weit verbreitet sind, um ihren objektiven Verlust zu minimieren. Paradoxerweise schaffen es die Optimierer, obwohl die Landschaft hochgradig nicht-konvex ist, immer noch vernünftige Minima zu finden, die gute Ergebnisse bei Validierungs-/Testsätzen liefern. In dieser Arbeit versuchen wir, etwas Licht in die Landschaft einer verbreiteten Verlustfunktion zu bringen, die in der modernen Deep-Learning-Optimierung weit verbreitet ist. Einige Statistiken werden in Bezug auf die Art kritischer Punkte bereitgestellt, an denen Gradientenabstiegsverfahren wahrscheinlich enden. Alle diese Aspekte werden in einem bestimmten Regime untersucht, in dem ein Netzwerk leicht überparametrisiert ist, d. h. ein Netzwerk, das etwas größer ist als dasjenige, das in der Lage ist, bei einem bestimmten Datensatz Null-Verlust zu erreichen. Darüber hinaus zeigen wir, dass das Netzwerk an symmetrieinduzierten kritischen Punkten (d. h. kritischen Punkten, die aus einem subparametrisierten Netzwerk stammen) stecken bleiben kann und dass es andere lokale Minima in der Nähe solcher Punkte gibt. Wir validieren diese Ergebnisse theoretisch und analysieren sie auf zwei Setups: einem, in dem wir einen Spielzeugdatensatz haben, und einem, der auf dem MNIST-Datensatz basiert.

Contents

1 Introduction

The historical context in neural networks optimization is centered on stochastic gradient descent ([1]) and its well-known varieties, such as Adam ([2]). However, this is in an apparent contradiction with the highly non-convex objective functions that are present in modern deep learning. The most popular optimization methods mentioned above provide convergence guarantees for convex functions only. Still, the very good results that modern deep learning has yielded raises again the question of the optimality of such methods. In what kind of points do these algorithms get stuck? And, more extensively, how does the landscape of the objective function look like? Are there, say, many global minima and less frequent local minima, such that optimization is easy?

More recent developments have shown that the loss landscape does vary depending on the architecture of the network, the hyper-parameters used, and potentially other factors too. Li et al. (2018) have shown in [3] in what way skip connections, for example, change the structure of the loss landscape such that training becomes easier. Jacot et al. outline in ([4]) that wide neural networks have a smoother landscape that leads to faster convergence. The architecture of neural networks seemingly affects convergence in various ways. In [5], Safran et al. (2018) show that spurious local minima are less common in mildly over-parameterized networks (i.e. networks that are wide but not that wide). In this setting, the better understanding of the landscape leads to a better understanding of the models themselves and of the factors that drive their design.

In this work, we attempt to study the nature of the points in which common optimization algorithms get stuck in the mildly over-parameterized regime. We want to understand both the frequency of ending in non-global critical points as well as their nature. Are they local minima, saddles, or perhaps something else? We use common deep learning optimization algorithms and neural network architectures that occur frequently in practice, as well as common software tools that incorporate them.

2 Artificial Neural Networks

Artificial Neural Networks have provided in recent years many innovations in various fields, which had been traditionally untouchable. The main advantage od neural networks is their undoubtedly efficient capacity to learn data sets of various types and produce accurate results on similar test data. In the following part, we will discuss some of the fundamental aspects that underpin modern Artificial Neural Networks.

2.1 Multi-Layer Perceptron

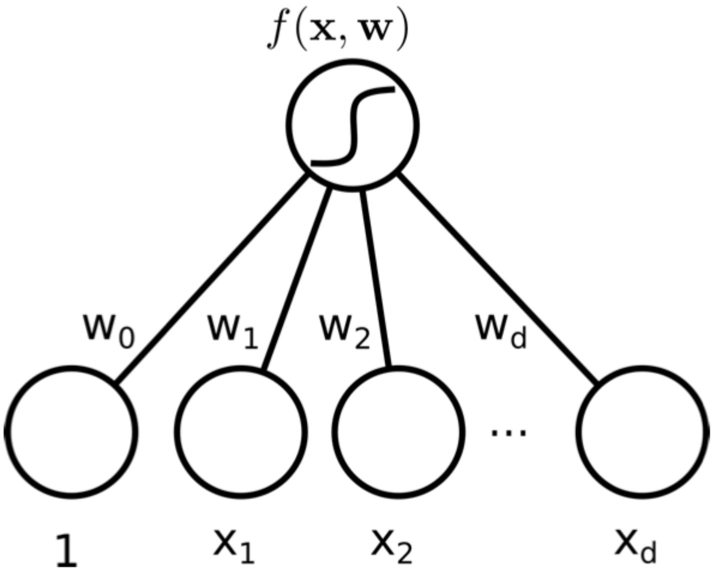


Figure 2.1: An instance of logistic regression as a neural network.

To begin with, let us start with the well-known model of logistic regression. Simply put, based on a D-dimensional input (and 2 classes), logistic regression models the distribution:

$$y|x \sim Bernoulli(\sigma(w^T x))$$

where

$$w^T x = w_0 + w_1 x_1 + w_2 x_2 + ... + w_D x_D$$

Furthermore, one can notice that logistic regression is a special case of a neural network, with 1 layer only.

Logistic regression works excellently if the data that is being learned is linearly separable. However, there are data sets which do not fulfill this condition and, hence, learning becomes more challenging. One such well-known example is the XOR data set (Figure 2.2).

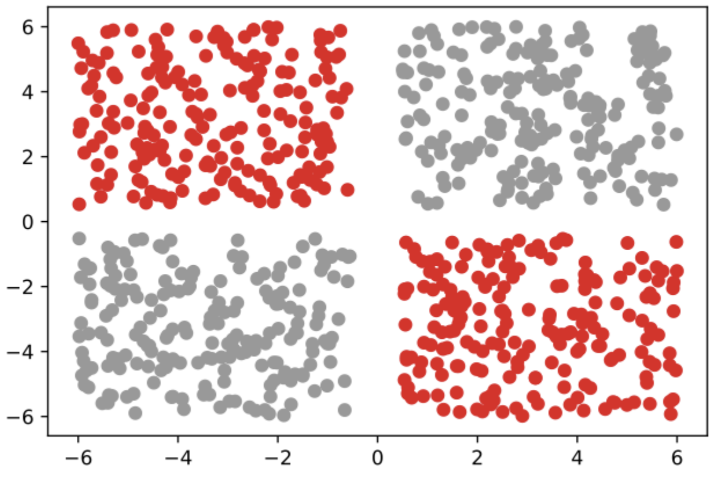


Figure 2.2: An instance of the XOR data set, which is not linearly separable. [1]

One elegant way to overcome the issues posed by the XOR data set is to use basis functions. By using basis functions, one can hope to transform the data set into a linearly separable one. That means that instead of applying $\sigma(w^T x)$, one can apply $\sigma(w^T \phi(x))$, where ϕ is a basis function. For the XOR data set, in particular, one could obtain a linearly separable data set, as the one in Figure 2.3. Such a basis function may be:

$$\phi(x) = \phi(1, x_1, x_2) = (\sigma(5 + x_1 + x_2), \sigma(5 - x_1 - x_2))$$

where 1 was added as the first argument of the basis function, to account for the bias added to the input. By using this basis function, the dataset is linearly separable (Figure 2.3).

Now, the question that automatically shows up is how exactly can one find this new space

[1]Figure taken from the slides of the course *Machine Learning* from TUM.

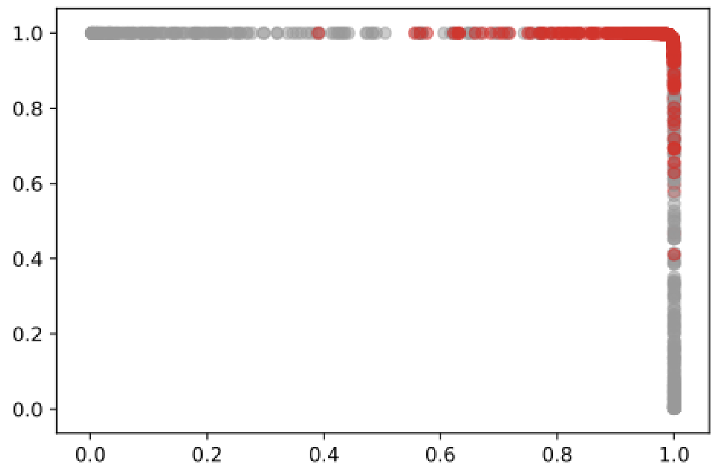


Figure 2.3: The XOR dataset after applying a basis function [2]

where the dataset is linearly separable in a general way? Well, one intuition of doing that would be to create a more complex boundary by using multiple linear boundaries. This can be simulated by adding multiple processing units, as follows:

$$\begin{gathered} w_1^T x = w_0^1 + w_1^1 x_1 + w_2^1 x_2 + ... + w_D^1 x_D \\ w_2^T x = w_0^2 + w_1^2 x_1 + w_2^2 x_2 + ... + w_D^2 x_D \\ ... \end{gathered}$$

This would also be analogous to adding multiple neurons to a neural network. At this point, we would ideally like a new space in which one point on one side of the decision boundary that we create is situated as far away as possible from the boundary that will be created in the new space, while, conversely, the points on the other side of the decision boundary should be situated as far as possible away in the opposite direction, such that a line can separate the sets of points easily. The sigmoid function does exactly that: it pushes the points on one side of the boundary towards 1 and the points on the other side towards 0.

However, when we have a composition of boundaries (as in the case above), we'll have multiple dimensions in the new space. Namely, we'll have as many dimensions as there are neuron units. The basis function for our data set would looks as follows:

[2]Figure taken from the slides of the course *Machine Learning* from TUM.

$$\phi : \mathbb{R}^2 - > \mathbb{R}^n$$

where n - the number of neurons or the number of boundaries that are added and the input is assumed to be $2D$.

In this case, the set of points belonging to class + (or any other label) would go towards 1 in at least one of the resulting dimensions, whereas the set of points belonging to class - (or any other label) would go mostly towards 0 across a multiple dimensions, hopefully leading to a straightforward linear separation in the new feature space.

Now, the last linear separation that was mentioned is translated architecturally by adding one output neuron to the network. Hence, we've obtained a neural network with 1 hidden layer and 1 output layer, also called a multi-layer perceptron (Figure 2.4).

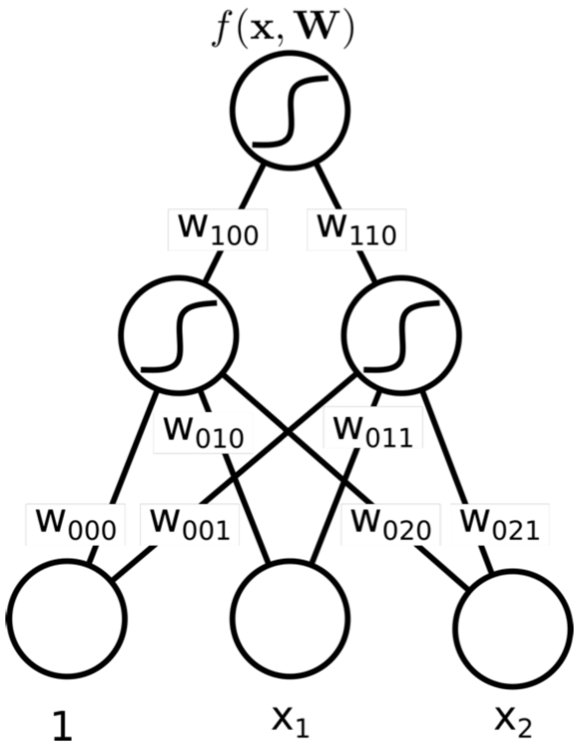


Figure 2.4: Multi-layer perceptron with 1 hidden layer and 1 output layer [3].

[3]Figure taken from the slides of the course *Machine Learning* from TUM.

2.2 Deep Neural Networks

Theoretically, neural networks with 1 hidden layer are able to learn any function. That is due to the Universal Approximation Theorem ([6]). The bad news is that the theorem does not give any hint into how one can find (e.g.) the number of neurons the hidden layer should have. Hence, one solution that has been adopted is to create neural networks with multiple hidden layers. The reason for that is two-fold:

- *a theoretical one*: for some families of functions, using only one layer would lead to a huge amount of neurons in that layer, making the number of parameters explode
- *a practical one*: deeper networks (with multiple layers) turn out to train faster and generalize better

Furthermore, deep learning neural networks can learn a hierarchy of representations.

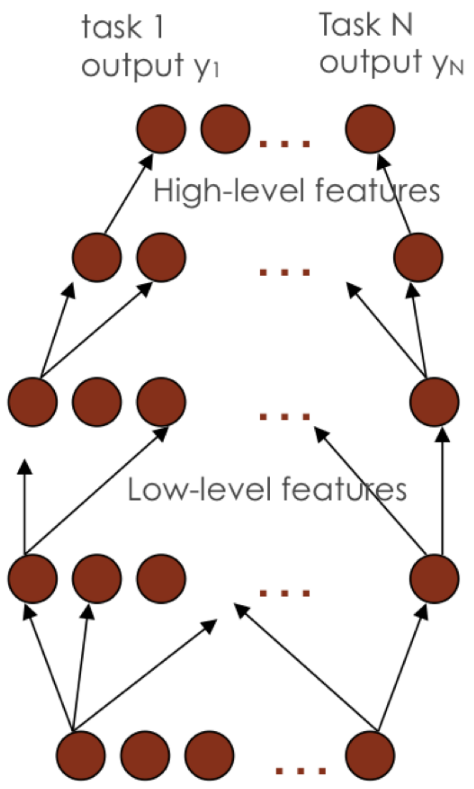


Figure 2.5: A DNN representation, with the low and high-level features it learns [4]

[4]Figure taken from : *Understanding and Improving Deep Learning Algorithms, Yoshua Bengio, ML Google Distinguished Lecture, 2010*

3 Optimization

3.1 Gradient Descent

In the realm of optimization for machine learning, there is perhaps one single algorithm which stands out: gradient descent. The algorithm is virtually ubiquitous and any current optimizer is essentially either vanilla gradient descent or an add-on to the vanilla version. Gradient descent can be stated in a very straightforward fashion:

$$x_{t+1} = x_t - \alpha \cdot \nabla f(x_t)$$

More explicitly, it moves the parameters in the direction of the negative gradient with a step size of α. Were the optimization problem convex, the algorithm would almost always converge (depending on the step size) to the global minimum. It is the non-convex nature of neural network loss functions that poses a rather great challenge. Here, convergence to the global minimum is rather problematic, due to the fact that there are multiple other stationary points, where $\nabla f(x) = 0$, and which can be either one of these:

- local minima
- local maxima
- saddles

3.2 Minima and Maxima

The difference between maxima and minima can be verified analytically, with the help of the second derivative. That is because the second derivative characterizes the curvature of the function (or the rate at which the slope of the function changes). The second derivative is positive if the function curves upwards and conversely it is negative if the function curves downwards.

[1]Figure taken from `https://www.usna.edu/Users/oceano/raylee/SM223/Ch14_7_Stewart%282016%29.pdf`

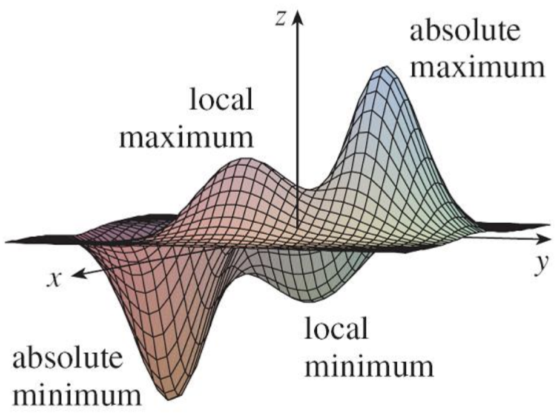


Figure 3.1: Example of local and global (absolute) minima/maxima[1].

In a 1D setting, a point with $\nabla f(x) = 0$ is a minimum iff $f''(x) > 0$. Similarly, it is a maximum, iff $f''(x) < 0$. In a 2D setting, the equivalent of the second derivative is the Hessian matrix.

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} \end{bmatrix}$$

The reason why we need the Hessian matrix is because we have multiple directions around the point, unlike in the 1D case where we have only one direction. The Hessian matrix characterizes how the second order derivative behaves in the direction of x_1, in the direction of x_2 and in the direction of both.

In order to ensure convexity, let us look at the Taylor expansion at a point y near x:

$$f(y) = f(x) + \nabla f(x) \cdot (y - x) + \frac{1}{2!} \cdot (y - x)^T \cdot H \cdot (y - x)$$

Informally, f is convex if, for every y in the vicinity of x, $f(y) \geq f(x)$. Furthermore, we know that $\nabla f(x) \cdot (y - x) = 0$, since x is either a local/global minimum or maximum. Therefore, to ensure convexity, one needs to ensure $(y - x)^T \cdot H \cdot (y - x) \geq 0, \forall y$. This is equivalent to ensuring that $u^T \cdot H \cdot u, \forall u$, a condition which is called **positive definiteness** in linear algebra.

Now, we can say that a point $x = (x_1, ..., x_n)$ of a multivariate function is a minimum iff:

- $\nabla f'(x) = 0$

- H is positive definite

Similarly, a point $x = (x_1, ..., x_n)$ is a maximum iff:

- $\nabla f'(x) = 0$
- H is negative definite

3.3 Saddles

Based on the nature of the Hessian matrix, there are other types of points in the landscape of a function. One such type is represented by **strict saddles**. Such a point very much resembles a horse saddle, since the function is convex in one direction and concave in another.

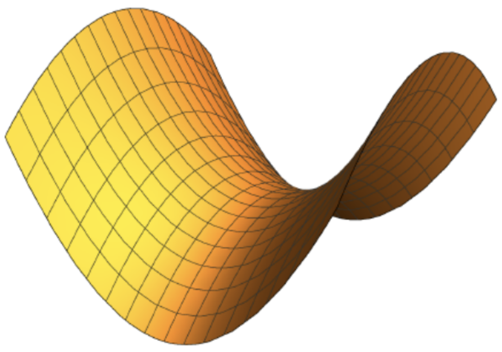


Figure 3.2: Example of a strict saddle point.[2].

For a point to be a strict saddle, the zero gradient ($\nabla f'(x) = 0$) condition must still stand and, further, there must be a direction vector u for which $u^T \cdot H \cdot u < 0$. That means that for a certain y from the Taylor expansion, it must be the case that $f(y) < f(x)$, which means that the function will curve downwards, making it concave. Similarly, there must be another direction u for which the function is convex, i.e. $u^T \cdot H \cdot u > 0$.

In addition to this, there is an even more direct way to characterize strict saddle points. When we demand that there is a u such that $u^T \cdot H \cdot u < 0$, we actually demand that the smallest eigenvalue of the hessian is negative. Why is that so? If one rephrases the equation that identifies eigenvalues ($Ax = \lambda x$) by multiplying with x on the left-hand side, one can obtain the condition aforementioned. Namely:

[2]Figure taken from `http://www.offconvex.org/2017/07/19/saddle-efficiency/`

$$x^T \cdot |Ax = \lambda x$$
$$x^T Ax = \lambda$$

That being said, minimizing $x^T Ax$ is equivalent to finding λ_{min}. Moreover, if for our point it holds that $\lambda_{min} < 0$, there are no zero eigenvalues, and there is at least one positive eigenvalue, our point is a strict saddle. Alternatively, if $\lambda_{min} = 0$, the point is considered a non-strict saddle.

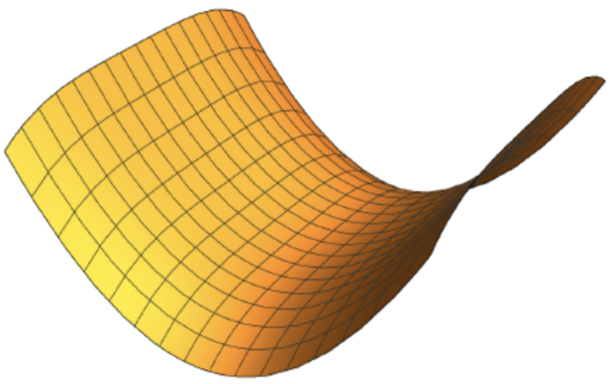


Figure 3.3: Example of a non-strict saddle point.[3].

3.4 First-order optimization

Modern neural network optimization is based on gradient descent. However, in order to speed up and overcome various limitations of vanilla gradient descent, different updated versions of the algorithm have been released. One of the most well-known versions of the algorithm is Adam. Adam is a combination of 2 well-known algorithms, both based on gradient descent:

- Gradient descent with momentum
- RMSProp

Both these algorithms attempt to remove unnecessary oscillations, which are present in the standard gradient descent algorithm (Figure 3.4).

[3]Figure taken from `http://www.offconvex.org/2017/07/19/saddle-efficiency/`

Firstly, momentum is a mechanism that takes into account the gradients of the past iterations (using an exponential moving average) in order to increase the current value of the gradient (if the past values have been predominantly large and the current value may be low) or to decrease it (if the past values have been small and the current value is large). It essentially smoothens out the update of the weights by making an analogy between the high-dimensional point (represented by the current weights) and the movement of a ball on a high-dimensional surface. More specifically:

$$w = w - \alpha \cdot V_{dw}$$

$V_d w$, in this case, is the exponential moving average of the past gradients:

$$V_{dw} = \beta_1 \cdot V_{dw} + (1 - \beta_1) \cdot dw$$

Secondly, RMSProp provides a way to damp down oscillations (like those on the w axis in Figure 3.4) and, at the same time, to increase the stable gradients to the minimum (on the b axis in Figure 3.4). The way it does this is by updating the weight with a ratio between the current gradient and the square root of an exponential moving average of the squares of previous gradients. In a nutshell:

$$w = w - \alpha \cdot \frac{dw}{\sqrt{S_{dw}}}$$

S_{dw} is exponential moving average that we mentioned above and can be computed as:

$$S_{dw} = \beta_2 \cdot S_{dw} + (1 - \beta_2) \cdot dw^2$$

At the end of the day, what Adam does is to combine GD with momentum and RMSProp, the intuition being that we want keep the gradients smooth, without oscillation (by removing the behavior from Figure 3.4) and, at the same time, speed up the gradients in case we reach a flat plateau after a steep slope. Thus, the update becomes:

$$w = w - \alpha \cdot \frac{V_{dw}}{\sqrt{S_{dw}}}$$

.

In practice, though, V_{dw} and S_{dw} are bias-corrected (to account for their initial values) and, for numerical stability, an ϵ is added to the denominator. The update becomes:

$$w = w - \alpha \cdot \frac{V_{dw}^{corr}}{\sqrt{S_{dw}} + \epsilon}$$

.

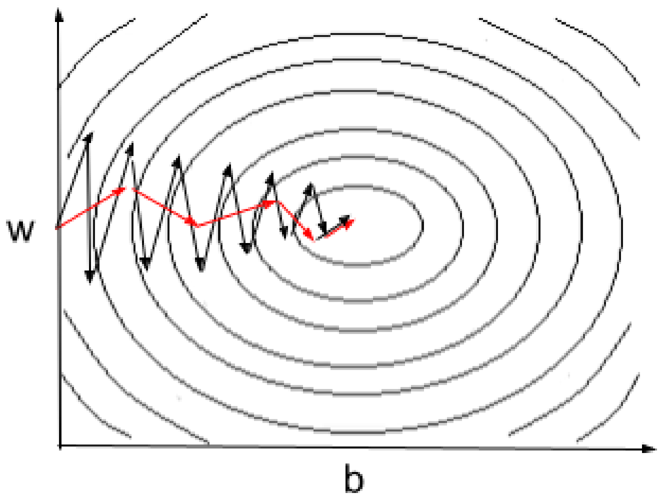


Figure 3.4: **Black**: Vanilla gradient descent, presenting high oscillations on the w axis and slow progress on the b axis. **Red**: Accelerated gradient descent via RMSProp/Momentum by damping down the gradients on the w axis and increasing them on the b axis.

3.5 Second-order optimization

One of the most well-know second-order optimization algorithms is Newton's method. Newton's method is traditionally used for finding the roots of a equation of the type $f(x) = 0$. However, in the context of finding a local minimum/maximum, one could use it in order to find the roots of $f'(x) = 0$ instead. Newton's method relies on assuming that the surface of the function around the critical point is a parable and that, by examining the first derivative of f, one could make a jump towards the minimum by following the tangent at the current point.

To see this process in an exemplified fashion, one could look at Figure 3.5. One wants to reach the point where $f(x) = 0$, i.e. the point where the function touches the x axis. For that, the direction of the tangent at x_0 is followed (which points towards the minimum direction). According to the equation of a tangent:

$$0 - f(x_0) = f'(x_0)(x_1 - x_0)$$

At this point, one is interested in the value of x_1:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

From this point one, one can apply the same process from x_1 in the same way as from x_0, until the desired point is reached.

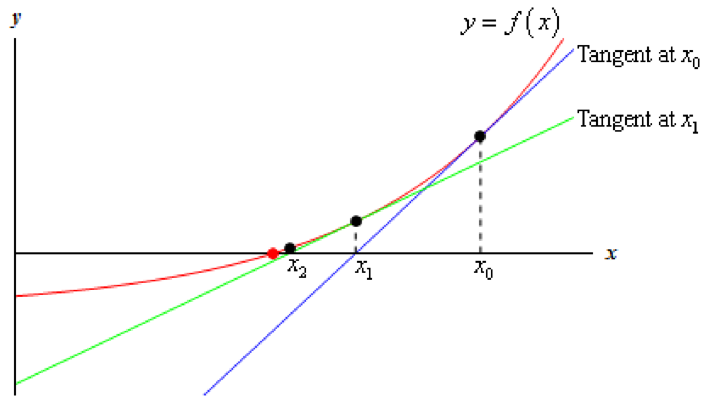


Figure 3.5: Example of applying Newton's method. x_2 is the minimum and one tries to get it by following the tangent at x_0. One reaches x_1 though, due to the imperfect approximation of the tangent and carries on from there. [4]

We have mentioned before that one could use this method in order to find the local minimum/maximum as well. So far we've only found the solution to $f(x) = 0$, but since critical points are defined by $f'(x) = 0$, one can adapt the Newton's method to finding the root(s) of this equation:

$$x_1 = x_0 - \frac{f'(x_0)}{f''(x_0)}$$

Naturally, this method can be adapted to high-dimensional cases as well, in the following way:

$$x = x_0 - H^{-1} \cdot J$$

H, J are the Hessian and the Jacobian, respectively, at x_0. The problem with this approach, however, is computational. Computing the Hessian in a high-dimensional scenario is not

[4]Figure taken from `https://tutorial.math.lamar.edu/classes/calci/newtonsmethod.aspx`

trivial. Therefore, in a typical setting, the BFGS (Broyden–Fletcher–Goldfarb–Shanno) method is used. This method computes the Hessian iteratively, based on the previous value of the Hessian.

The intuition behind this method lies in understanding how the Hessian changes in the direction of one step $\Delta = x_1 - x_0$. In order to understand how the Hessian changes, one can follow the direction of the Hessian along Δ and see how that differs from the actual change in f'. In particular, $f'(x_1) - f'(x_0) - H_0 \cdot \Delta$ gives the missing part in the Hessian than one should have in order to approximate H_1 (where we have noted H_0 - the Hessian at x_0 and H_1 - the Hessian at x_1). All in all, the new Hessian can be obtained with the following formula (in the multi-dimensional case):

$$H_1 = H_0 + \frac{f'(x_1) - f'(x_0) - H_0 \cdot \Delta}{||\Delta||^2} \cdot \Delta^T$$

The process is iterative and it unfolds similarly for a general H_n.

In a general second order optimization routine one may need equality and inequality constraints too. **LD_SLSQP** (local, gradient-based, Sequential Least Squares Programming) is such a method which is applicable for certain objective functions. It makes use of the Lagrangian in order to integrate the constraints and apply BFGS updates on the new objective function.

4 Loss Landscape

The loss landscape of a neural network has fundamental implications in training modern deep learning models and in finding good minima that can yield good results at inference. The loss function is fundamentally non-convex and is abundant in the amount of saddle points, local and global minima.

Recent deep learning models have proved the success of constructing models with huge amounts of parameters. This has been shown to be due to the nature of the landscape of the loss functions. Recent work ([7]) has shown that over-parameterization can lead to an increase in the global minima and a decrease in the number of local minima. That can be translated into a decrease in the roughness of the landscape (Figure 4.1).

4.1 Setup

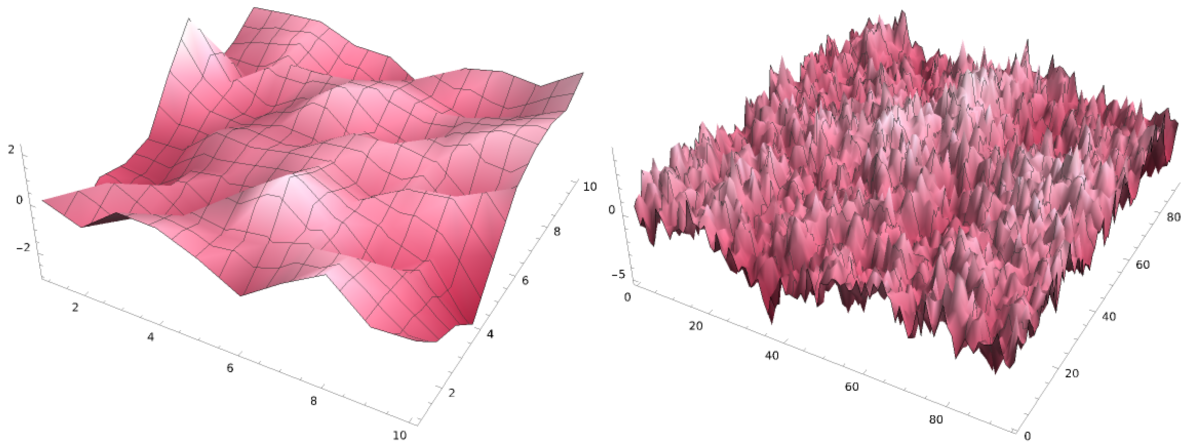
In order to help argue in favor of this claim, let us exemplify our statement based on an actual neural network. Hence, assume we have a network like the one in Figure 4.2, with one hidden layer and a width of 4. The network is fully connected, receives a 2-dimensional input, and spits out a 1-dimensional output. Therefore, each neuron in the hidden layer has 2 incoming weights and one outgoing weight. Moreover, each one uses the sigmoid activation function and the loss function of the model can be defined as:

$$L^4 = \sum_{i=1}^{4} a_i \cdot \sigma(w_i \cdot x)$$

,

where w_i - incoming weight of a neuron,

a_i - outgoing weight of a neuron, and x - the input.



(a) Potential loss landscape for a neural network with a small number of parameters.

(b) Potential loss landscape for a neural network with a large number of parameters.

Figure 4.1: The (general) effect that occurs by increasing the number of parameters of a model (generated with code from [8]).

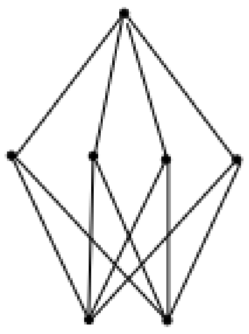


Figure 4.2: A neural network with 1 hidden layer and a width of 4.

4.2 The Global Minima Manifold

Now, let us get back to the original question we wanted to answer: why does the number of global minima increase? In order to understand that precisely, let us quantify this number for a certain example.

Let us have a look at the neural network we have just defined in Figure 4.2. According to the previous statements, should one add one neuron to the hidden layer, the number of global minima would increase. Assume we have a global minimum of 4-sized neural network. In the parameter space, the point can be written as:

$$\theta^4 = (w_1, w_2, w_3, w_4, a_1, a_2, a_3, a_4)$$

For further reference, we call this a 4-neuron point. Assume it corresponds to a loss value of 0 and that it is irreducible. What that means is that, one cannot get rid of one of the neurons and preserve the loss as well. It also means that no **zero-type neurons** occurs in this point, i.e. *no neurons whose outgoing weights sum up to 0* are present.

That being said, the claim that we made can be rewritten as follows: if one is to add an additional neuron to θ^4 and obtain an induced point in a 5-sized neural network, like the one in Figure 4.3, one can obtain an entire manifold of global minima in this higher dimensional space.

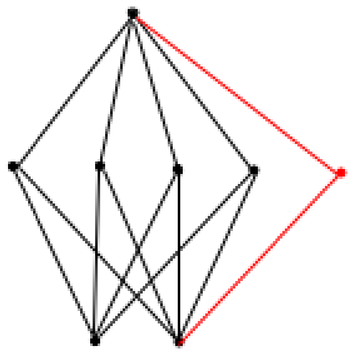


Figure 4.3: A neural network with 1 hidden layer and a width of 5.

Simply put, by duplicating one of the neurons from the 4-sized neural network, one can obtain a 5-neuron point that has the same loss level as the original neural network. Assume now that we duplicate the first neuron (w_1, a_1). One 5-neuron point can be constructed as follows:

$$\theta^5 = (w_1, w_2, w_3, w_4, w_5, \frac{a_1}{2}, \frac{a_1}{2}, a_2, a_3, a_4)$$

The value of the loss function at this new point can be written as:

$$L^5 = \sum_{i=1}^{5} a_i \cdot \sigma(w_i \cdot x) = \frac{a_1}{2} \cdot \sigma(w_1 \cdot x) + \frac{a_1}{2} \cdot \sigma(w_1 \cdot x) + \sum_{i=2}^{4} a_i \cdot \sigma(w_i \cdot x)$$
$$= \sum_{i=1}^{4} a_i \cdot \sigma(w_i \cdot x) = L^4$$

More generally, any manipulation of the outgoing weights of the duplicated neuron (of the type $\theta^5 = (w_1, w_2, w_3, w_4, w_5, \mu \cdot a_1, (1-\mu) \cdot a_1, a_2, a_3, a_4)$) yields 5-neuron points with loss equal to the loss of the 4-neuron point. For this reason, a global minimum in the 4-sized neural network results in a line of global minima points in the 5-sized neural network. Similarly, by duplicating additional neurons, one can obtain additional lines of global minima.

In addition to this, it turns out that by duplicating a neuron, one can generate an entire **affine subspace** of points, all of which are equivalent to the original point. This happens due to the fact that one can set the outgoing weight of the duplicated neuron to 0 and, as a consequence, the incoming weights of that very neuron can take any values. In fact, the affine subspace can be defined as follows:

$$(\underbrace{w_1, ..., w_1}_{k_1}, \underbrace{w_2, ..., w_2}_{k_2}, \underbrace{w_3, ..., w_3}_{k_3}, \underbrace{w_4, ..., w_4}_{k_4}, \underbrace{w', ..., w'}_{b_1},$$
$$a_1^1, ..., a_1^{k_1}, a_2^1, ..., a_2^{k_2}, a_3^1, ..., a_3^{k_3}, a_4^1, ..., a_4^{k_4}, \underbrace{0, ..., 0}_{b_1})|$$
$$\sum_{i=1}^{k_t} a_t^i = a_t, \forall t \in [4],$$
$$b_1 + \sum_{i=1}^{4} k_i = 5, k_i \geq 1, b_1 \geq 0\}$$

Furthermore, the loss of the neural network remains invariant under any permutation of neurons from such a 5-neuron point. More generally, there is a formula that provides the number of global minima induced in a 5-sized neural network from a 4-sized neural network, based on the process mentioned above (based on [7]):

$$T(4,5) = \sum_{\substack{k_1+k_2+k_3+k_4=5 \\ k_i \geq 1}} \binom{5}{k_1, ..., k_4} + 5$$

The first part of the term stands for the total number of permutations that occur by duplicating each of the 4 neurons. The second part (the *5*) stands for the number of perturbations that can occur by adding a neuron with an outgoing weight of 0. Computing this value yields:

$$T(4,5) = 4 \cdot \frac{5!}{2!} + 5 = 245$$

That means that a global minimum in a 4-sized neural network generates a manifold of 245 distinct affine sub-spaces in a 5-sized neural network. This provides a very nice insight into the magic of over-parameterized and into why it is that ubiquitous in virtually all current models.

4.3 The Critical Points Manifold

Similarly, any critical point in the weight space of a 4-sized neural network can generate an entire manifold of connected affine subspaces of critical points. In order to see why, let us have a look of how one can count the generated subspaces of critical points.

Let us assume that we start from an **irreducible** critical point in the parameter space of a neural network of size 3. The reason why we take the critical point from a lower dimensional space is because we want to make sure that the loss at this point is non-zero (assuming that a network of size 3 is not good enough to achieve zero loss in a certain setup, but one of size 4 is capable of that). By a similar procedure to the one used for obtaining the global minima manifold, one can obtain a manifold of distinct affine subspace of critical points, which have all the same loss value as the 3-neuron point and a gradient of 0. Namely, the affine subspace that can be obtained by adding one neuron looks as follows:

$$\{(\underbrace{w_1, ..., w_1}_{k_1}, \underbrace{w_2, ..., w_2}_{k_2}, \underbrace{w_3, ..., w_3}_{k_3}, a_1^1, ..., a_1^{k_1}, a_2^1, ..., a_2^{k_2}, a_3^1, ..., a_3^{k_3}) |$$

$$\sum_{i=1}^{3} k_i = 4,$$

$$\sum_{i=1}^{k_t} a_t^i = a_t, \forall t \in [3]\}$$

In this construction of the subspace, only **non-zero-type neurons** are taken into account, since it is easily shown that they lead to points of gradient 0, like the original critical point.

Furthermore, like in the case of the global minima manifold, the network function is invariant to any permutation of the aforementioned neuron and it is possible to count the total number of affine subspaces that construct the entire manifold which is generated from the 3-neuron point. In particular, the number of distinct affine subspaces obtained by expanding in a 4-sized neural network is (based on the formula from [7]):

$$G(3,4) = \sum_{\substack{k_1+k_2+k_3=4 \\ k_i \geq 1}} \binom{4}{k_1, k_2, k_3}$$

4.4 The Mildly Over-parameterized Regime

Now that we can actually count how many affine subspaces belong to the global minima manifold and how many belong to the critical points manifold, let us see the variation in their number and how they change the more we over-parameterize the network.

$$G(3,4) = 3 \cdot \frac{4!}{2!} = 36$$
$$T(4,4) = 4! = 24$$

What we get is that, by permutation alone, in a neural network of size 4, 24 global minima show up, whereas 36 new affine subspaces originate from a critical point from a neural network of size 3. If we move to a network of size 5:

$$G(3,5) = 3 \cdot \frac{4!}{3!} + 3 \cdot \frac{4!}{2! \cdot 2!} = 30$$
$$T(4,5) = 4 \cdot \frac{5!}{2!} + 5 = 245 = 245$$

Now, is is evident that the number of affine subspaces containing global minima surpasses the number of affine subspaces containing critical points original in an irreducible 3-neuron point. This trend holds true the more we over-parameterize and serves as an argument in favor of the claim that we made at the beginning (over-parameterized networks converge more reliably due to the plethora of global minima that show up).

In addition to this, we have just taken into account the critical points manifold originating in a 3-neuron point. There could be manifolds originating in a 2-neuron point or a 1-neuron point. The trend, however, remains and applies for all situations (Figure 4.4).

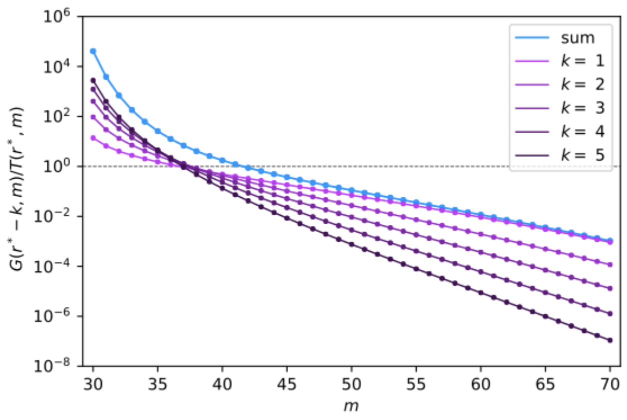


Figure 4.4: The ratio between each G term and T for a minimal network of size $r^* = 30$. The ratio between the sum of all G terms and T is shown in blue ([7]).

5 Prior Work

In this chapter, we will investigate prior and related work which has served as a basis for the content of our work.

5.1 Landscape of Neural Networks

An important amount of work has focused on the landscape of neural networks. Neural networks work primarily in a multi-dimensional space and that brings with it a plethora of challenges. Dauphin et al.(2014) show in [9] that a low-dimensional error function (possibly sampled from a Gaussian process) has an abundant number of local minima and maxima, but a negligible amount of saddles. The curse of dimensionality shows itself here as well, since the ratio of the number of saddle points to local minima increases exponentially with the number of dimensions.

Geometrically. these saddle points are often in the vicinity of plateaus, which slow down gradient descent-based methods considerably. Furthermore, second-order optimization methods (Newton and quasi-Newton methods) also fail to converge, as reported in the paper. They propose an alternative second-order algorithm, named saddle-free Newton method, that aims at leveraging curvature information in order to escape saddles.

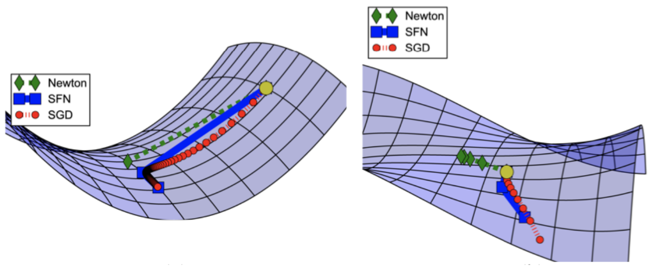


Figure 5.1: Saddle-free Newton, proved to be able to avoid saddles (compared to stochastic gradient descent and Newton method), as reported in [9].

Similarly, in [10], Fort et al.(2019) propose a way of modelling the loss landscape of a deep network. The landscape is modelled as a union of n-dimensional manifolds, called n-wedges, which are intended to emulate the behavior of real high-dimensional loss functions. They exemplify and further analyze the impact hyperparameters have on the loss landscape. Furthermore, the existence of low-loss tunnels that connect two optima on a low loss path is proved and extended to m-tunnels that connect $(m+1)$ optima on a m-dimensional hypersurface.

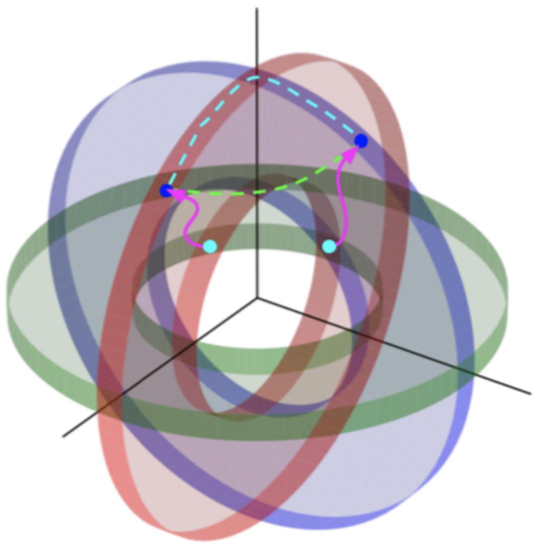


Figure 5.2: A 2-wedges model intended to model a 3-dimensional low-loss manifold, presented by [10]. The intersection of the wedges represents a low-loss connection.

5.2 Gradient Descent - Convergence Analysis

In [4], Jacot et al.(2018) show that wide neural networks in the function space can be modelled with kernel methods. In particular, in an infinite-width regime, a neural network can be described by the limit of the neural tangent kernel, which only depends on depth, non-linearity and initialization. Under certain conditions (one being that the cost function is convex), gradient descent converges invariably to the global minimum.

In [11], Du et al.(2017) outline that the proliferation of saddle points in certain regimes can significantly slow down gradient descent. In particular, it takes an exponential amount of time to escape saddle points in a standard optimization. A new algorithm is proposed that introduces perturbations around saddle points (named *perturbed gradient descent* - Figure 5.3) such that gradient descent may be able to escape.

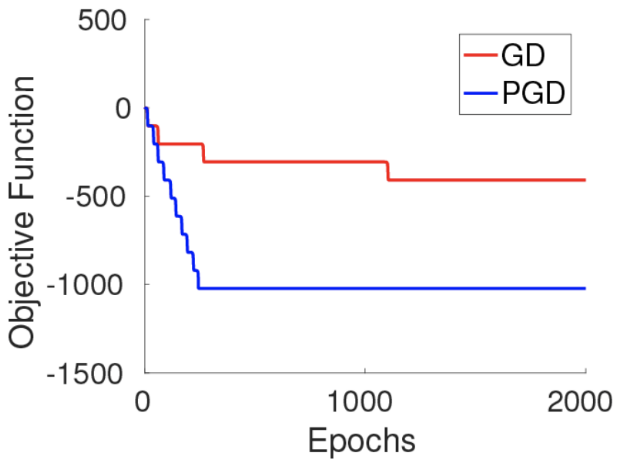


Figure 5.3: Evolution of loss function in a standard setup, by applying both vanilla and perturbed gradient descent ([11]).

Another aspect that has been actively researched is the impact of initialization on convergence. Frankle et al.(2018) have shown in [12] that densely parameterized networks contain a subnetwork that yields the same performance as the network itself. If one were to train the subnetwork from the same set of initialized parameters, one would reach the same loss level and this subnetwork would be called a winning ticket, since it has won the lottery of initialization. However, winning tickets are rather rare in dimensionally lower spaces and adding more parameters naturally eases convergence. If one could devise a method to decrease the number of parameters, but keep the same performance, one could improve the complexity of neural networks both time-wise and complexity-wise.

Therefore, in [12], a pruning algorithm is devised that is able to overcome these limitations and that can accelerate training:

- Randomly initialize the network
- Train it
- Prune a proportion of the parameters
- Reset the remaining parameters to their initial values

The remaining weights will constitute the winning ticket, which is more efficient (both in terms of space and time complexity) and also generalizes better. As a consequence of this observation, one can note that convergence is more reliable in a high-dimensional space than in a lower-dimensional space, which is consistent with previous lines of work.

5.3 Connectivity of Global Minima

In [5], Safran and Shamir (2018) prove the abundance of spurious local minima in a typical neural network setup of the type $x \rightarrow \sum_{i=1}^{k} max\{0, w_i^T x\}$ with respect to the squared loss. Furthermore, they prove that the probability of hitting such a minimum is quite high and that it increases with the network size.

Firstly, they formally prove the existence of local minima around points with a rather small gradient. 1 lemma that showcases this result stands out.

Lemma 1. Assume that $||\nabla F(w_1^n) \leq \epsilon||$, $|R_{w_1^n,u}| \leq B$ for some ϵ, $B > 0$ and all unit vectors u, and let $\lambda_{min} > 0$ denote the smallest eigenvalue of $\nabla^2 F(w_1^n)$. If $9\lambda_{min}^2 - 25B\epsilon \geq 0$, then the function F contains a local minimum, within a distance of at most

$$r = \frac{3\lambda_{min} - \sqrt{9\lambda_{min}^2 - 25B\epsilon}}{2B}$$

from w_1^n.

There is an additional lemma that the authors provide and that checks whether the minima found in this way are also non-global.

Secondly, the authors prove that these theoretically verified spurious local minima are not only possible, but in fact common. The experiments focus mostly on the case in which the network size varies between 6 and 20. On a more positive note, there is a regime for which spurious local minima seem to be more rare, namely the mildly over-parameterized one.

In [13], Frankle et al. (2020) present a way through which the minima that an ANN may find during multiple instances of training are connected. According to the lottery ticket hypothesis (introduced by Frankle et al. (2018) in [12]), one may find a smaller ANN, by selecting a subset of the weights of an over-parameterized network that yields the same performance. If one repeats this process, multiple winning tickets (subnetworks the reach the same accuracy as the original network) are discovered. Is there a connection between these networks?

In order to discover a winning ticket, a pruning algorithm is introduced - IMP (iterative magnitude pruning). The authors show that the two identified subnetworks are connected via a line of equal loss (Figure 5.4). Furthermore, this occurs only when these subnetworks are stable to SGD noise (i.e. randomness in the order of the data during training, data augmentation, etc.), which occurs either at initialization (for small-scale settings) or early in the training process (for large-scale settings).

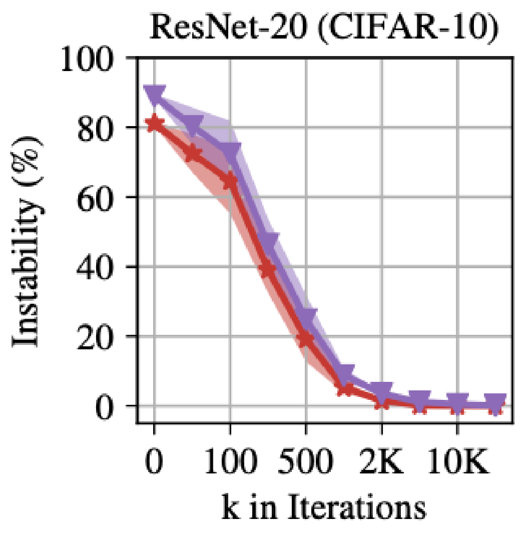
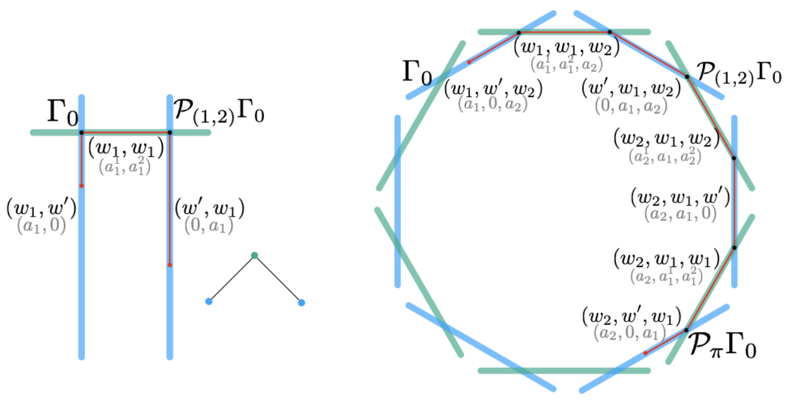


Figure 5.4: Computing the instability between two instances of ResNet-20 on CIFAR-10 (initialized in the same way) by evaluating the loss across the multi-dimensional line between them. After iteration $2k$, the instability becomes 0, meaning that the 2 networks are connected via an equal-loss line. Test error is represented with red, while training error is represented with blue ([13]).

Additionally, Simsek et al. (2021) have shown in [7] that a critical point in an m-dimensional space can generate an entire manifold of critical points in an n-dimensional space, where $n > m$. Furthermore, symmetry plays a role in generating even more critical points too. By permuting some of the parameters that comprise a critical point, one could obtain another critical point which yields the same loss value. This new higher-dimensional space is called an expansion manifold (Figure 5.5).

In this setup, an abundance of expansion manifolds originating in lower-dimensional global minima relative to expansion manifolds originating in lower-dimensional non-global critical points explains why heavily over-parameterized neural networks have such good convergence rates.

More recently, Zhang et al. (2021) outline in [14] a similar formalization of the landscape of a wide neural network. According to their embedding principle (Figure 5.6), the loss landscape of a wide DNN encodes all the critical points from all the narrower DNNs. Based on this embedding principle, they justify the faster rate convergence in heavily over-parameterized neural networks when compared to neural networks of smaller size.



Γ_0
$\mathcal{P}_{(1,2)}\Gamma_0$
(w_1, w_1) (a_1^1, a_1^2)
(w_1, w') $(a_1, 0)$
(w', w_1) $(0, a_1)$
(w_1, w_1, w_2) (a_1^1, a_1^2, a_2)
(w_1, w', w_2) $(a_1, 0, a_2)$
(w', w_1, w_2) $(0, a_1, a_2)$
(w_2, w_1, w_2) (a_2^1, a_1, a_2^2)
(w_2, w_1, w') $(a_2, a_1, 0)$
(w_2, w_1, w_1) (a_2, a_1^1, a_1^2)
(w_2, w', w_1) $(a_2, 0, a_1)$
$\mathcal{P}_\pi\Gamma_0$

Figure 5.5: Generating an expansion manifold from a pre-existent global minimum point. **Left**: A 2D expansion manifold obtained from a global minimum point of a neural network with 1 neuron. First, the neuron (w_1, a_1) is duplicated and $(w_1, w', a_1, 0)$ is obtained. The expansion manifold is constructed by 3 subspaces, connected by 3 segment lines: one across which w' varies and the others are constant, one across which a_1^1 and a_2^2 vary such that $a_1^1 + a_2^2 = a_1$, and one across which the first incoming weight varies, such that w_1 goes back to w'. **Right**: A 3D expansion manifold obtained from a critical point of a neural network with 2 neurons ([7]).

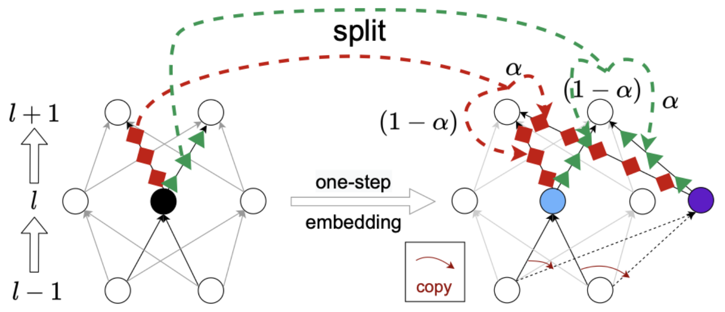


Figure 5.6: One-step embedding in action, by duplicating one neuron in the l^{th} layer of a DNN ([14]).

5.4 How Does Our Approach Compare with Prior Work?

What we have seen so far in a large proportion of literature is why over-parameterized neural networks are more successful than the under-parameterized ones. In a nutshell, we have seen that there are more and more global minima the more we over-parameterize and, conversely, we expect more and more non-global critical points the more we under-parameterize. The configuration with the highest amount of diversity is part of a regime that we call **mildly over-parameterized**. That is, there are not too many global minima, such that zero-loss is easily attainable, and there are not too many non-global critical points, such that only large losses are reachable. There is a reasonable mixture of both and that provides a very good insight into the landscape of a typical network.

The purpose of our study is to investigate this regime and understand empirically the landscape in this situation. We will attempt to validate the theory and to understand the nature of the points of failure. What kind of critical points are the points of failure where we get stuck? Are they strict saddles? Perhaps non-strict saddles? Or local minima? These are the sorts of questions that we will attempt to answer.

6 Experiments

In the following, the whole methodology of investigating the landscape of a neural network objective function is presented. The setup, the software tools, and the interpretation of the theoretical results will be provided here.

6.1 Teacher-student Framework

In this study, the setup that is used is a teacher-student setup. In order to investigate the various regimes, we'll use a reference network, the teacher, which can fit the dataset.

Although the teacher-student architecture is inspired primarily from knowledge distillation, where a reduced student is trained in order to replicate the outputs of a larger teacher and *to distill* its number of parameters simultaneously, a separate approach is used here. The teacher in the current study is a smaller network, whereas the student is a larger network.

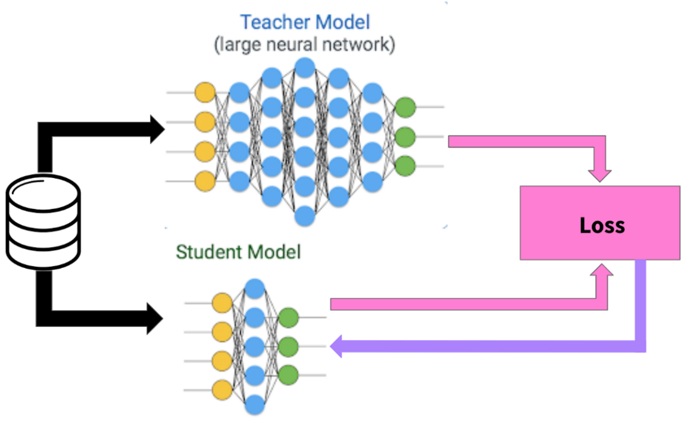


Figure 6.1: Teacher-student architecture in a knowledge-distillation setting [1].

Based on the teacher, then, a student is generated by adding multiple neurons. The student will be trained in order to replicate the output of the teacher entirely. In this way, one can measure the ease or difficulty with which the same point can be reached again from different

[1]Figure taken from `https://towardsdatascience.com/knowledge-distillation-simplified-dd4973dbc764`

initializations in the parameter space. Thus, by successively over-parameterizing the teacher, the evolution of the roughness to reach the very same point can be measured.

The totality of experiments that we have made have been implemented in Python. For implementing the models, PyTorch and JAX have been used, with support on Colab or on the EPFL cluster. For second-order optimization, two frameworks have been tried out:

- SciPy - proved to be somewhat unreliable, due to the difficulty in making the algorithm get really close to the critical point
- NLopt - way more reliable, being able to get really close to the critical point; the documentation is not accurate though, and the implementation is rather difficult to debug

For all the plots, *Matplotlib* was used, whereas *pandas* was used for dataset manipulation and anything alike.

6.2 The Toy Dataset

The first set of experiments that have been made are based on a toy dataset, borrowed from [7]. The toy dataset is made of N points (x_k, y_k), where $x_k \in \mathbb{R}^{d_{in}}$ and $y_k \in \mathbb{R}^{d_{out}}$. In our case we choose $N = 1681$ data points sampled from a bi-dimensional grid ($d_{in} = 2$), with 1D labels ($d_{out} = 1$). The grid is defined as follows:

$$\{(x_1, x_2) | 4x_1 = -20, ..., 20, 4x_2 = -20, ..., 20\}$$

The teacher that is used is an ANN with 1 hidden layer (Figure 6.2). The incoming weights are pre-set to the shown values and the outgoing weights (a_i) alternate between 1 and -1. The reason for this is related to allowing the model to learn more flexibly, by creating more flexible decision boundaries in the feature space that is generated by the hidden layer. Moreover, the activation function used in the hidden layer is the sigmoid and the model contains no biases. Hence, the labels with which a student may be trained is what the teacher outputs:

$$y_i = \sum_{i=1}^{4} a_i \cdot \sigma(\sum_{j=1}^{2} w_{ij} x_j)$$

$$y_i = \sigma(w_{11}x_1 + w_{12}x_2) - \sigma(w_{21}x_1 + w_{22}x_2) +$$
$$+ \sigma(w_{31}x_1 + w_{32}x_2) - \sigma(w_{41}x_1 + w_{42}x_2)$$

where $w_{11} = 0.6$, $w_{12} = 0.5$, $w_{21} = -0.5$, $w_{22} = 0.5$, $w_{31} = -0.2$, $w_{32} = -0.6$, $w_{41} = 0.1$, $w_{42} = -0.6$.

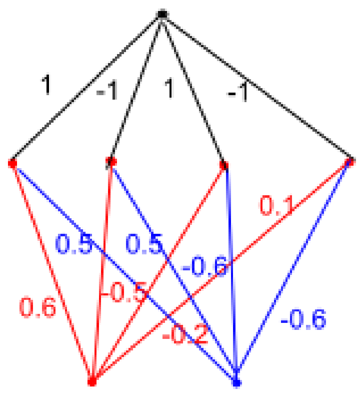


Figure 6.2: The teacher network architecture, together with its preset weights.

6.2.1 Outlook of Results

1000 different initializations starting from different seeds have been chosen with the setup aforementioned. The Xavier initialization ([15]) is used to initialize the weights of the student.

Every time the student is trained to replicate the teacher's output. Adam is used as an optimizer to train the network end-to-end with a fixed learning rate of 10^{-4}. The network is trained in a full-batch fashion for 10^7 epochs and second order optimization is applied at the end in order to get closer to the critical point.

LD_SLSQP from the *NLOpt* package is chosen for second order optimization. It is a sequential quadratic programming algorithm that allows the inclusion of both equality and inequality constraints. It does not provide convergence guarantees for finding the global minimum/-maximum (the *L* in the name stands for *local*) and it is gradient-based (the *D* in the name stands for *derivative*). *LD_SLSQP* is run for 10^4 seconds or until the difference between the current step and the last step is less than a preset tolerance (10^{-32}).

The gradients at convergence go as low as 10^{-19} and the loss values are as low as 10^{-32} for critical points that are also global minima. Moreover, the smallest eigenvalues is either 0 or slightly positive (e.g. 10^{-9}), but never negative. That means that there is never a negative curvature around the point where convergence stops, and hence no direct escape direction. This is true for both global minima and for the other points (Figure 6.3).

In Table 6.1, a general statistics of the results of the experiments is presented (the number of local minima, global minima, etc.).

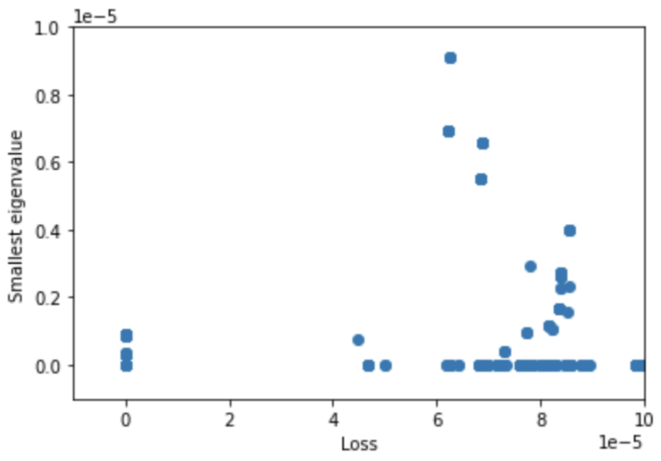


Figure 6.3: The correlation between the smallest eigenvalue and the loss value for each point at convergence.

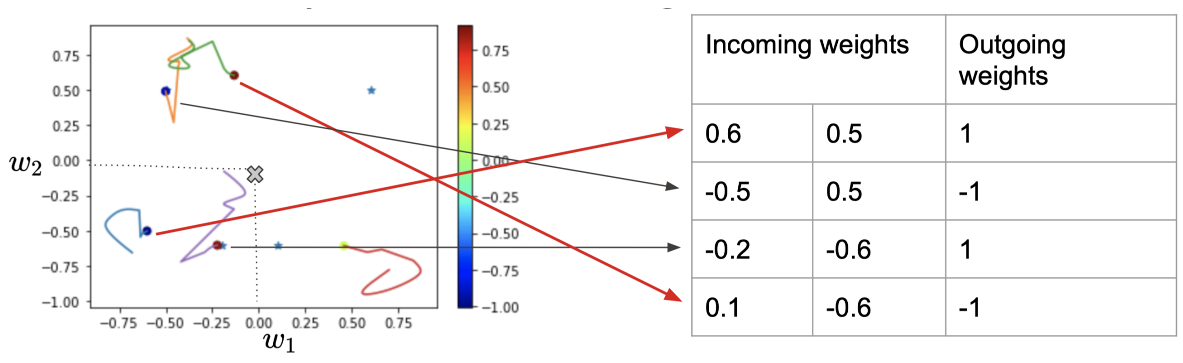
Outlook of initializations			
		Symmetry-induced critical points	Other local minima
Global minima	574	-	-
Local minima	426	55	371
Total	1000		

Table 6.1: Of 1000 initializations, 574 have ended in a global minimum. 426 of initializations have ended in local minima, among which 55 are local minima which originate in a smaller sized neural network and 371 are other local minima.

A visualization of the local and global minima is attempted next in order to better understand their nature and attempt to empirically confirm the theoretical expectations that correspond to these points.

6.2.2 Empirically Confirming Global Minima

In order to empirically confirm that the points which were found are global minima, one can attempt to investigate them visually. Given that the training is performed in a teacher-student setup, one would expect that the student be identical to the teacher in terms of the weights. That is intuitive because one would expect that the student has reached the very same point in the parameter space that the teacher had reached in its own parameter space. The points in both parameter spaces should be equivalent (we will exemplify what this means), even though they do not have the same dimensionality.



Incoming weights		Outgoing weights
0.6	0.5	1
-0.5	0.5	-1
-0.2	-0.6	1
0.1	-0.6	-1

Figure 6.4: **Left**: An example of a configuration of global minima. With stars, each neuron's incoming weights of the teacher are represented. With circles, each neuron's incoming weights of the student are showed and the color of the circles represents the value of each neuron's outgoing weight. Gradient trajectories are also showcased for each neuron. **Right**: The effective values of the teacher's incoming and outgoing weights are represented.

In Table 6.4, we show the evolution of the student's network weights from initialization to convergence on a per neuron basis. Two of the neurons (the black arrows) converge precisely to 2 of the teacher's weights. The other remaining 2 weights diverge (at least apparently). In order to understand why this happens, one needs to take into account a symmetry property of the sigmoid:

$$\sigma(-wx) = 1 - \sigma(wx)$$

Based on this, 2 neurons can learn $-w$ instead of w, such that their sum can still be identical. More specifically, assuming that the outgoing weights of the 2 neurons are 1 and -1

respectively, we have:

$$1 \cdot \sigma(-w_1 x) + (-1) \cdot \sigma(-w_2 x) = 1 - \sigma(w_1 x) - 1 + \sigma(w_2 x) =$$
$$= \sigma(w_2 x) - \sigma(w_1 x)$$

This means that learning $-w_1$ (with outgoing weight 1) and $-w_2$ (with outgoing weight -1) is equivalent to learning w_1 (with outgoing weight -1) and w_2 (with outgoing weight 1). This is exactly the case with the 2 remaining neurons from Table 6.4, which are signalled via the 2 red arrows. They learn $w = [-0.1, 0.6]$ with outgoing weight 1 and $w = [-0.6, -0.5]$ with outgoing weight -1.

Hence, this point is equivalent to the teacher's point and is empirically validated to be a global minimum. Many of the 574 points are global minima which incorporate the symmetry property of the sigmoid, but there are cases in which it is obvious that the student's weights are identical to the teacher's ones.

6.2.3 Empirically Checking Local Minima

Among the local minima that we have found, we would like to pose next the following question: what is the nature of these minima? We have found that 55 of the points are symmetry-induced. The remaining 371 are unclassified local minima and we will look into it in the next parts.

But before everything, we have seen in Figure 6.3 that all points (local minima included) at convergence have a smallest eigenvalue of at least 0. For the points for which the smallest eigenvalue is slightly positive, the convergence procedure was imperfect and it landed on a point in the proximity of the actual critical point. For the points for which the smallest eigenvalue is 0 though, one fundamental question can be asked. Namely, is there an escape direction?

6.2.4 Escaping Local Minima

Finding an escape direction is theoretically possible in the direction of the eigenvalue of 0. One could follow the corresponding eigenvector and go ahead until a point of a lower loss value is found. These points are typically called non-strict saddles (Figure 3.3). A simple algorithm can be devised in order to check the direction of the smallest eigenvector for an escape direction (Algorithm 1). It is worth mentioning though that the perturbation is 2-fold (in the direction of $-$*smallest_evector* and *smallest_evector*), since for any v - eigenvector, $-v$ is also an eigenvector ($Av = \lambda v$ and $A(-v) = \lambda(-v)$).

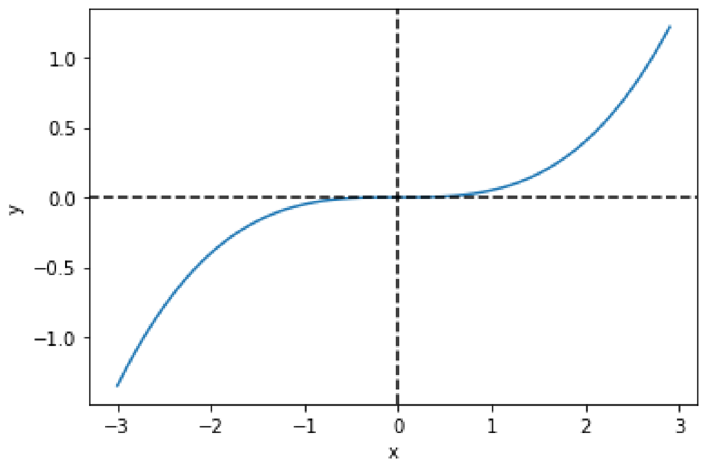


Figure 6.5: An example of 2D non-strict saddle.

Algorithm 1: An algorithm to check for lower loss points in the direction of the smallest eigenvector.

```
Data: $weights, smallest\_evector$
/* $smallest\_evector$ is the eigenvector corresponding to the smallest
   eigenvalue.                                                          */
/* $weights$ is an array containing all the network weights            */
Result: $True$ or $False$
/* $True$ if a lower loss value is found.                               */
/* $False$ otherwise.                                                   */
$lower\_bound \leftarrow -8.0$;
$upper\_bound \leftarrow 8.0$;
$perturb\_step \leftarrow 0.01$;
for $eps$ in $range(perturb\_lower\_bound, perturb\_upper\_bound, perturb\_step)$ do
    $perturbed\_weights = weights + eps * smallest\_evector$;
    if $loss(perturbed\_weights) < loss(weights)$ then
        return $True$;
    end
end
return $False$;
```

Based on this algorithm, no lower loss point was detected for any of the 426 local minima. A typical evolution of the loss around the given point is the one from Figure 6.6. The loss increases starting with a perturbation factor of 2 and -2, which indicates that there is no lower loss point in the direction of the smallest eigenvector. However, the fact that the smallest eigenvalue becomes negative at the same perturbation step may lead one to suspect that the smallest eigenvalue (and implicitly the smallest eigenvector) changes and that there may an escape direction if one follows the next smallest eigenvector from that position. This may indeed happen from a theoretical perspective. Nevertheless, in this case, the gradient norm also increases substantially, which means that the perturbed point is not a critical point anymore. If the points is not critical anymore, the changes in the curvature are not of interest anymore.

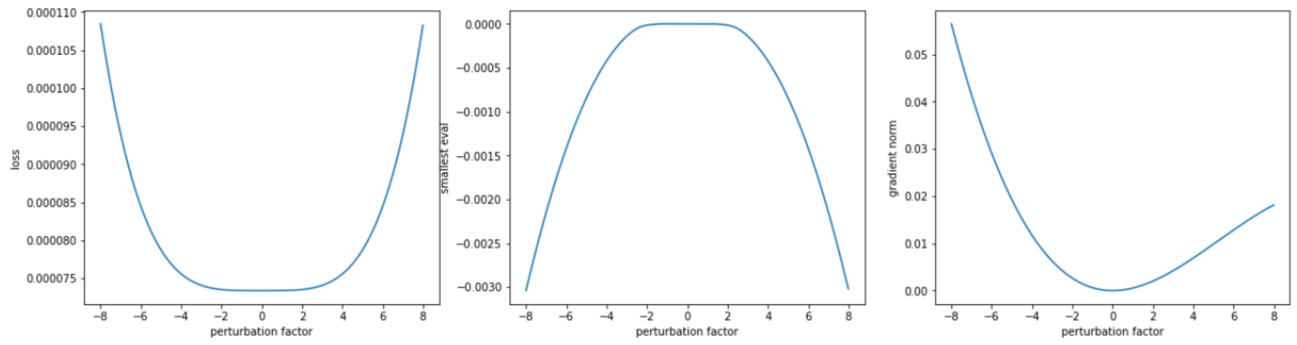


Figure 6.6: **Left**: The evolution of the loss with respect to the perturbation factor across the smallest eigenvector. **Middle**: The evolution of the smallest eigenvalue with respect to the perturbation factor. **Right**: The evolution of the gradient norm with respect to the perturbation factor.

An improvement to this algorithm implies two major changes:

- make sure that the gradient norm stays approximately 0 during perturbation
- dynamically recompute the smallest eigenvalue and the smallest eigenvector at each step and perturb using the new eigenvector

A sketch of the algorithm can be seen under Algorithm 2.

Although the algorithm seems somewhat heavy, its inner workings are straightforward. It resembles a BFS (breadth-first search) algorithm on a grid, the grid being simulated here via the discrete steps that we take from the weights where we start. The algorithm advances in a certain direction as long as the gradient norm is roughly 0 (a threshold is used) and as long as in a certain direction the point where we jump has not been explored already. The latter is done by computing the distance between the destination point and the point from the very start and by dividing it by the perturbation step (line 19 and 24). This gives us the

Algorithm 2: An improved algorithm to check for lower loss points across the flat surface.

```
Data: start_weights, smallest_evector
/* smallest_evector is the eigenvector corresponding to the smallest
   eigenvalue.                                                        */
/* start_weights is an array containing all the network weights       */
Result: True or False
/* True if a lower loss value is found.                               */
/* False otherwise.                                                   */
1  $q \leftarrow Queue()$;
2  $q.append((start\_weights, 0))$;
3  $eval\_zero\_threshold \leftarrow 1e-10$;
4  $grad\_threshold \leftarrow 1e-9$;
5  $perturb\_step \leftarrow 0.01$;
6  while not $q.empty()$ do
7      $weights, dist \leftarrow q.pop()$;
8      if $loss(weights) \leq loss(start\_weights)$ then
9          return True;
10     end
11     $H \leftarrow compute\_hessian(weights)$;
12     $evals, evectors \leftarrow spectral\_decomposition(H)$;
13     for $idx$ in $length(evals)$ do
14         if $evals[idx] \geq eval\_zero\_threshold$ then
15             continue;
16         end
17         $evector \leftarrow evectors[idx]$;
18         $perturbed\_weights \leftarrow weights + perturb\_step * evector$;
19         $perturbed\_dist = norm(perturbed\_weights - start\_weights) / perturb\_step)$;
20         if $norm(grad(perturbed\_weights)) \leq grad\_threshold$ and $perturbed\_dist > dist$
            then
21             $q.append((perturbed\_weights, perturbed\_dist))$;
22         end
23         $perturbed\_weights \leftarrow weights - perturb\_step * evector$;
24         $perturbed\_dist = norm(perturbed\_weights - start\_weights) / perturb\_step)$;
25         if $norm(grad(perturbed\_weights)) \leq grad\_threshold$ and $perturbed\_dist > dist$
            then
26             $q.append((perturbed\_weights, perturbed\_dist))$;
27         end
28     end
29 end
30 return False;
```

number of steps that have been made since the very start and, thus, in this way, some already explored states can be marked by keeping this number in the queue, together with the state (the weights).

However, even with Algorithm 2, no lower loss point has been identified.

6.2.5 Critical Points Close to Saddles

Based on existing literature and ongoing research, it is often the case that when neural networks get stuck in a local minima, that could very well be in the vicinity of a saddle point. In this context, we are wondering if and how many of our points are indeed in the proximity of a saddle.

The reason why this happens is because local minima in a network get transformed in saddle points in an over-parameterized network ([7]). In the particular case of a teacher network of width 4 and a student network of width 5, we would expect a saddle point to contain 2 neurons which essentially overlap. Because such a saddle is escapable, the 2 neurons diverge during the convergence process, but whenever we see 2 neurons relatively close to each other at convergence, we suspect that, during their history, they were once merged.

We attempt to empirically identify such nearby saddles and confirm some theoretical results with regard to the existence of local minima in their vecinity.

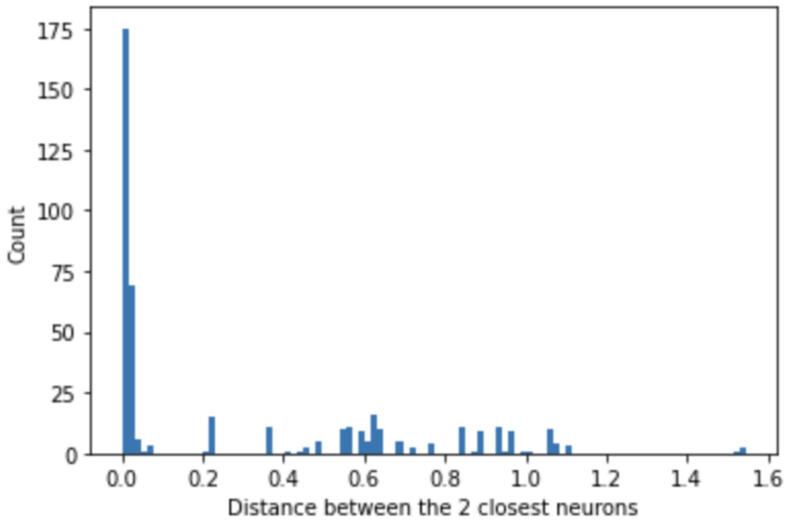


Figure 6.7: Histogram of the smallest distance between any 2 neurons for the 426 local minima.

In Figure 6.7, one can notice that a large proportion of the points have a smallest distance between any two neurons concentrated around 0. Some of the remaining ones also correspond to a small distance (e.g. 0.2) and may also be local minima from the vicinity of saddle. Now, for all such points, we want to trace back their origin to the saddle and understand the proximity between them. For this, we have devised an algorithm starting from what Brea et al. (2019) have devised in [16]. The main idea is to try to reconstruct the saddle from the current local minimum. The idea works as follows:

- Create a new objective loss which optimizes over the merging of the two closest neurons.
- At this point, 2 of the network's neurons overlap. We aim at reducing the network by one neuron, hoping to land close to the local minimum of the reduced parameter space.
- Optimize from here in order to get as close as possible to the local minimum from the reduced network.
- Generate the saddle in the over-parameterized network from the local minimum in the reduced network.

We will further discuss these points one by one and we will see how we can actually implement them.

A New Objective Loss

As proposed in [16], a new loss is proposed by incorporating the distance between the two smallest neurons. By minimizing the loss, we force the minimization of the distance between the two neurons too. The loss that we use works as follows:

$$L = L_{MSE} + \lambda ||w_i - w_j||$$

, where L_{MSE} is the mean squared error loss that we normally train the network with.

λ is a constant that must be fine-tuned in order to not ignore the norm term (by making λ too small) and at the same time to not make it too high (by ignoring L_{MSE}). A value of $\lambda = 0.1$ has worked for us by merging the neurons to the point of overlap. It is worth noting that this method works similarly to a regularization method.

Reducing the Network By One Neuron

After virtually merging the 2 neurons together, we attempt to obtain only 1 neuron out of them. In order to do that, the incoming weight vectors of the 2 neurons are averaged, whereas the outgoing weights are summed up. The hope is that, after this process, one can land at a point in the reduced parameter space that is very close to the local minimum.

Optimizing the Reduced Network's Loss

There is no guarantee that we have landed close to the local minimum in the reduced network. Hence, we optimize the reduced network once more using the *MSE* loss, hoping to reach the local minimum this time. Second order optimization is used this time due to its capability to get arbitrarily close to the critical point.

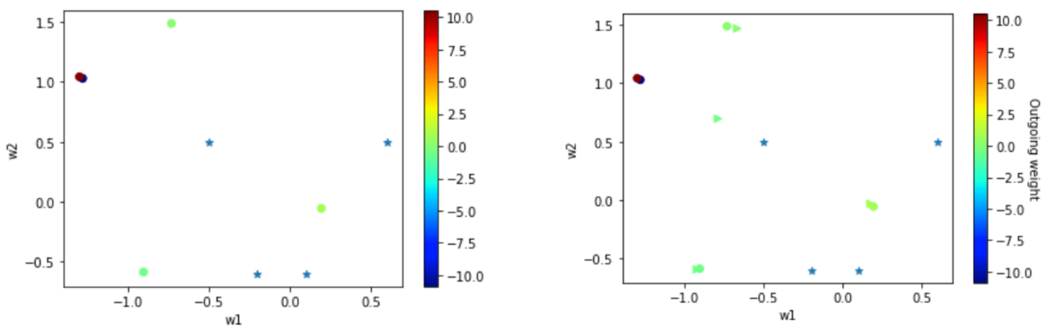


Figure 6.8: **Left**: The teacher's configuration (stars) and the student network at convergence (circles). **Right**: The reduced network at convergence after merging and applying second order optimization (triangles).

Generating the Saddle

After obtaining the local minimum in the reduced network, one can obtain a saddle point in the student network by duplicating the very same neuron that was obtained by merging the 2 initial closest neurons. In fact, one can obtain an entire line of saddle points by varying the outgoing weights of this neuron, as we discussed in previous chapters:

$$a \cdot \sigma(wx) \mapsto (\mu \cdot a \cdot \sigma(wx), (1-\mu) \cdot a \cdot \sigma(wx)), \forall \mu$$

Hence, by varying μ, one can obtain the aforementioned line of saddle points, due to the fact the sum of the resulting 2 neurons is equivalent to the neuron that we duplicate.

Analysis of the Saddle-Local Minimum Distance

After applying this routine, we analyze the distances between the identified saddle and the original local minimum in the larger network. In Figure 6.9, we can see that less than 200 of the local minima are situated within a small distance to the identified saddle. Intuitively, we expect saddles situated at smaller distances to the local minimum to be more reasonable and accurate, since the algorithm that we employed is a mere heuristic and for more distant initial neurons (i.e. neurons which are not close to each other in the student network) it may fail to converge to a meaningful point and hence yielding a potentially large distance.

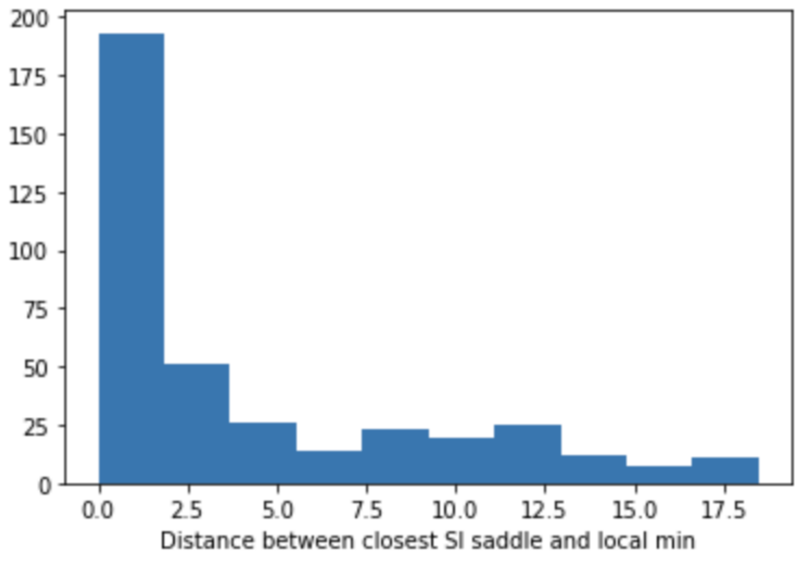


Figure 6.9: Distribution of the distances between the identified closest saddle and the original local minimum.

Therefore, for some of these distance that we have just seen, we would like to empirically investigate the loss between the corresponding closest saddle and local minimum. Naturally, we would expect to see the loss decreasing from the closest saddle in a certain direction, due to the negative curvature of the saddle, and end up after a certain amount of time at the local minimum. This type of investigation can be done two-fold: one can create a 1D visualization of the loss between the saddle and the local minima and a 2D visualization between the saddle line and the local minimum.

In Figure 6.10, a sample of such a visualization is provided. We can see that the loss decreases in the left figure from the saddle point to the local minimum, showcasing that there is indeed a negative curvature which also provides the escape direction. This can also be seen in the figure on the right, in which the valley gets darker in between the saddle line and the local minimum. The three smallest eigenvalues at the closest saddle are $-6.5x10^{-5}$, $1.4x10^{-11}$, and $6.3x10^{-6}$. The negative direction that we observe is seemingly the one correspondent to the smallest eigenvalue.

Let us look now at another point, for which the 1D visualization is not entirely representative of the evolution of the loss one would expect (Figure 6.11). Here, one can see, in the 1D case,

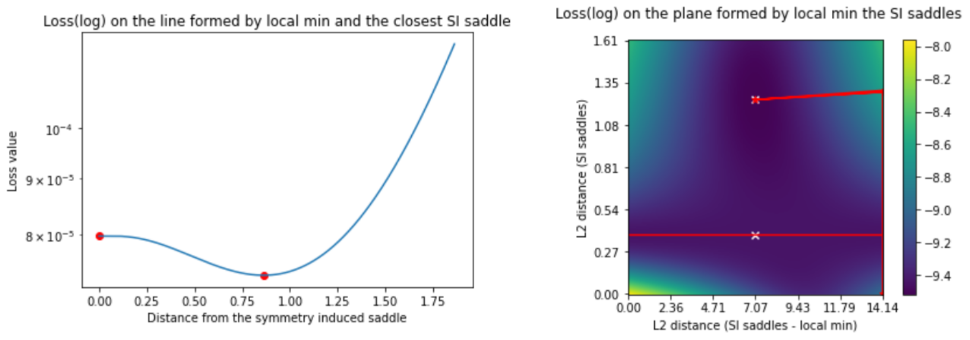


Figure 6.10: **Left**: A 1D visualization of the loss function along the line that connects the closest saddle and the local minimum. **Right**: A 2D visualization of the space between the saddle line (bottom red line) and the local minimum (top). The upper red line showcases the projected trajectory of the gradient on this 2D space from initialization up until convergence. The color represents the value of the loss in logarithmic scale (the darker the color, the lower the loss).

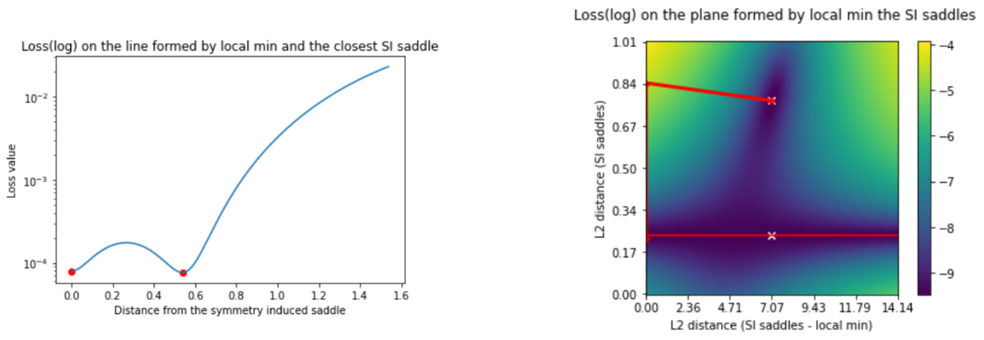


Figure 6.11: An example of visualization for which the decrease in loss is not apparent. **Left**: 1D visualization of the loss function along the line that connects the closest saddle and the local minimum. **Right**: 2D visualization of the space between the saddle line (bottom red line) and the local minimum (top). The upper red line showcases the projected trajectory of the gradient on this 2D space from initialization up until convergence. The color represents the value of the loss in logarithmic scale (the darker the color, the lower the loss).

that the loss first increases and then decreases. However, this is further clarified in the 2D scenario, in which one can see that there is a path of decreased loss further to the left of the geometrically closest saddle. For completion, the top 3 smallest eigenvalues in this case are $-2.1x10^{-4}$, $-2.3x10^{-18}$, and $6.3x10^{-6}$. A negative eigenvalue is also present here, but following the direction of its corresponding eigenvector does not guarantee the reaching of the local minimum.

Further, what one can notice based on these two examples is that the second smallest eigenvalue is a negligible quantity (effectively 0). This can be explained by the duplication process. After identifying a local minimum in the parameter space of the reduced network, we duplicate one of the neurons to generate the saddle line in the student network's space. Let us assume this neuron is the first one, (w_1, a_1), where w_1 represents the incoming weights vector and a_1 represents the outgoing weight for that neuron. After duplication, there will be two neurons, whose sum will equal the value of this neuron: (w_1, a_1^1) and (w_1, a_1^2), where $a_1^1 + a_1^2 = 1$. The Hessian at one of the points on the resulting saddle line will be of the form:

$$H = \begin{bmatrix} \frac{\partial L^2}{\partial w_1^2} & \frac{\partial L^2}{\partial w_1 \partial w_2} & \dots & \frac{\partial L^2}{\partial w_1 \partial a_5} \\ \frac{\partial L^2}{\partial w_1^2} & \frac{\partial L^2}{\partial w_1 \partial w_2} & \dots & \frac{\partial L^2}{\partial w_1 \partial a_5} \\ \frac{\partial L^2}{\partial w_2 \partial w_1} & \frac{\partial L^2}{\partial w_2^2} & \dots & \frac{\partial L^2}{\partial w_2 \partial a_5} \\ \dots & & & \\ \frac{\partial L^2}{\partial w_5 \partial w_1} & \frac{\partial L^2}{\partial w_5 \partial w_2} & \dots & \frac{\partial L^2}{\partial w_5 \partial a_5} \\ \frac{\partial L^2}{\partial a_1 \partial w_1} & \frac{\partial L^2}{\partial a_1 \partial w_2} & \dots & \frac{\partial L^2}{\partial a_1 \partial a_5} \\ \dots & & & \\ \frac{\partial L^2}{\partial a_5 \partial w_1} & \frac{\partial L^2}{\partial a_5 \partial w_2} & \dots & \frac{\partial L^2}{\partial a_5 \partial a_5} \end{bmatrix}$$

From this Hessian matrix, it is visible that the first two rows are identical. As an immediate consequence, the matrix is not full rank, and, therefore, $det(H) = 0$. Hence, a solution to the equation $det(A - \lambda I) = 0$ (which is required to be solved in order to identify eigenvalues) is $\lambda = 0$. Consequently, there must at least one eigenvalue of 0, which is in utter accordance with what we have seen in the experiments.

Now, given the intuitive results that we've got so far, we'd like to go beyond mere intuition and confirm theoretically that these points are in the vicinity of a saddle. For that, we attempt to use some ongoing theoretical results, which prove the existence of local minima in the vicinity of a saddle under several constraints. The theoretical results can be phrased as follows:

Theorem 1 *Assume that the saddle point x^* has index-1: its Hessian has one negative eigenvalue and the other eigenvalues are positive. For all unit u such that $u^T \nabla^2 f(x^*) u \leq 0$, let*

$B = \min_{\|u\|=1} \max\{\nabla^3 f(x^*)(u), \nabla^3 f(x^*)(-u)\}$. We assume $B > 0$ exists. Assume that there is an $R > 0$ such that $\nabla^4 f(\xi)(u) \geq 0$ for all $\|\xi - x^*\| < R$ and $\|u\| = 1$. If

$$\frac{-3\lambda_{\min}}{B} \leq R,$$

then we have a local min. within an l_2-distance $-3\lambda_{\min}/B$ from the saddle point x^*.

Theorem 2 *Assume that the saddle point x^* has an arbitrary index. Assume further that there is an $R > 0$ such that $\nabla^4 f(\xi)(u) \geq \delta > 0$ for all $\|\xi - x^*\| < R$ and $\|u\| = 1$. Let $B = \min_{\|u\|=1} \nabla^3 f(x^*)(u)$. If*

$$r = \frac{2}{\delta}(\sqrt{B^2 - 3\lambda_{\min}\delta} - B) \leq R,$$

then we have a local min. within an l_2-distance r from the saddle point x^*.

By applying Theorem 1 and Theorem 2 on the newly found saddle points, we would like to formally prove the existence of local minima within a quantifiable distance from the saddle. Depending on the nature of the saddle (how many negative eigenvalues it has), one of the 2 theorems is applicable.

Applying the Theorems

First and foremost, before applying the theorem, the question we may naturally ask is: does the saddle fulfil the assumptions of the theorem? If we look at the point from Figure 6.10 and, in particular, its eigenspectrum, we see that one of the eigenvalues is negative, one is essentially 0, and the others are positive. The point is not exactly index-1, but since the theorem was originally devised for ReLU networks and our network is a sigmoid one, we still intuitively expect the theorem to hold and attempt to test it regardless in order to verify if it is still applicable.

In order to do this, we identify the unit vectors u for which $u^T \nabla f(x^*) u \leq 0$. Intuitively the areas in which we are interested should give us two cones - Figure 6.12 -, due to the symmetry of eigenvectors - if for u the inequality from above holds, then it must also hold for $-u$.

The second condition that must be verified is the one involving the third derivative. Namely, we should find the solution to the following minimization problem:

$$B = \min_{\|u\|=1} \max\{\nabla^3 f(x^*)(u), \nabla^3 f(x^*)(-u)\}$$

.

However, since the directional third order derivative is positive in only one of the two cones (either $f(x^*)(u) > 0$ or $f(x^*)(u) < 0$), it suffices to search in only one of the two cones.

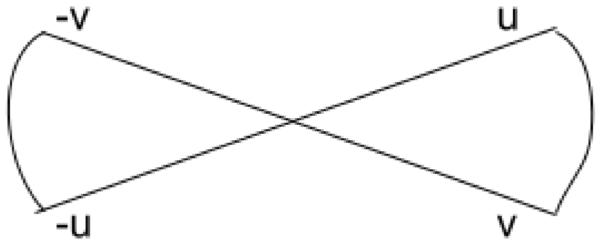


Figure 6.12: The two cones which respect the property $u^T \nabla f(x^*) u \leq 0$.

Nevertheless, we formulate this problem into a non-convex optimization problem with convex constraints and feed it into a solver from *NLOpt*.

$$B = \min \max\{\nabla^3 f(x^*)(u), \nabla^3 f(x^*)(-u)\}$$

, with the constraints:

$$\|u\| = 1$$
$$u^T \nabla f(x^*) u \leq 0$$

The last but not least condition that is to be verified is the fourth-derivative condition. We must check that $\forall \xi$, $\nabla^4 f(\xi)(u) \geq 0$. For this condition, an easy way that we have used was to sample arbitrary ξ within a radius and check that the condition is fulfilled for all of them. However, not even for $\xi = x^*$ has the condition been fulfilled, which has led us to a stall. The theorem could not be verified on this setup, with a sigmoid activation function and 1 hidden layer, due to the fact that the fourth derivative does not fulfil the required conditions.

This results also hold for the other saddle points starting from the other local minima. No matter the index of the points (i.e. the number of negative eigenvalues), the fourth derivative is never positive within a certain radius.

6.2.6 Symmetry-induced Critical Points

Further, we would like to discuss another category of local minima. Among the 426 local minima, 66 have been classified in Table 6.1 as symmetry-induced local minima. As it was explained in previous chapters, that means that these points essentially originate in a lower-dimensional parameter space. Two approaches have been devised in order to verify this properly. Firstly, we have noticed experimentally, through mere visualization, this phenomenon and then, secondly, we confirmed it by verifying a theoretical result.

From a visual perspective, in Figure 6.13, one can identify such neurons, for which the heuristic of finding the closest saddle does not yield substantial differences in terms of the value of the weights. This strongly indicates that the point in the reduced network is essentially identical to the point in the student network.

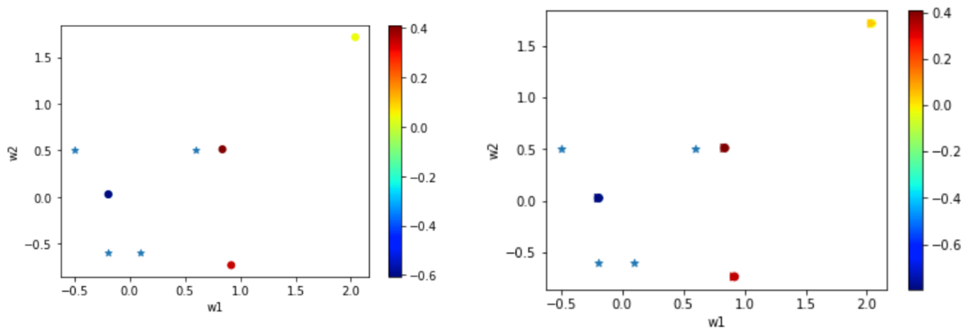


Figure 6.13: **Left**: Two neurons on the left are perfectly overlapped. They got merged together when the point was a saddle during the convergence history and even after escaping the saddle, they did not get separated. **Right**: The triangles (denoting the point in the reduced network after applying our heuristic for identifying the closest saddle) virtually overlap with the initial point in the student network.

Furthermore, one can also see the same pattern in the surrounding loss values of the identified closest saddle. In Figure 6.14, in the 1D visualization from the left-hand side, one can notice that the closest saddle is negligibly close to the local minimum (the distance is on the order of 10^{-4}. Similarly, in the 2D visualization on the right-hand side, one can see the oscillations at convergence. The initialization point is in the upper part of the figure, it goes down beyond the local minimum, and then it goes up again towards the saddle point.

Nevertheless, besides the empirical identification of such points, we would like to prove the existence of such points more formally. For that, an ongoing theoretical result has been applied that formally checks if such points fulfil some prerequisites in order to be classified as symmetry-induced saddles. The theorem goes as follows:

Theorem 3 *Let θ^* be an irreducible local minimum of a two-layer network of width r. Let $(\theta^* \oplus [\theta^*]_j)(\mu)$ be a symmetry-induced critical point corresponding to the duplication of $[\theta^*]_j$ by splitting its outgoing weight into $(\mu a_j^*, (1-\mu)a_j^*)$. The spectrum of the Hessian of this SI critical point has $(d + d_{out})r$ (where d-the dimensionality of the input and d_{out}-the dimensionality of the output) positive eigenvalues. The signs of the remaining $d + d_{out}$ eigenvalues of the Hessian are given by the signs of*

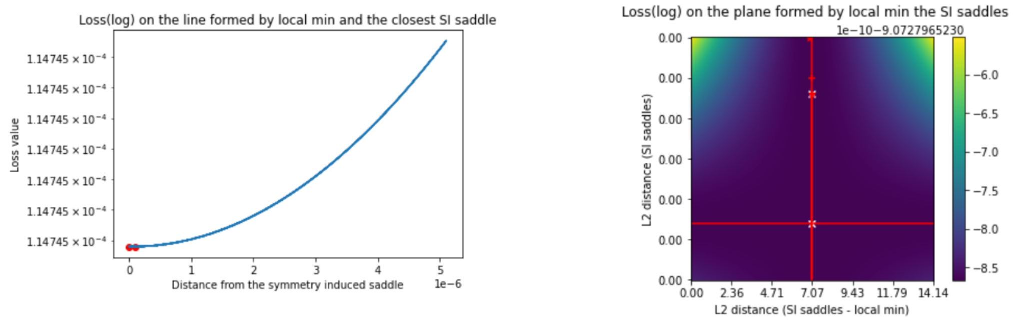


Figure 6.14: **Left**: Small distance between the local minimum and the closest saddle in 1D. **Right:** A negligible difference between the saddle line and the local minima (showcased on the oy axis by values of virtually 0).

the eigenvalues of the following $(d + d_{out}) \cdot (d + d_{out})$ *matrix:*

$$\begin{bmatrix} \mu(1-\mu)Y(w^*, a^*) & U(w^*, a^*) \\ U(w^*, a*)^T & 0 \end{bmatrix}$$

where

$$Y(w^*, a^*) = \mathbb{E}[\sigma''(w^* \cdot x)xx^T a^* c'(f(x), f^*(x))] \in R^{dxd},$$
$$U_{ij}(w^*, a^*) = \mathbb{E}[\sigma'(w^* x)x_i c'(f(x), f^*(x))_j].$$

Above we have noted with $c'(f(x), f^*(x))$ *the difference between the output of the network and the ground truth (in our setup) and, more generally, the differential of the loss function.*

Firstly, we will show that $U_{ij} = 0$ always because we have only one output neuron. We will start from the obvious property that the point at convergence has a gradient of 0:

$$\frac{\partial L}{\partial w_j} = 0$$

The loss, however, is:

$$L = \sum_{i=1}^{n} (y^i - \sum_{j=1}^{4} a_j \sigma(w_j x^i))^2$$

This leads to:

$$\frac{\partial L}{\partial w_j} = \sum_{i=1}^{n} 2(y^i - \sum_{j=1}^{4} a_j \sigma(w_j x^i)) \cdot (-a_j) x^i \sigma'(w_j x^i) = 0$$

$$\sum_{i=1}^{n} 2e(x)_i(-a_j)x^i \sigma'(w_j x^i) = 0$$

$$\sum_{i=1}^{n} e(x)_i x^i \sigma'(w_j x^i) = 0$$

But since the input x^i is two-dimensional, the last expression from above is actually a vector. Therefore, by taking each element of the vector, one can get:

$$\sum_{i=1}^{n} e(x)_i x_1^i \sigma'(w_j x^i) = 0$$

, and:

$$\sum_{i=1}^{n} e(x)_i x_2^i \sigma'(w_j x^i) = 0$$

Let us go back to U now. Taking into account the 2 last expressions from above, we get:

$$U_{11} = \frac{\sum_{i=1}^{n} \sigma'(w^* x) e(x) x_1^i}{n} = 0$$
$$U_{12} = \frac{\sum_{i=1}^{n} \sigma'(w^* x) e(x) x_2^i}{n} = 0$$

Hence, the U matrix is always 0 due to the fact that we have only one output neuron.

Corollary 3.1 *For the particular case of $U_{ij} = 0$, one can conclude that the SI critical point is a degenerate local minimum of minimum eigenvalue 0 if $\mu(1-\mu)Y$ has only positive eigenvalues. Similarly, the SI critical point is a saddle point if Y has at least one negative and one positive eigenvalue.*

In our situation, we expect $\mu(1-\mu)Y$ to have only positive eigenvalues for the closest critical point we find, in order to prove that the original critical point of the student network is a degenerate local minimum. This has indeed been true for all the 66 local minima for which we noticed very small distances between the closest saddle and the local minimum. The right μ was detected from the very small difference between the closest critical point and the local minimum, as before. The computation of $\mu(1-\mu)Y$ and U fulfill the required conditions and, thus, the formal check has been successful.

6.3 The MNIST Dataset

In order to get closer to modern deep learning, we have worked on extending our results on another setup. In this setup, we use a larger fully-connected neural network, with 3 hidden layers, and with 10 neurons each (Figure 6.15). The model also contains biases this time and it uses sigmoid as an activation function. To train the network, we use the top 10 PCA components of the MNIST dataset and as labels, we use $+1$ if the digit is odd and -1 if the digit is even. The MSE loss is used for training and a different approach to the student-teacher setup from before is used. Namely, the student is not trained to mimic the teacher anymore and, instead, we detect critical points in the teacher's parameter space and once they are found, the network is over-parameterized by one neuron and a saddle line is generated in the over-parameterized space.

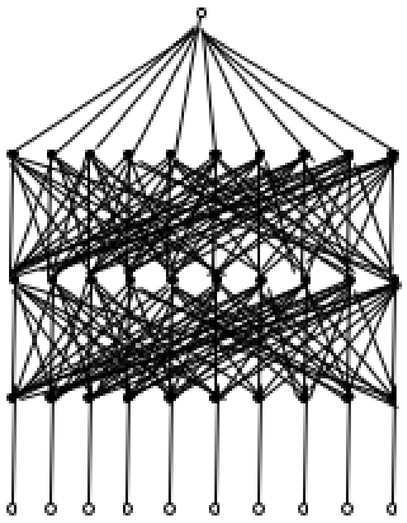


Figure 6.15: A fully-connected neural network with 3 hidden layers and one output neuron.

Furthermore, we train the network with Adam (with a constant learning rate of 10^{-4} for $2 \cdot 10^7$ epochs, stopping when the gradient reaches a value on the order of of 10^{-6}. No second order optimization is applied anymore in this setup.

6.3.1 Symmetry-induced Critical Points - Checks

After convergence, we want to perform the same checks that we have performed for the toy dataset setup. Namely, we want to check whether the symmetry-induced critical point is a saddle, a local minimum, or a degenerate local minimum. For that we will apply Theorem 3 and Corollary 3.1.

Duplicating on the Last Layer

In this case, every U_{ij} is 0 as before, by following the same arguments, since we have only one output neuron. We want to examine the nature of Y next. However, the neural network, this time, is a deeper one and, hence, the Y matrix should also be adjusted. According to the ongoing theoretical results, they Y matrix for a neuron to be duplicated on the layer l is:

$$Y(w_j^l, b_j^l, a_j^l) = \mathbb{E}[\sigma''([w_j^l, b_j^l] \cdot [f^{l-1}(x), 1])[f^{l-1}(x), 1][f^{l-1}(x), 1]^T a_j^l (f^L(x) - f^*(x))]$$

$$\in R^{(r_{l-1}+1)x(r_{l-1}+1)}$$

, where we have noted with f^L - the output of the network and with $f^*(x)$ - the ground truth.

By applying this formula, one can see in Figure 6.16 that Y has both positive and negative eigenvalues, which, according to theorem 3, means that the symmetry-induced critical point is a saddle point for certain values of μ, for which $\mu(1-\mu)Y$ preserves the eigenspectrum of positive and negative eigenvalues.

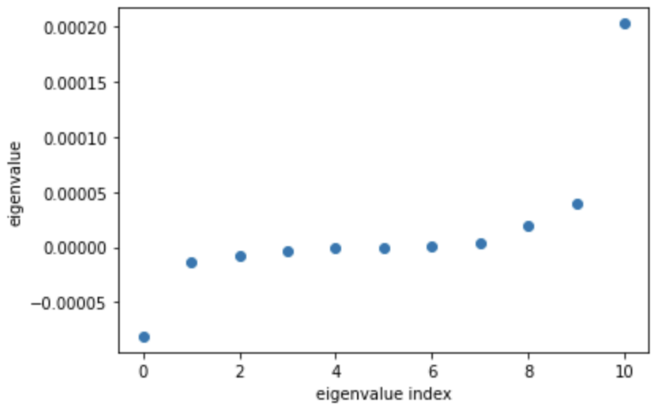


Figure 6.16: Eigenvalues for the Y matrix on a critical point of a neural network trained on the MNIST dataset.

Inspecting the Saddle Line

One further check that we want to perform is about the eigenvalues of the Hessian for the points along the saddle line. We have said before that, according to the Y matrix, the SI critical point is a saddle point for certain values of μ. We want to inspect for various values of μ what is going on and how the eigenspectrum of the Hessian looks like. Before, we start though, we will introduce another corollary to the theorem previously mentioned that clarifies the notion of *certain values of μ*.

Corollary 3.2 *Let $d_{out} = 1$ and $Y(w^*, a^*)$ have i^+ positive, i^0 zero, and i^- negative eigenvalues. Note that $i^+ + i^0 + i^- = d$ (where d - the dimensionality of the input). If both $i^+, i^- \geq 1$, then the SI-critical point $(\theta^* \oplus [\theta^*]_j)(\mu)$ is an index $- i^-$ (i.e. the Hessian contains i^- negative eigenvalues) saddle for $\mu \in (0,1)$, whereas it is an index $- i^+$ saddle for $\mu \in R \setminus [0,1]$. If $i^+ = d$, then the SI-critical point $(\theta^* \oplus [\theta^*]_j)(\mu)$ is a degenerate local minimum for all $\mu \in (0,1)$ (respectively, if $i^- = d$, then for all $\mu \in R \setminus [0,1]$, the SI-critical point is a degenerate local minimum.*

In particular, according to Corollary 3.2, at $\mu \in 0, 1$, the SI-critical point is a degenerate local minimum, having the smallest eigenvalues equal to 0. Because the matrix $\mu(1-\mu)Y$ is $11x11$, and the matrix from Theorem 3 will be a 0-matrix of size $12x12$, we expect 12 eigenvalues to cross 0 at $\mu = 0$. By a similar reasoning, we expect the same to happen for $1-\mu = 0$, or equivalently $\mu = 1$.

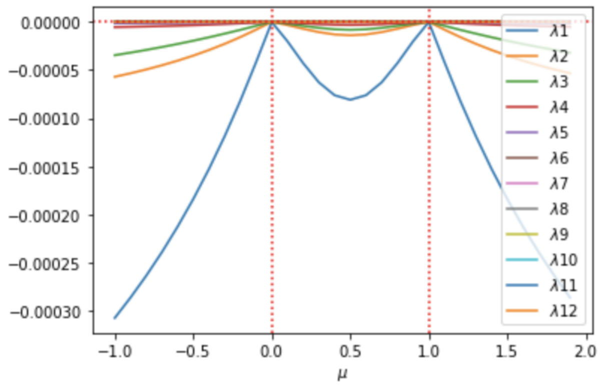


Figure 6.17: The smallest 12 eigenvalues of the Hessian on the generated saddle line for a neuron duplicate on the last layer.

As one can see in Figure 6.17, 12 eigenvalues cross 0 at $\mu = 0$ and $\mu = 1$ as the theory predicts. Furthermore, 10 eigenvalues of the Y matrix are positive, which according to Corollary 3.2, leads 10 negative eigenvalues of the Hessian for a point on the saddle line for $\mu \in R \setminus [0,1]$. This is also confirmed numerically based on Figure 6.17, since 10 eigenvalues are negative on those segments.

Furthermore, the theory is more difficult to apply for neurons on other than the last layer. A question one might validly ask is how many eigenvalues cross 0 at $\mu = 0$ when one duplicates a neuron on the second layer.

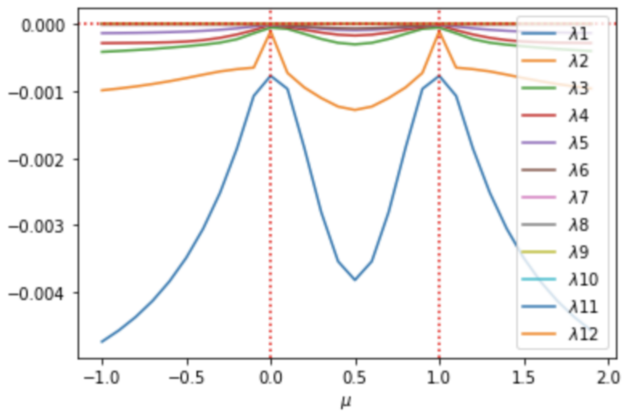


Figure 6.18: The smallest 12 eigenvalues of the Hessian on the generated saddle line for a neuron duplicated on the second layer.

In Figure 6.18, one can see that 4 eigenvalues cross 0 at $\mu = 0$ and $\mu = 1$ (eigenvalues less than 10^{-9} were considered). This has applications in the next part, where we attempt to start from the saddle line (ideally where the curvature is steep enough) and try to find the closest minimum to the saddle line.

6.3.2 Finding the Minimum Closest to The Saddle Line

Having identified a local minimum, expanded it in a saddle line and having made sure it contains strict saddles, we would like to restart convergence from one such strict saddle and attempt to find the closest local minimum to this point. Since the gradient is zero at any point on saddle line, however, gradient descent-based methods cannot make any convergence step from there. For this reason, a perturbation is required in the vicinity of the saddle line such that the zero gradient area can be escaped and only after that convergence can be restarted.

The perturbation is chosen from a Gaussian array of mean 0 and standard deviation 0.01. In terms of optimization, different optimizers have been tried out (Adam and SGD) with constant learning rate (different fixed values ranging from 10^{-3} to 10^{-6}), as well as a cyclical learning rate scheduler (with values equal to the previous ones). It turns out that in every such experiment (for different μ, i.e. different points on the saddle line which were afterwards perturbed), we end up in a global minimum after convergence. This can indicate either some hyperparameter issue (e.g. learning rate, μ) or some theoretical issue with the current seed or something alike. Further effort is required in order to investigate the cause of this issue.

6.4 Future Work

The main step is performing a more comprehensive set of experiments on the MNIST dataset. An end-to-end set of statistics in the mildly over-parameterized regime should be made, in order to identify the likelihood of ending up in symmetry-induced critical points, local minima, and global minima. In addition to this, further efforts should be made in order to identify local minima near the symmetry-induced saddle line for multiple seeds and to understand how dense the local minima are in the vicinity of the saddle line.

From a more theoretical standpoint, the theorems that we discussed about in this work should be extended and applied to the same as well as to other realistic setups. After empirically identifying a local minimum in the vicinity of a symmetry-induced saddle, we would like to theoretically prove its existence as well. This can be done by having a theorem applied which can work in a wide range of setups and for different activation functions.

7 Conclusion

In this work, we have put into perspective how the loss landscape of a standard neural network with at least 1 hidden layer can look like. We have explored how common saddle points, local minima, and global minima are in the mildly over-parameterized regime and how local minima in a sub-parameterized regime can become a manifold of saddle points and, more generally, a manifold of symmetry-induced critical points in an over-parameterized regime. Furthermore, we have shown that local minima can and do exist in the vicinity of symmetry-induced saddle points and that such saddles provide an escape direction towards them. What is more, local minima in one network can result in symmetry-induced degenerate local minima (with eigenvalues of 0) in the mildly over-parameterized regime, a fact seen in our experiments. On a more complex setup (the MNIST dataset), the same properties of the saddle line hold too.

Conversely, we have shown that some of the ongoing theoretical results are not entirely applicable to the setup we have made, potentially due to the fact that there is an abundance of local minima close to the saddle line or due to theoretical differences between the various activation functions (ReLU and the sigmoid). Additionally, we have noticed that convergence issues may stop us from reaching the local minima nearby the saddle line on more complex datasets (such as MNIST) and models.

For future work we have left an investigation of the theoretical limitations that we have previously mentioned. Convergence issues in the MNIST setup have to be further inquired too, in order to understand if it is a hyperparameter issue or something else. Another line of work could go into investigating how common symmetry-induced local minima are in a more realistic setup, such as the MNIST one.

List of Figures

List of Tables

Bibliography

[1] L. Bottou et al. “Stochastic gradient learning in neural networks”. In: *Proceedings of Neuro-Nımes* 91.8 (1991), p. 12.

[2] D. P. Kingma and J. Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[3] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein. “Visualizing the loss landscape of neural nets”. In: *Advances in neural information processing systems* 31 (2018).

[4] A. Jacot, F. Gabriel, and C. Hongler. “Neural tangent kernel: Convergence and generalization in neural networks”. In: *Advances in neural information processing systems* 31 (2018).

[5] I. Safran and O. Shamir. “Spurious local minima are common in two-layer relu neural networks”. In: *International conference on machine learning*. PMLR. 2018, pp. 4433–4441.

[6] G. Cybenko. “Approximation by superpositions of a sigmoidal function”. In: *Mathematics of control, signals and systems* 2.4 (1989), pp. 303–314.

[7] B. Simsek, F. Ged, A. Jacot, F. Spadaro, C. Hongler, W. Gerstner, and J. Brea. “Geometry of the loss landscape in overparameterized neural networks: Symmetries and invariances”. In: *International Conference on Machine Learning*. PMLR. 2021, pp. 9722–9732.

[8] S. Sheffield. “Gaussian free fields for mathematicians”. In: *Probability theory and related fields* 139.3 (2007), pp. 521–541.

[9] Y. N. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio. “Identifying and attacking the saddle point problem in high-dimensional non-convex optimization”. In: *Advances in neural information processing systems* 27 (2014).

[10] S. Fort and S. Jastrzebski. “Large scale structure of neural network loss landscapes”. In: *Advances in Neural Information Processing Systems* 32 (2019).

[11] S. S. Du, C. Jin, J. D. Lee, M. I. Jordan, A. Singh, and B. Poczos. “Gradient descent can take exponential time to escape saddle points”. In: *Advances in neural information processing systems* 30 (2017).

[12] J. Frankle and M. Carbin. “The lottery ticket hypothesis: Finding sparse, trainable neural networks”. In: *arXiv preprint arXiv:1803.03635* (2018).

[13] J. Frankle, G. K. Dziugaite, D. Roy, and M. Carbin. "Linear mode connectivity and the lottery ticket hypothesis". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 3259–3269.

[14] Y. Zhang, Z. Zhang, T. Luo, and Z. J. Xu. "Embedding principle of loss landscape of deep neural networks". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 14848–14859.

[15] X. Glorot and Y. Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

[16] J. Brea, B. Simsek, B. Illing, and W. Gerstner. "Weight-space symmetry in deep networks gives rise to permutation saddles, connected by equal-loss valleys across the loss landscape". In: *arXiv preprint arXiv:1907.02911* (2019).