EBERHARD-KARLS-UNIVERSITÄT TÜBINGEN
Wilhelm-Schickard-Institut für Informatik
Lehrstuhl Rechnerarchitektur

Bachelor thesis

myttile

Philipp Noel von Bachmann

Betreuer:	Prof. Dr. rer. nat. Andreas Zell Wilhelm-Schickard-Institut für Informatik
	Maximus Mutschler Wilhelm-Schickard-Institut für Informatik
Begonnen am:	
Beendet am:	July 11, 2020

Erklärung

Hiermit versichere ich, diese Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Tübingen am July 11, 2020

____

Max Mustermann

Contents

1. Introduction

1.1. Optimization in Neural Networks

1.1.1. Difference to normal optimization

Optimization is a core part of the current deep neural networks, as it is used in the learning process of the neural network. However, optimization in the context of deep learning differs from traditional optimization in several ways. In traditional optimization, we usually optimize on the data directly. However in deep learning, we don't have access to the test set. That's why we use the training set to optimize indirectly. Our hope is, that the distribution of the training set is close to the one of the later test set, so that reducing training error will result in reducing test error.

Formally, we want to reduce the test error given by

$$J(\theta) = E_{(x,y)\sim p_{data}} L(f(x;\theta),y) \tag{1.1}$$

where L is the Loss-Function, $f(x;\theta)$ the output of the network with respect to the input x and parameters θ, and y the labels for the input.This equation is known as **risk**. However, during the training process, we only have access to the training data which is distibuted according to $\hat{p}_{data}$ rather than p_{data}. Therefore, we can only optimize

$$J(\theta) = E_{(x,y)\sim \hat{p}_{data}} L(f(x;\theta),y) \tag{1.2}$$

To overcome this issue, we use a technique called empirical risk minimization.

1.1.2. Empirical risk minimization

As we have already seen, although we cannot reduce the generalization risk of equation 1.1, as we only have the training data. To convert the problem back to a normal optimization problem, we use the expectation over the empirical distribution. This is known as empirical risk minimization. Here, we use the mean as an unbiased estimator for the true mean of $\hat{p}_{data}$.

$$J(\theta) = E_{(x,y)\sim \hat{p}_{data}} L(f(x;\theta),y) = \frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)};\theta),y^{(i)}) \tag{1.3}$$

As the training set from the empirical distribution is restricted in size, this quickly leads to overfitting, where the network memorizes the training set. Additional measurements like Regularization have to be taken to account for that problem.

Another part in which we often optimize indirectly is the loss function $L(f(x;\theta),y)$. Take a classification task for example. An intuitive loss function would be the accuracy, which is computed by:

$$\frac{\text{\# of correct classifications}}{\text{\# of total classifications}} \tag{1.4}$$

When using this loss function with gradient descent however, this would lead to no useful gradient as it is 0 everywhere[??]. That's why we use another loss function like Cross-Entropy to optimize for this function indirectly.

1.1.3. Minibatch Algorithms

We have seen in equation 1.3 that we use the mean over the whole training set to approximate the empirical risk. From a computational perspective, this is rather expensive. That's why we normally only use a subset of the training set for each parameter update. These subsets are called batches. Some statistical considerations justify this.

The standard error of mean is given by $\frac{\sigma}{n}$, where n is the number of training examples. The root in the denominator results in that, when increasing the number of training examples, the standard error of mean will only decrease sublinear. That's why it is unattractable to use large batches of training data. The second factor is redundancy in the training set. As some examples might be quite similar to each other, the mean of a subset will not differ much from the whole training data, but requires much less computation time.

Most loss-functions allow us to divide the data into batches easily. The most common example is maximum likelyhood, which is defined as:

$$\begin{aligned} \theta_{ML} &= argmax_\theta p_{model}(X;\theta) && (1.5) \\ &= argmax_\theta \prod_{i=1}^{m} p_{model}(x^{(i)};\theta) && (1.6) \end{aligned}$$

If we convert it to log-space, the product decomposes into a sum:

$$\theta_{ML} = argmax_\theta \sum_{i=1}^{m} log p_{model}(x^{(i)};\theta) \tag{1.7}$$

In this space, we can easily divide the sum into batches and train on them seperatly. The same idea can be applied to the gradient:

$$\nabla_\theta J(\theta) = E_{x,y \sim \hat{p}_{data}} \nabla_\theta log(p_{model}(x,y;\theta)) \tag{1.8}$$

Optimization algorithms which use the whole trainig set are called **batch** or **deterministic** algorithms, while deterministic is prefered, as the term batch is also used in minibatch methods. The other extreme are **stochastic** or **online** methods, where only one training example is processed at a time. In between lie the methods, where more than one training example is used, but not all. These are called **minibatch** methods and are most commonly used in machine learning. An typical example for stochastic methods is stochastic gradient descent (SGD).

Both methods have their advantages other from the computational perspective. While large batches offer a more accurate estimate of the gradient, small batches can add a regularization effect. They add some noise to the gradient, and therefore leads the optimization algorithm to areas, where a small inaccuracy in the gradient converge to the same point. As the point is a very stable or wide minimum, this results in a regularization effect which decreases the generalization error. This is also supported by the work of Keskar et al. (2017), who argued that larger batches converge to sharp minimas, whereas small converge to flat. As we will argue later flat minima are believed to be better in generalization. Therefore, a small batch which ends up in a flat minima results in better generalization capabilities.

The effect of batch size is also sensitive to the choice of the optimization algorithm. In particular, second-order algorithms suffer from a small batch size. Hessian matrices H require a much larger sample size to be accurate than the Jacobian. Especially when H is of large condition number, this leads to the an amplification of the preexisting errors in g.

To get an unbiased esimate of the gradient, it is important to sample the mini-batches randomly. This is a problem in particular when the training data is corellated. This may arise as autocorellation where consecutive examples are corellated for example. The problem can be overcome by sampling the minibatches uniformly out of the training data. However, that would lead to a large computational effort every time we want to construct a batch. Fortunally, it seems sufficient to shuffle and divide the training data into batches only once.

Another property is that the gradient of minibatch algorithms like SGD follow the true gradient, as long as the training data is only used once. As each datapoint from the training set is only used once, it also follows the true data distribution p_{data}. Therefore we get an estimate of the true gradient of the generalization error. When reusing the data from $\hat{p}_{data}$, it no longer follows p_{data}, so the estimation becomes inaccurate.

1.1.4. Challenges in Neural Networks

In the last part, we demonstrated how optimization in deep learning differs from traditional optimization from a statistical point of view. This difference is emphasized further by other factors, in particular the non-convex loss landscape. This part will summarize some of these problems and their implications for deep learning.

Ill-Conditioning

???

Local Minima

In convex optimization problems, every local minima is guaranteed to be the global minima. Therefore finding a minimum is a sufficient enough condition to stop. In neural networks however, the loss landscape is highly non-convex. This results in a local minima possibly having higher cost value than the global minima. Proofs that these local minima exist can be constructed quite easily.

One example is the weight space symetrie. Suppose you swap out nodes i and j by swapping their incoming and outgoing weights. Then the activation in the next and all subsequent layers of the networks will stay the same, but the networks are located at different places in the loss landscape, therefore creating two local minimas. Although a large number of these local minima exist, they form no problem for optimization. As mentioned, the swap will lead to the same activation, therefore the networks will performe the on the data and the value of these local minimas will be equal.

However, there can also exist local minima with high cost compare to the global minimum. In theorie this was believed to be an issue, but in practice it seemed to cause no problem. Recently, some theoretical work support these findings. Chromanska et al. [2] applied spin glass theory from physics to neural networks. This allowed them to verify, that indeed nearly all local minima are of roughly the same cost and lead to the same test error. Furthermore, the propability of a local minima with high cost quickly diminishes as the network grows in size. One reason is that most local extrema are in fact saddle points.

Saddle Points

Saddle points are another type of local extrema where the gradient is 0. In contrast to local maxima or minima where the Hessian only has negative or positive eigenvalues, the Hessian of saddle points has both positive and negative eigenvalues.

In fact, saddle points get more common for higher dimensional space. The idea of coin flipping can be used to describe this phenomena. Suppose every eigenvalue of the Hessian is generated by flipping a coin. In a low-dimensional space, it quite likely that all of these flips will be positive or negative, resulting in minima or maxima. The higher the dimension gets however, the more likely the Hessian is to have both positive and negative eigenvalues. Therefore, there exist more saddle points. Chromanska et al. [2] also showed that these saddle points are more likely to occur in areas of higher loss.

What problems arise from saddle points depends in the choice of the optimization algorithm. For first-order algorithms, the situation is unclear. Although the gradient might get small in regions close to saddle points and as a result could slow down training, in pratice it seems like it isn't a problem for gradient descent. Especially when adding Momentum to the algorithm, it is very unlikely that that the gradient reaches 0, because there is no gradient in opposing direction which would decrease the momentum.

For second-order algorithms, saddle points are clearly a problem. Newton's algorithm for example explicitly solves for point with zero gradient, and is therefore attracted to saddle points or even other extrema like maxima. This is partly resolved by the introduction of saddle-free Newton.

Vanishing gradients

Today's neural network become very deep. But with increasing depth, there arises a problem called vanishing and exploding gradient. It refers to the fact that when back-propagating the gradient, it either vanishes or explodes. More formally, consider a matrix W which is multiplied by repeatedly on a computation path. If a eigendecomposition $Vdiag(\lambda)V^{-1}$ exists, this results in $(Vdiag(\lambda)V^{-1})^t = Vdiag(\lambda)^t V^{-1}$. Therefore, all values are scaled according to the eigenvalues of W. If these eigenvalues are smaller than 1, the gradient will vanish. If they are larger then 1, it will explode. This occurs as the network becomes increasingly deep, as the power of these eigenvalues is taken. One solution for this problem is the ResNet architecture, which will be described in detail in section **??**

Poor correspondence betwenn local and global structure

The previous sections have focused on which problems we are facing when computing a gradient or updating locally. However, the local structure can often be misleading, as it doesn't reflect the global one. Even if we are able to perform the best move locally and end up in a local minima, we are not guaranteed to be in the globally best area. [Figure 8.4] This can have various reasons like a bad starting point, where the initial gradient push us away into an area we cannot recover from.

Another issue may be that there even is no global minimum. This happens for example with the usage of the softmax, where the weights are increased without bound even when the accuracy is very good. This occurs because the actual labels for the softmax can only approach 1 with larger values, but never actually reach it. On solution to the second phenomena would be label smoothing, where instead of a hard 0-1 coding of the classification, each of the 0 are replaced by $\frac{k}{\text{\# of output units } -1}$, and the label for the true classification is replaced by $1 - \frac{1}{k}$. The network is now able to resemble the labels without extreme output values.

Solutions for this problem primarly focus on choosing the right inital values at the moment. [Maybe add solutions] The global structure will be discussed in detail in section 1.2.2.

Initilization strategies

1.1.5. Algorithms

Stochastic Gradient Descent

Stochastic Gradient Descent is one of the most popular optimization algorithm in deep learning. It is also one of the most basic ones, as it only takes the gradient at the current position into account. In contrast to normal gradient descent, SGD computes the gradient on minibatches, not on the whole dataset. Nevertheless, as long as the training data is only used once, SGD gives an unbiases estimate of the true gradient (see Section 1.1.3). Algorithm x shows a pseudo-code of SGD. [TODO: Add pseudo code]

Learning rate decay

One of the most important hyperparameters of SGD is the learning rate. While the optimal learning rate differs for every problem, it is important to decay the learning rate with the number of epochs. Initilally, it is good to choose a large learning rate. This leads to fast learning at the beginning, and avoids the algorithm getting stuck in high loss areas. As the number of epochs increases, it is important to shrink the learning rate. SGD only is a stochastic algorithm, therefore its gradient is inaccurate. Even when we find a local minima with a gradient of 0 on the minibatch, the true gradient will not be 0. A low learning rate secures to get close to the minimum while not overshooting it repeatedly.

[TODO: Formula 8.12, 8.13] How the learning rate is decayed varies from algorithm. A popular decay is step decay, where the learning rate is decayed by a constant factor γ after a fixed number of epochs.

$$lr = lr_0 \cdot \gamma^{\lfloor epoch/stepsize \rfloor} \tag{1.9}$$

Another popular algorithm is cosine decay, where the learning rate is decayed continously.

$$\varepsilon_t = \varepsilon_{min} + \frac{1}{2}(\varepsilon_{max} - \varepsilon_{min})(1 + cos(\frac{T_{cur}}{T_0}\pi)) \tag{1.10}$$

Here ε_{min} and ε_{max} define the range of the lr, $T_c ur$ is the current and T_0 is the maximum number of epochs. [Maximus paper]

Momentum

Momentum is a popular variation of SGD. It is used to speed up the training of SGD. The term momentum is used, as the underlying idea is the similar to the physical context. Consider a frictionless ball rolling down a hill. The ball builds up speed the further it rolls down by adding the acceleration of current gradient to the velocity. The ball will speed up, until it faces an uphill, where it will slow down again.

In context of deep learning, we use the velocity v rather than only the current gradient g to update the parameters.

$$v_{t+1} = \alpha v_t - \varepsilon g \tag{1.11}$$

Here, α controls how strong the past gradient is taken into account, while ε denotes the current learning rate. For the ball to build up velocity, it requires a constant downhill motion. The same is true for this case. The gradient can only build up, if it points in the same direction for some consecutive updates similar to a ball, which only speeds up when rolling downhill for an amount of time. Therefore, momentum speed up the gradients of parameters, whose gradient is constant in one direction. Parameters with alternating gradients for example will only experience small updates. The different orientations of their gradient will level out. Formally, if parameter p experiences the same gradient g every time, it will reach a terminal velocity of

$$\frac{\varepsilon \|g\|}{1-\alpha} \tag{1.12}$$

This also shows that α can be used to control the speed up of the training. If ε is kept constant, the larger α, the faster training will become. An value of 0.9 for example would lead to a speed up factor of 10, while 0.8 would lead to a speed up of 5.

Algorithm [TODO add algorithm] shows the implementation of momentum in SGD.

Momentum also adds an regularization effect, because it is attracted to stable or flat minima. If the minima is too small, it won't be able to stop the momentum and therefore the SGD will move on. That's similar to a ball, which won't stay in a small hole but keep on going, if it's speed is larger enough.

Nesterov??

1.2. Related work

1.2.1. Generalization gap

In section 1.1.1, we saw how learning differs from normal optimization, namely that we only have acces to the training set and thus have to optimize indirectly. With an increasing number of training epochs, the network starts to memorize the training data. Because the data distribution of the training data is not identical to the distribution of test data, this usually leads to a better performance of the network on the training set than on the test set. The difference between those performances is known as Generalization gap.

Strategies that are developed to deal with this problem are subsumed under the term **regularization**. One common Reguralizer is the L_2 Reguralizer. It's goal is, to encourage the weights to stay small. Small weights have some advantages for the generalization capabilities. First of all, small weights remove the dependency of a unit to one of it's inputs. Because the weights are really small, a strong activation cannot be solely achieved by the presence of one input, but rather has to rely on multiple units. Therefore small changes in the input will only cause small changes in the output, instead of the absence of one feature for example leading to a different output. This benefits the generalization, as the distribution of training and test data is slightly different. A network with L_2 regularization will nevertheless produce a quite similar output, in contrast to a normal one. Formally this can be incorporated in the loss function by adding the squared L_2 norm:

$$L = L(f(x;\theta), y) + \lambda \cdot \|\theta\|_2^2 \tag{1.13}$$

The paramter λ controls the strength of the L_2 and has to be adjusted for each problem.

Other work has gone into understanding the connection to the loss surface. Hochreiter & Schmidhuber [8] argued that flat minimas have better generalization capabilities. A flat minima is a region where the loss stays constant in conrtrast to sharp one, where small steps can increase the loss significantly. Therefore, flat minimas will perform constantly even for small changes in the input, whereas sharp minimas will lead to an increase in generalization error. Support also comes from the minimum description length theory, which states that fewer bits are needed to describe a flat minimum than a sharp. Lower complexity leads to a better generalization error. The idea is, that in the network we try to compress the data. The more we compress the data while also beeing able to resemble it, the more of the structure of the data we uncover. Therefore, a model with lower complexity can fit the underlying data better and achieve a better generalization error. Keskar et al. (2017) draw the same conclusion. They also provide a solution for finding flat minimas in using small batch sizes, see chapter 1.1.3.

Work from Dinh et al. [3] however contradicts this view. They argue that the notion of flat is problematic in the context of deep learning, as the loss surface is highly complex. Based on previous definitions from the papers above, they construct parameter values which lie on a sharp point of the surface, but are also able to generalize well. Therefore, at least some caution in needed when arguing about flatness beeing a reason for generalization capabilities.

1.2.2. Areas of same loss

In section 1.1.4, we showed that the poor correspondence between local and global structure may propose a major issue for optimization. Bad initializations may lead to path which moves away from the global otimum, often without chance to recover. Initialization strategies try to adress this problem, but cannot guarantee to solve it.

An open question is if this suboptimal structure is present in deep neural networks. Fengxiang et al. [6] showed that there exists infinetly many local minima which are of higher cost than the global one. Furthermore, these minima are arranged in cells, where each minima of one cell is of same loss as the others and also connected with them by valleys of low loss. These cells are seperated by nondifferentiable boundaries. Unfortunally, it remains unclear if these cells are of different cost. If this is the case, this would propose a major problem. When the training process gets into one of these cells, it is likely to get stuck, probably in an area of suboptimal cost. A trivial way to recover is not present at the moment.

Draxler et al. [4] get to a smiliar result, that local minimas are connected by valleys. In these valleys, the training loss stays the same to the one of the connected minima, while the test error rate slighly increases.

Fort & Jastrzebski [5] this in a more formal context. For each tuple of dimensions, they construct disks to describe the hyperplanes defined by them. They calls these hyperplanes wedges. One property is that the valleys of low loss described above lay on these wedges. Therefore the connecting valley for two points on different wedges has to pass through the intersection of them. When viewed from an cross-section, these valleys from a tunnel. Some techniques to improve optimization have similar effects on the size of these tunnels, namely they widen them. This happens for example for higher L_2 regularization 1.2.1, smaller batch sizes 1.1.3 or higher learning rates.

1.2.3. Cosine decay with warm restart

In section 1.1.5, we introduced learning rate schedulers. The idea was to have a high learning rate at the beginning for fast improvements, and then an decrease to fine adjust the parameters. On scheduler was cosine decay, with the formula 1.10. In the paper of Loshchilov & Hutter [11], they use this scheduler in combination with another technique, called warm restart. In constrast to the naive approach, where the learning rate is only decayed once, warm restart decayes until a fixed number of epochs, and then set back up to the inital learning rate. This procedure can be repeated for several times. The idea is, that in phases of high learning rate, the network explores the loss surface due to large steps. In epochs of low learning rate, the network exploits a small area to find the best parameter values. This may also add an regularization effect by letting the optimizer escape from unstable minima during high learning rate.

The authors report an new state of the art result at 3.14% test error for Wide-Residual-Net 28-20. Their method also perfoms better when compared to step decay.

1.2.4. Ensemble methods

The general idea of ensemble methods is to combine the predictions of multiple networks to get a more accurate prediction. The fact that this leads to an improvement can be seen in a simple regression problem. Suppose there exist k models that make an error of e_i with mean 0, variance $E[e_i^2] = v$ and covariances $E[e_i e_j] = c$ on every particular prediction. If these models are combined, the variance reduces to

$$E[(\frac{1}{k}\sum_i e_i)^2] = \frac{1}{k^2}E[\sum_i(e_i^2 + \sum_{j \neq i} e_i e_j)] = \frac{1}{k}v + \frac{k-1}{k}c \tag{1.14}$$

If all models make the same predictions, so $c = v$ for all combinations of models, then the sum decomposes to v, so the prediction error is the same as before. On the other hand, if $c = 0$, the prediction error is reduced by a factor of $\frac{1}{k}$. Therefore, for ensemble methods to be succesfull, every model has to achieve a low prediction error, while the predictions of all models should be as different as possible. While the low error is achieved in deep learning by standard training of the models, there are several approaches to ensure that these models are different.

The first idea would be to vary the training data for the model. One way this can be realized is by k-fold cross-validation. Here, we split the training data into k different smaller datasets. Then we train k independent networks on one of these subsets. This however decreases the size of the training set for each model drastically. Therefore another common approach is bagging [1], where we draw a subset from the training data, but with replacement. Here, individual examples might occur in more than one training set. Training on different datasets leads to different models, which we can use to our advantage as we have seen above. As the number of training examples is limited however, we might not be able to sustain a sufficient low error.

To use the whole training data while also creating different models, we can alternatively alter the model itself. An naive approach would be to just use different model architectures. If we want to use the same architecture for all models, we can vary the parameter initilializations. This is often enough to create models that have different predictions. However, for every initialization a network has to be trained from beginning, which can become very costly. A novel approach is to train one network, but to take snapshots of the network parameters at different steps. This approach adds no additional cost, as we only train one network. To ensure different networks, Huang et al. [10] use the method of cosine decay with warm restarts [11], as explained in section 1.2.3. Recall that at each restart, the learning rate is set up to the initial learning rate. This results in the optimizer taking larger update steps and consequently, the network parameter values will distance from their current state. The snapshots of the networks are taken before each restart, as the network converges to area of low loss when the learning rate is low. With this method, we are able to get different models with low error and no additional cost.

After we have ensured that the conditions for the models are met, we have to think about how to combine these predictions. For the case of classification with the use of softmax layer, we can use a technique called model averaging. Here, we sum the predictions of the individual models, and take the class with the highest prediction, as in standard classification. Formally, if the probability output of a model i for a given class c is p_{c_i}, then we sum the propabilites of each individual model: $p_c = \sum_i p_{c_i}$. To predict the class we take $argmax_c(p_c)$. If we have reasons to believe in a better prediction of one model over another, we can add weights w_i to the propabilites of the individual models: $p_c = \sum_i w_i \cdot p_{c_i}$.

2. Methods

2.1. Distance Function

2.1.1. Motivation

In section 1.2.2, we got a brief overview over the loss landscape of deep neural networks and the resulting challenges for optimization. Although there exists a large number of local minima, most of them have low cost. Furthermore, they are arranged in cells, where each minima has low cost and is connected to the others via valleys of low loss.

These cells create a challenge for optimization. Suppose the learning gets into the area of one of these cells. As mentioned in chapter 1.2.2 the cells are surrounded by nondifferential boundaries and areas of higher loss. [TODO: where is this stated?] This will likely cause the algorithm to get stuck into this cell. As all of the connected minimas in this cell are of approximatly same loss, there will be a boundary until the algorithm can improve, which may be higher than the global optimum. This imposes the question, if continuing to train is useful, as after one of the minima of the cell is found, the nearby minima it can reach will offer no significant improvement and other cells are out of reach.

If we reuse the ball analogy from section 1.1.5, the beginning of the training process would probably place the ball at the top of a mountain landscape. The initial training let's the ball move into one of the valleys. This valley may connect the ball to other points of low altitude, like minima are connected in a cell. When the ball is only allowed to move downhill however, the ball can never reach a spot which he is seperated by a hill. Thus, his minimum altitude is bound to the lowest place in his current valley. If we add momentum, we have seen that the ball is able to get over hills of certain size by using his momentum. Warm restarts add the possibility to jump over a hill with a high learning rate at each restart. This can lead to an escape of the cell, but without guarantee. What is more likely to happen is that after a large step, the ball is in an area of higher altitude, which will let him roll down to the same valley again in the next step.

In contrast, we try a different approach. Rather than letting the ball make one large steps and then allwoing him to move on freely, we want to constantly push the ball away from it's current position. The idea beeing if the ball is only surrounded by hills, we have to push it up one of these until it reaches the top and can then roll down into another valley, therefore escaping the cell.

Transferred back to a real network, this translates to repeatedly updating the parameter values in a way, that the new values will increase their distance to the values we want to get away from.

In Pseudo-Code, this looks the following:

First, we train the network the traiditional way - we compute the foward and backward pass of the training data with the resulting gradient, and then update the parameter values with an optimizer. After we have reached a sufficiently low error, we create the point we want to push away from, which we call **checkpoint**. The difference in the next training loop in contrast to the standard one at the beginning is how the parameter values are computed. Here, our goal is to distance from the checkpoint. This will also result in a gradient, which will be applied the

Algorithm 1 Network training with distancing

```
Require: a neural network architecture and a dataset
Return: a trained neural network
initialize the network, dataset and training parameters
for $i \leftarrow 1$ to desired number of epochs do
    compute foward and backward pass of training data
    update parameter values with optimizer
end for
create checkpoint we want to distance from
for $i \leftarrow$ next epoch to end do
    compute new parameter values different to checkpoint
    update parameter values with optimizer
end for
return: the trained network
```

same way as above.

How can we encourage a network to distance from a checkpoint, while at the same time not sacrificing the networks performance. Back in the ball analogy, we instead of pushing the ball up a hill, we could place a hill at the current position, and let the ball roll down. Figure 2.1.1 shows a graphical illustration for the 2D case. An advantage of this technique is that the hill will only change the loss landscape locally. So when we have distanced from the hill, we follow the normal loss landscape again.

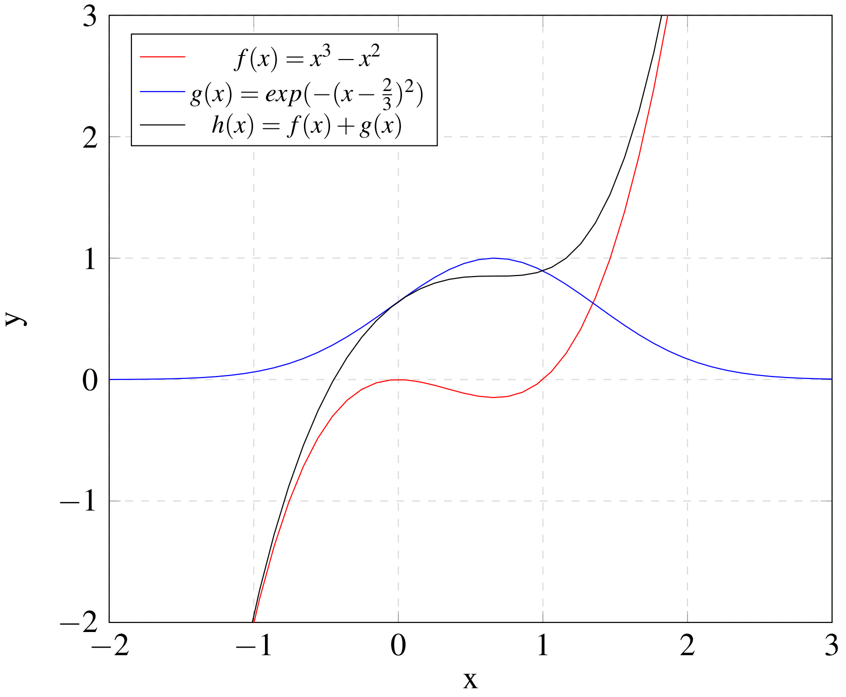

Figure 2.1.: Example of a function with a local minimum at $x = \frac{2}{3}$. If we place an Gaussian function on top, the minimum disappears. Note, that the black function approaches the red if we distance from the local minimum.

Formally, this hill is realised by adding first computing the distance of the parameter values to the checkpoint, and then adding a penalty for small distances. If we use and exponential Kernel, the idea of the hill can be taken quite visually. Section 2.1.2 describes the mathematical approach in detail.

What happens when our penalty hill is not large enough to get the ball over the hill? This means we would get back to the present parameter values. That's not necesarryly a problem, as it would mean that our current position is very robust, which would mean a good generalization capability. Therefore our network would stay in areas with stable minima, which is a desired property.

For ensemble methods, there are other possible benefits of this approach. Recall that for an ensemble to perform well, the errors of the participating networks should be uncorellated. SGD with warm restart [11] tried to ensure this with warm restarts, where a high learnin rate increases the distance of the networks and therefore leads to uncorellated errors. We could use our method as alternative approach, where we ensure the distance between the networks not only by high learning rate but by explicitly increasing the distance. Of course, the underlying idea for both is that networks which are more distant in parameter space will behave more differently, and therefore have more uncorellated errors.

2.1.2. Mathematical approach

Distance function

As we want to measure the distance between two points, we need to define a distance function. A common choice is the euclidean distance also know as L_2 norm, which is defined as:

$$\|x\|_2 = \sqrt{\sum_i |x_i|^2} \tag{2.1}$$

This norm can also be squared to get rid of the root. Squaring does not change the direction of the gradient, and is therefore possible. To measure the distance between two parameter states θ_1 and θ_2 of the networks, this results in:

$$d(\theta_1, \theta_2) = \sum_i (\theta_{1_i} - \theta_{2_i})^2 \tag{2.2}$$

The size and shape of the paramters θ is not important, as each paramter is only compared to another state of itself, and is combined via a sum.

Another property is that the values the distance function can take is partly dependend on size. Consider two networks θ_1, θ_2 with the same classification task, but the size of θ_1 is larger than θ_2.[may be confusing with before] If we assume all of the parameters are distributed the same way, then θ_1 would output a larger distance than θ_2. However it would be desirable for the functions to be in the same bound, as it would make the transfer of hyperparameters for example possible. That's why we use a function to control for the output to be in a certain bound.

If we take a look at support vector machines, they use kernels to compute the similarity between two samples. One popular choice is the radial basis function kernel, defined as:

$$k(\theta_1, \theta_2) = exp(-\frac{\|\theta_1 - \theta_2\|^2}{2\sigma^2}) \tag{2.3}$$

where *exp* is the exponential function. With the use of 2.2, we can convert this to:

$$k(\theta_1, \theta_2) = exp(-\frac{d(\theta_1, \theta_2)}{2\sigma^2}) \tag{2.4}$$

If we plot this function in the two dimensional case (see Figure 2.1.2), we can see some nice properties. First of all, the values are now bound between 0 an 1, regardless of the size of θ. Second, we can see as the values of the distance get larger, the function approaches 0 asymtotically. This leads to a really small gradient for extreme distances. Consequently, the Kernel will initially push the parameters away from the checkpoint, but when this is achieved, will have little influence on the loss function. How far the function encourages to distance from the checkpoint can be controlled by the parameter σ, which defines the width of the function and can be tuned as a hyperparamter. As σ gets larger, the function becomes wider. Therefore, the influence of the distance function will reach further the larger σ is.

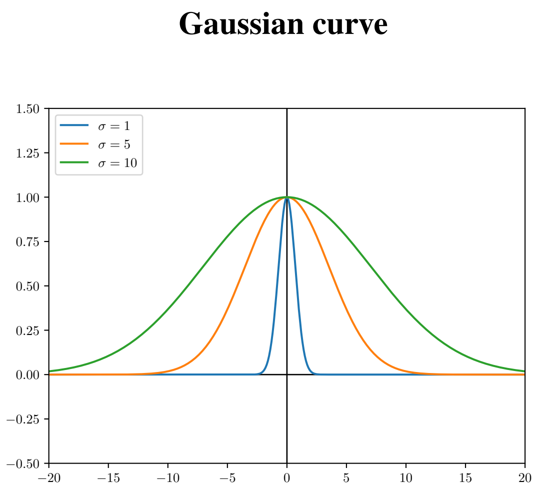

Figure 2.2.: Plot of the function $f(x) = exp(-\frac{x^2}{\sigma^2})$ for different values of σ.

Loss function

This section will show how the distance function from 2.1.2 is combined with the normal loss function. The state of the art loss function for image classification, which will be used for testing is the cross-entropy-loss defined as:

$$-\sum_i \delta_{yc} log(f(x)_i) \tag{2.5}$$

Where δ is the Kronecker-Delta function defined as:

$$\delta_{xy} = \begin{cases} 1, \text{ if } x = y \\ 0 \text{ otherwise} \end{cases} \tag{2.6}$$

To account for the distance term, we just sum it with the cross-entropy-loss:

$$L = \sum_i \delta_{yi} log(f(x)_i) + distance(\theta, \theta_c) \tag{2.7}$$

Where θ_c is the checkpoint. Note that we can do this multiple times, so we can incooperate multiple checkpoints:

$$L = \sum_i \delta_{yi} log(f(x)_i) + \sum_c distance(\theta, \theta_c) \tag{2.8}$$

When computing the derivative for the backpropagation, the sum decomposes into two terms, so the cross-entropy-loss will be computed the same as before. Another property we want to control for is the influence of the distance versus the cross-entropy-loss. When training is in later stages, the cross-entropy-loss may be very small. If the values of the distance function a too large in comparison, this would cause the parameters to be updated only based on the distance, which is undesirable as the performance wouldn't be taken into account anymore. The same is also true the other way around, if the distance function is too small, it wouldn't affect training at all. That's why we introduce a hyperparameters w called weight to control this:

$$L = \sum_c \delta_{yc} log(f(x)_c) + w \cdot distance(\theta, \theta_c) \tag{2.9}$$

2.1.3. Pytorch implementation

Checkpoint creation

To measure the distance, we have to create the checkpoint. The model parameters in pytorch are stored as matrices for each layer and can be accesed via $model.parameters()$, which outputs an iterable. We therefore opt to keep this structure and safe the parameters in a list.

Algorithm 2 Checkpoint

```
import torch

def create_checkpoint(model):
    checkpoint = []
    for param in model.parameters():
        newparams = param.clone().detach().to(device='cuda')
        checkpoint.append(newparams)

    return checkpoint
```

First, the checkpoint list is initialized. Then we iterate over the model parameters. For each layer, we have to clone the parameters in order to create new variables, and not just pointers to the existing ones. In addition they have to be detached, to remove connection of the gradient to the model parameters.

Distance function

The L2 norm is implemented the follwing way: We iterate over the checkpoint and the current parameter values. For each layer, we compute the difference, and then square and add the values to our distance.

2.2. Configuration

2.2.1. Library and Training

The code for the network was written in Python [Version], with the use of Pytorch [12] as the machine learning library. Data that occured during training was logged and plotted with

Algorithm 3 L2 norm

```
import torch

def computedistance(checkpoint, model):
    distance = 0
    for checkpoint_param, param in zip(checkpoint, model.parameters()):
        difference = torch.add(-param, checkpoint_param)
        distance += torch.sum(torch.mul(difference, difference))
    return distance
```

Tensorboard. For a detailed overview of the code, see Appendix [add]. Training was done on the TCML Cluster of the university of Tübingen.

2.2.2. Datasets

Two common datasets were used.

- MNIST
  MNIST is a classical example of hand-written digits from 0 to 9 that have to be classified accordingly. It contains a train set of 60000 and a test set of 10000 examples.

- CIFAR-10 This dataset consists of 60000 32x32 colour images with 10 different classes. The dataset is seperated into 50000 train and 10000 test images. The state of the art accuracy is 97.3%, reported from Kolesnikov et al. (2019). The dataset can be accessed via https://www.cs.toronto.edu/ kriz/cifar.html.

2.2.3. Networks

ResNet

The Residual Network (ResNet) architecture was first proposed by He et al. [7]. The idea of the ResNet was, that it is more easy for a network to learn the residuals rather than the full transformation of an input. Formally, consider the input x. The network transforms x accoring to $H(x)$. Rather than letting the network do the full transformation, ResNet produces $H(x) = x + f(x)$, where $f(x)$ is the residual transformation learned by the network, and x the identity data which is realized by a skip connection. Figure 2.2.3 shows a graphical illustration. Beside the assumably easier to learn representation, ResNet also solves other problems. The issue of the vanishing gradient as discussed in section 1.1.4 for example is tackled, as skip connections backpropagate the gradient better to earlier layers of the network. This allows for deeper networks.

ResNet creates basic building blocks by applying a convolution, a Relu and another convolution before adding the skip connection with a final Relu. Figure 2.2.3 illustrates this block. The blocks are then stacked after each other. To let the network learn a more complex representation, after a number of block, they increase the number of channels. Here, the identity mapping of the skip connection is replaced by a pointwise convolution to increase the number of channels. Alternatively, the new channels are padded with 0.

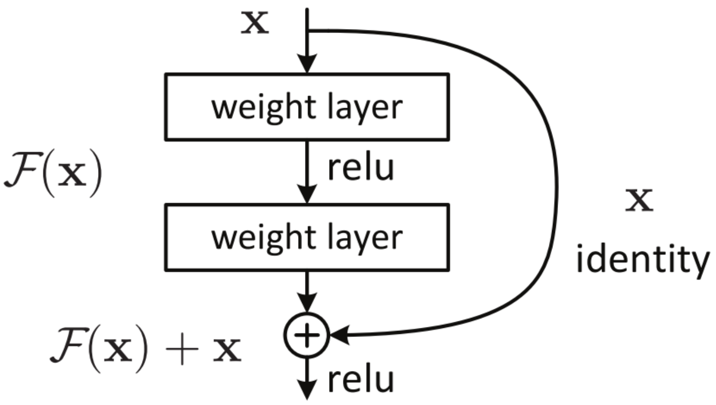
x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 2.3.: Residual Block from [7, Page 2]

MobileNetV2

MobileNet was first introduced by Howard et. al [9] as a lightweight neural network for the use on mobile devices. The reduce the computation effort of the network, the made use of the depth-wise seperable convolution. Consider a 32x32x3 image of the CIFAR-10 dataset. A traditional 3x3 convolution with a stride and padding of 1 would produce an output of $\frac{32+2-3}{1}+1$ the shape 32x32x1. To get more channels, we would need more kernels. For a total of k output channels, we would need to do 3x3x3 x32x32 xk multiplications. Instead of doing the computation in one kernel, depth-wise seperable convolution divides it into two kernels. First they perform a depthwise convolution, where a kernel of 3x3x1 is applied to every channel of the input, so in this case we end up same size of 32x32x3. To get to k channels, they use a pointwise convolution across the channels. This is a 1x1x3 convolution, which upscales the image to more channels. So if we want k output channels, we need k 1x1x3 kernels. The benefit of this technique is that, while getting the same output size, we only need 32x32 3x3x3 multiplications in the first step, and 32x32 1x1x3 xk in the second step. This is way less than the from the beginning. Figure 2.2.3 shows a graphical illustration of this idea.

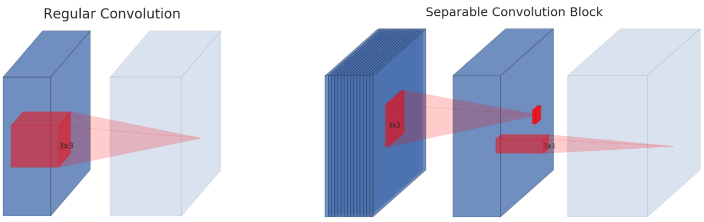

Figure 2.4.: Depthwise Separable Convolution from [13, Page 3]
On the left, a regular convolution with a 3x3xDepth kernel is drwan. On the right, this convolution is separated in depthwise convolution with a 3x3x1 Kernel, followed by a pointwise convolution with a 1x1xDepth Kernel. This methods saves computation time over the standard method, while also allowing interactions between channels.

With the second iteration called MobileNetV2 [13], they added the concept of inverted residuals or as they call it bottlenecks. The ideas is, that intermediate representations should be low rather than high dimensional. The bottleneck layer starts with a matrix with a small number of depth channels. In the first step, the representation is expanded by a pointwise convolution, followed by a Relu6. The expansion factor here controls how much channels the intermediate representation will have. Then a depthwise convolution is applied as an transformation step, followed again by a Relu6. Finally, a pointwise convolution is applied again, but this time without a nonlinear transformation following afterwards. The idea is that when compressing the data back to a low dimensional space, a nonlinearity would only lead to a loss of information. Finally, a skip connection from the input to the output is applied to get a residual mapping from input to output.

For ImageNet, MobileNetV2 reaches an top-1 accuracy of 72%, which outperforms the competitors like MobileNetV1 or ShuffleNet 1.5 while having a similar number of parameters.

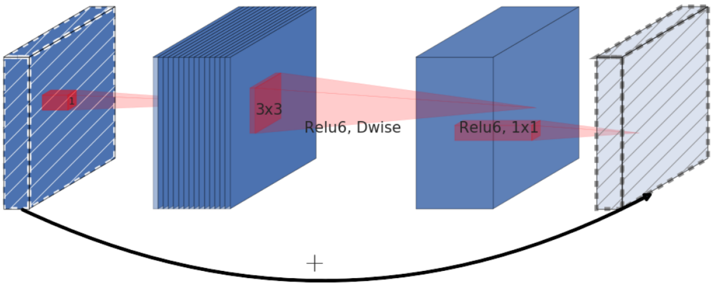

Figure 2.5.: Inverted Residual Block from [13, Page 3]
First, a Relu 6 followed by a Pointwise Convolution is applied to map the data to a higher dimensional space. Then a depthwise convolution is applied, followed again by a Relu6. Finally, the data is compressed back to the original size. This time, there is no Relu applied to avoid the loss of information.

2.2.4. Network hyperparameters and training loop

For both networks, the base hyperparameters are the same. Note that these parameters are only standard configurations and may be changed to investigate the effects:

- Optimization Algorithm
  Stochastic Gradient descent combined with Momentum is used. A λ of 0.9 is set for Momentum.

- Learning rate
  Initially 1e-2

- Learning rate scheduler
  Both step decay and cosine decay are used. For step decay, the epochs until a warm restart are divided into 3 parts and in every part, the learning rate is decayed by a factor of 0.1. For other learning rates, see 3.6.1.

- Warm restart
  A warm restart is performed after 90 epochs (See 3.6.2 for changes). The learning rate is set back up to the initial learning rate.

- Regularization
  An L_2 Regularization as described in chapter 1.2.1 is used with a factor of 1e-3.

- Checkpoints for Distance function
  Every time a warm restart is performed, a new checkpoint for the distance function is created.

- Number of epochs
  The number of epochs varies. As this is an explorative analysis, the training is often run for a large number of epochs, to investigate long term changes. Usually an epoch number of 900 is used.

3. Results

Even with such a simple formulation, there are many different variations that can be tested. First is the influence of the strength and width parameter of the distance function, which will be tested in section 3.2. We will also briefly test different kernels besides the RBF Kernel from equation 2.3 in section 3.3. In section 3.5, we will use more than one checkpoint and will also try to combine these to a global one. The interaction of training hyperparameters like learning rate and epochs will be discussed in section 3.6. Finally, the usage for ensemble methods will be tested.

3.1. Baseline

3.1.1. Base Network

For the baseline, we use the hyperparameters as defined in section 2.2.4. Figure 3.7 shows the result, if we add a checkpoint after 100 epochs of training. For the validation accuray, we can see that it stays similar to the accuracy without the distance function. If we plot the distance the network gets to the checkpoint however, we can see that the network trained with distance function clearly increases its distance more than the one trained without. Furthermore, the red plot follows the shape we would expect from the distance function. At the beginning, the distance increases relatively fast, as can be expected from the high value of the RBF Kernel. But as the distance increases, its gradient becomes smaller, similar to the gradient of the Kernel. The blue plot in contrast follows an almost linear curve, whith a much smaller gradient. Therefore, the additional term succeeds in the initial goal, which was to distance from the checkpoint.

scheduler

3.2. Distance function Hyperparameters

Recall the formulation of the distance function from equation 2.4:

$$k(\theta_1, \theta_2) = exp(-\frac{d(\theta_1, \theta_2)}{2\sigma^2}) \tag{3.1}$$

and the total combination with the ordinary loss function from equation 2.9:

$$L = \sum_c \delta_{yc} log(f(x)_c) + w \cdot distance(\theta, \theta_c) \tag{3.2}$$

Here, we can see the two hyperparameters width σ and weight w, whose influence will be investigated in this chapter.

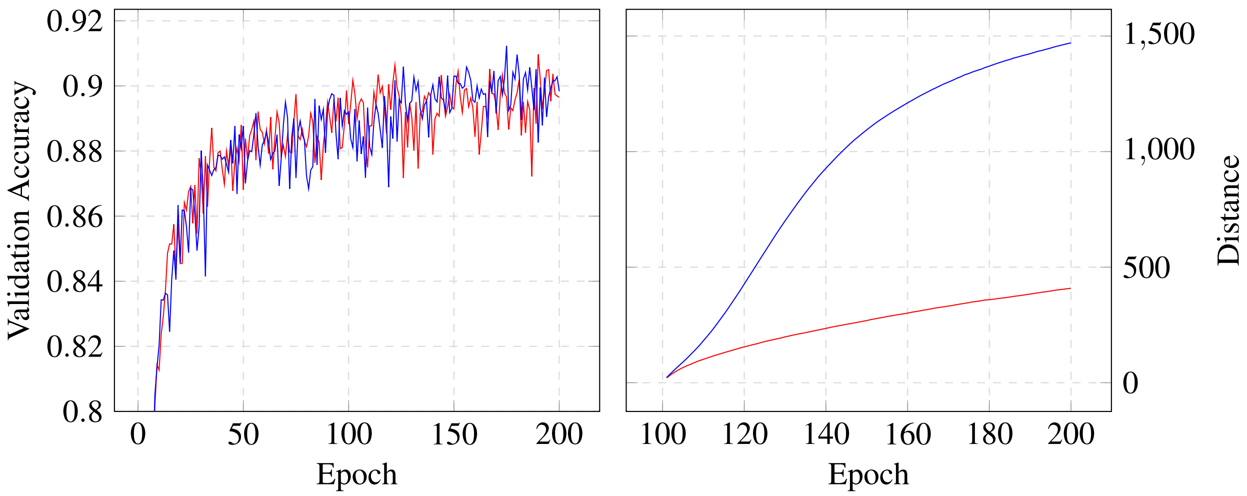

Figure 3.1.: Validation accuray and Distance values for a network trained without distance function (red) and with distance function (blue).

3.2.1. weight

3.2.2. width

We have seen in chapter 2.1.2, how the width σ influences the distance function: The larger σ gets, the wider it becomes. Therefore, a larger σ should result in the network distancing more from it's checkpoint than for smaller values.

3.3. Kernel

3.4. Regularization

3.5. Multiple Minimas

3.5.1. multiple

3.5.2. merge

3.6. Training Hyperparameters

3.6.1. lr

wrong lr and how scheduler is used

3.6.2. epochs

epochs and epoch time

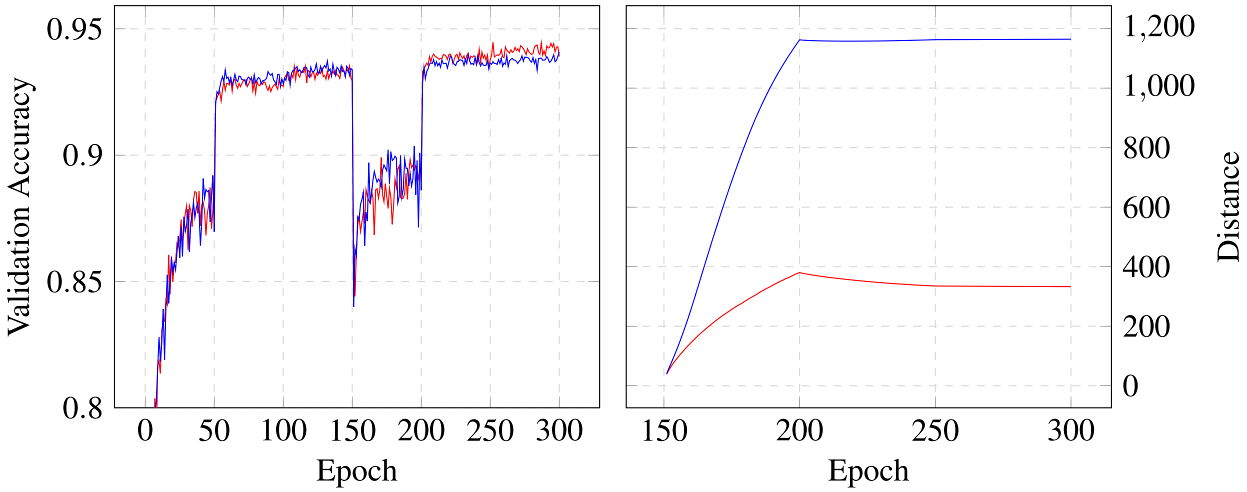

Figure 3.2.: Validation accuray and Distance values for a network with a step scheduler and warm restarts. The learning rate is dropped after 50 epochs and restarted after 150. The red curve shows a network trained without distance function and the blue curve with distance function.

3.7. ensemble methods

example picture:

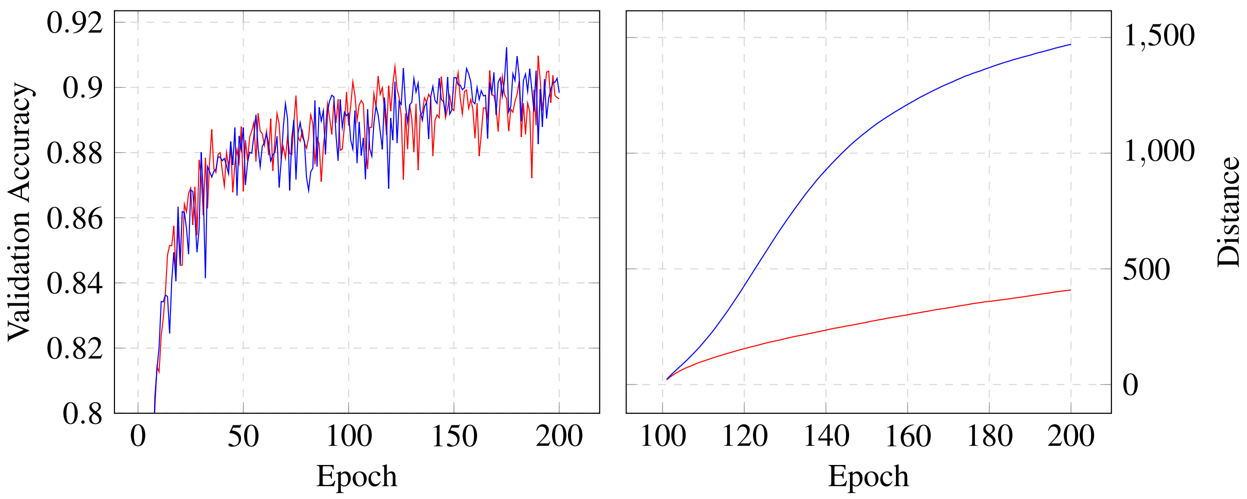

Figure 3.3.: Validation accuray (left) and Distance values (right) for a network trained without distance function (red) and with distance function (blue).

4. Dicussion

5. Summary

A. Abkürzungsverzeichnis

tRNA	Transfer-RNA

Bibliography

[1] L. Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[2] A. Choromanska, M. Henaff, M. Mathieu, G. B. Arous, and Y. LeCun. The loss surfaces of multilayer networks. In *Artificial intelligence and statistics*, pages 192–204, 2015.

[3] L. Dinh, R. Pascanu, S. Bengio, and Y. Bengio. Sharp minima can generalize for deep nets. *arXiv preprint arXiv:1703.04933*, 2017.

[4] F. Draxler, K. Veschgini, M. Salmhofer, and F. A. Hamprecht. Essentially no barriers in neural network energy landscape. *arXiv preprint arXiv:1803.00885*, 2018.

[5] S. Fort and S. Jastrzebski. Large scale structure of neural network loss landscapes. In *Advances in Neural Information Processing Systems*, pages 6709–6717, 2019.

[6] F. He, B. Wang, and D. Tao. Piecewise linear activations substantially shape the loss surfaces of neural networks. *arXiv preprint arXiv:2003.12236*, 2020.

[7] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

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

[9] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[10] G. Huang, Y. Li, G. Pleiss, Z. Liu, J. E. Hopcroft, and K. Q. Weinberger. Snapshot ensembles: Train 1, get m for free. *arXiv preprint arXiv:1704.00109*, 2017.

[11] I. Loshchilov and F. Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

[12] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[13] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.