EBERHARD-KARLS-UNIVERSITÄT TÜBINGEN
Wilhelm-Schickard-Institut für Informatik
Lehrstuhl für Kognitive Systeme

Bachelor thesis

Escaping local minima in DNNs by rewarding different parameter values

Philipp Noel von Bachmann

Betreuer: Prof. Dr. Andreas Zell
Wilhelm-Schickard-Institut für Informatik

Maximus Mutschler
Wilhelm-Schickard-Institut für Informatik

Begonnen am: May 5, 2020

Beendet am: August 4, 2020

Erklärung

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

Tübingen am August 4, 2020

_____

Max Mustermann

Contents

1 Introduction

1.1 Optimization in Neural Networks

Optimization is a core part of the current deep neural networks, as it is the foundation of the learning process. However, optimization in the context of deep learning differs from traditional optimization in several ways. This section focuses on these differences and other current challenges of optimization in deep neural networks. Most of the sections are inspired by [8].

1.1.1 Difference to traditional optimization

In traditional optimization, we usually optimize on the data we want to perform later directly. However in deep learning, we usually don't have access to the test data. Consider autonomous driving. Here, the data the self-driving agent has to act on will be generated while driving, with no chance of getting it in advance. But we can capture data of other cars and optimize on them indirectly. The hope beeing, that the distribution of the training set is similar 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 set 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**. The arimethic mean serves 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. Therefore, additional measurements like Regularization have to be taken into account.

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 for classification is maximum likelyhood, which is defined as:

$$\theta_{ML} = argmax_\theta \, p_{model}(X;\theta) \tag{1.4}$$

$$= argmax_\theta \prod_{i=1}^{m} p_{model}(x^{(i)};\theta) \tag{1.5}$$

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

$$\theta_{ML} = argmax_\theta \sum_{i=1}^{m} log\, p_{model}(x^{(i)};\theta) \tag{1.6}$$

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:

$$\nabla_\theta J(\theta) = E_{x,y\sim\hat{p}_{data}} \nabla_\theta log(p_{model}(x,y;\theta)) \tag{1.7}$$

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. A typical example for stochastic methods is stochastic gradient descent, see section 1.1.5.

Both methods have their advantages beyond 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 lead the optimization algorithm to areas, where a small inaccuracy in the gradient converges 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. [14], who argued that larger batches converge to sharp minimas, whereas small converge to flat. Section 1.2.1 shows, that 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. For example autocorellation can arise, when data is taken from a time series. 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 stochastic gradient descent 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. 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.

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 same on the data and the value of these local minimas will be equal.

However, there can also exist local minima with high cost compared 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. [3] applied spin glass theory from physics to neural networks. This allowed them to verify that for large networks, local extrema with higher cost are exponentially likely to be saddle points. Therefore, all local minima are approximately of the same cost, which is close to the global optimum and it is very unlikely to get stuck in a local minimum with high cost.

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. [3] 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 becomes 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 between 0 and 1, the gradient will vanish. If they are larger then 1, it will explode. This especially occurs when 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 2.2.3

Poor correspondence between 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. Local gradients can often point away from or costruct a subotimal path to areas of low loss. 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 1.1.4 shows an example of suboptimal initialization. As the local structure for $x < 0$ doesn't reflect the global, these initializations will not lead to the global minimum. Therefore, theretical work has gone into finding good initialization strategies.

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 $1 - k$. The network is now able to resemble the labels without extreme output values.

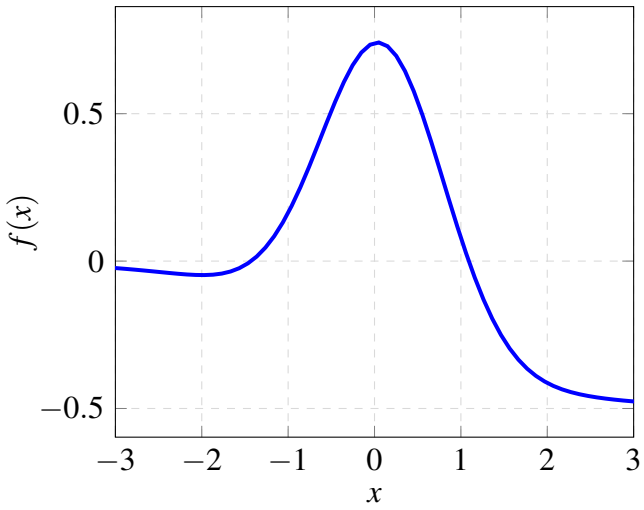

Figure 1.1: Example of poor correspondence between local and global structure. If $x < 0$ is initialized, gradient descent will lead us in negative direction. However, this leads away from the global optimum on the right.

Initialization strategies

In the last section, the poor correspondence between local and global structure was shown. Initializations at the wrong place on the loss landscape lead to a gradient descent in a direction away from the global optimum. But because we don't know the shape of the loss landscape in advance, no initialization can guarantee a good solution or even convergence at all. However, some strategies seem to perfofm better than other and have become widely used.

The first and only known property of initialization is that is has to be asymetric. This means that two units that share the same input must have different weights attached to these inputs. If this is not the case, a deterministic optimization algorithm will update both these parameters in the same way. To break this symetry, the weights are initialized randomly. If a gaussian or uniform distribution is used doesn't seem to matter.

However, the range of these functions is important. Wide distributions have a stronger symetry breaking effect, but come with the risk of exploding gradients as discussed in section 1.1.4. The problem of vanishing gradients arise if the distribution is too small. The statictical viewpoint suggests a initialization to small weights nevertheless. Here, large weight initialization is seen as a large gaussian prior on which and how the units interact. As we have no reason to encourage one interaction over another, the weights should be as small as possible.

Some heuristics try to balance these different motivations. One of the most famous one is the normalized initialization from Glorot [7]. For a fully connected layer of m inputs and n outputs, the weights should be initialized according to a uniform distribution:

$$W \sim U(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}) \tag{1.8}$$

The biases are normally initialized to a constant. A value of 0 seems to behave well for most applications.

Stopping criteria

In traditional optimization, simple heuristics are used to determine the point of stopping, like a gradient of 0. However many local minima exits on the loss landscape of deep neural networks, making a stop at the first extremum unattractable. In addition, batch algorithms only give an approximation of the true gradient, so we never find the true gradient of 0.

Therefore, other stopping criteria have been developed. What we normally see in terms of performance is an initial decrease in training loss paired with an increase in validation accuracy. But after some epochs, we enter the overfitting region, where training loss keeps decreasing but validation accuracy also decreases again. In early stopping, this is used the determine the stopping point. After each epochs we store a copy of the parameter values. Then we train the network for a fixed number of epochs. At the point when the validation error starts to rise again, we can return to the parameters before. As no additional computation time is required, this is a very efficient method from a computational perspective, but may require some extra storage. This apporach is very different to traditional optimization,as gradients may still be very large.

1.1.5 Algorithms

Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) 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 unbiased estimate of the true gradient, see Section 1.1.3.

Algorithm 1 Stochastic gradient descent from [8]

Require: learning rate ε
Return: a trained neural network
1: initialize the network, dataset and training parameters
2: **while** stopping criteria is not met **do**
3: sample minibatch of m examples $x^{(1)},...,x^{(n)}$
4: compute gradient estimate $\hat{g} = \frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)};\theta),y^{(i)})$
5: apply parameter update $\theta = \theta - \varepsilon \cdot \hat{g}$
6: **end while**
7: **return: the trained network**

First, the network is initialized. Then, the training loop repeats, until the stopping criteria is met. At the beginning of each iteration, a minibatch is sampled. Then, the gradients are calculated, multiplied by the learning rate ε and finally applied to update the parameter values.

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. Initially, it is good to choose a large learning rate. This leads to fast learning at the beginning, and avoids the algorithm getting stuck in areas of high loss. 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 small learning rate secures to get close to the minimum while not overshooting it repeatedly.

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_{cur} is the current and T_0 is the maximum number of epochs. The advantage of cosine decay is that the learning rate is decayed down to 0 in the defined interval, independent of the initial learning rate. In contrast, the smallest learning rate of step decay is dependent of the initial one.

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, because 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.

In pseudo code, SGD with Momentum looks similar to normal SGD 1.1.5. The only difference is line 5 and 6, where the velocity is updated and then used to update the parameters rather than the gradient itself.

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.

Algorithm 2 Stochastic gradient descent with Momentum from [8]

Require: learning rate ε
Require: momentum parameter α
Return: a trained neural network
1: initialize the network, dataset and training parameters
2: **while** stopping criteria is not met **do**
3: sample minibatch of m examples $x^{(1)},...,x^{(n)}$
4: compute gradient estimate $\hat{g} = \frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i);\theta}), y^{(i)})$
5: compute velocity update $v = \alpha v - \varepsilon \hat{g}$
6: apply parameter update $\theta = \theta - v$
7: **end while**
8: **return: the trained network**

1.2 Related work

1.2.1 Wide minima and Generalization

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. 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 individually for each problem.

Other work has gone into understanding the connection to the loss surface. Hochreiter & Schmidhuber [11] 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. [14] 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. [4] 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 is needed when arguing about flatness beeing a reason for generalization capabilities.

1.2.2 Loss landscape

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. [9] showed that there exist infinitly 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. [5] 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 [6] 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, and in their direct connection is an area of high loss. 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 [17], 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. At every restart, the performance becomes worse for some epochs due to high learning rates. But when the learning rate decays, the previous performance is reached and even topped.

The authors report an new state of the art result at 3.14% test error for Wide-Residual-Net 28-20 on CIFAR-10. 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 [2], 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. [13] use the method of cosine decay with warm restarts [17], 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 probabilites 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 probabilites 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 exist 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. 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 allowing 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.

Algorithm 2.1.1 shows the formalization of the idea into pseudo-code. 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 same way as above.

Algorithm 3 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 ← 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 ← next epoch to end do
    compute new parameter values different to checkpoint
    update parameter values with optimizer
end for
return: the trained network
```

How can we encourage a network to distance from a checkpoint, while at the same time not sacrificing the networks performance? Back to the ball analogy, 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. The function $f(x)$ has a local minimum, where an algorithm might get stuck. If we add a gaussian curve $g(x)$ with the maximum aligned to the minimum of $f(x)$, we can see that the minimum disappears. 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.

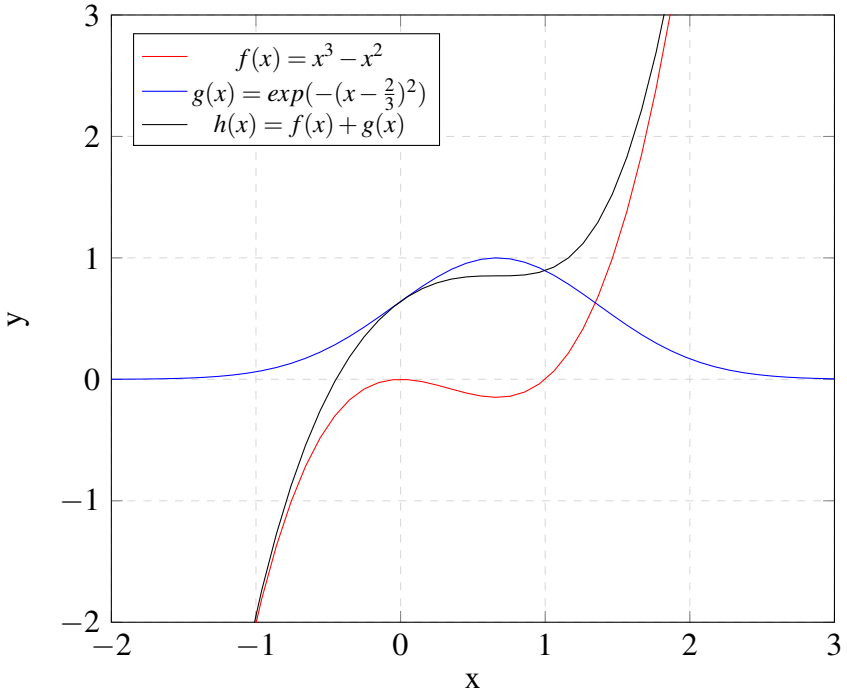

Figure 2.1: Example of a function with a local minimum at $x = \frac{2}{3}$. If we place an Gaussian function on top, the minimum disappears. Note, that the black function approaches the red if we distance from the local minimum.

Formally, this hill is realised by 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 lead to 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 [17] 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:

$$L_2(\theta_1, \theta_2) = \sum_i (\theta_{1_i} - \theta_{2_i})^2 \tag{2.2}$$

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.

An undesireable property of the current formulation is that the values the distance function can take is partly dependend on the size of the network. Consider two networks θ_1, θ_2 with the same classification task, but the number of parameters of θ_1 is larger than θ_2 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 need a function to let the output stay in a certan 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:

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

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

$$distance(\theta_1, \theta_2) = exp(-\frac{L_2(\theta_1, \theta_2)}{\sigma^2}) \tag{2.4}$$

If we plot this function in the two dimensional case in 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.

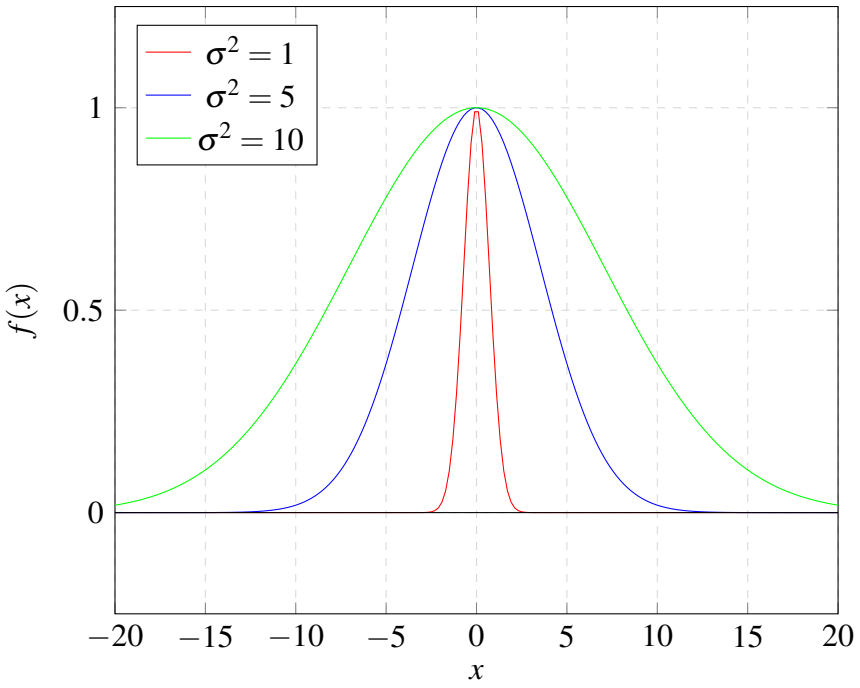

Figure 2.2: Plot of the function $f(x) = exp(-\frac{x^2}{\sigma^2})$ for different values of σ^2.

Loss function

The last section has focused on how to implement a loss function which achieves distancing from parameter values. However, only using this loss function would sacrafice network performance. Therefore, we combine it with the standard loss function of the task. The state of the art loss function for image classification, which will be used for testing later, 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. Here, we divide by the number of checkpoints to keep the influence of the distance function on the loss function irrelevant of the number of checkpoints.

$$L = \sum_i \delta_{yi} log(f(x)_i) + \frac{1}{c} \sum_c distance(\theta, \theta_c) \tag{2.8}$$

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 s called strength to control this:

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

Figure 2.1.2 shows the infuence of s on the distance function. The values are now bound between 0 and s. Increasing the strength also has the same effect of the function getting wider as in figure 2.1.2, but at a much smaller scale.

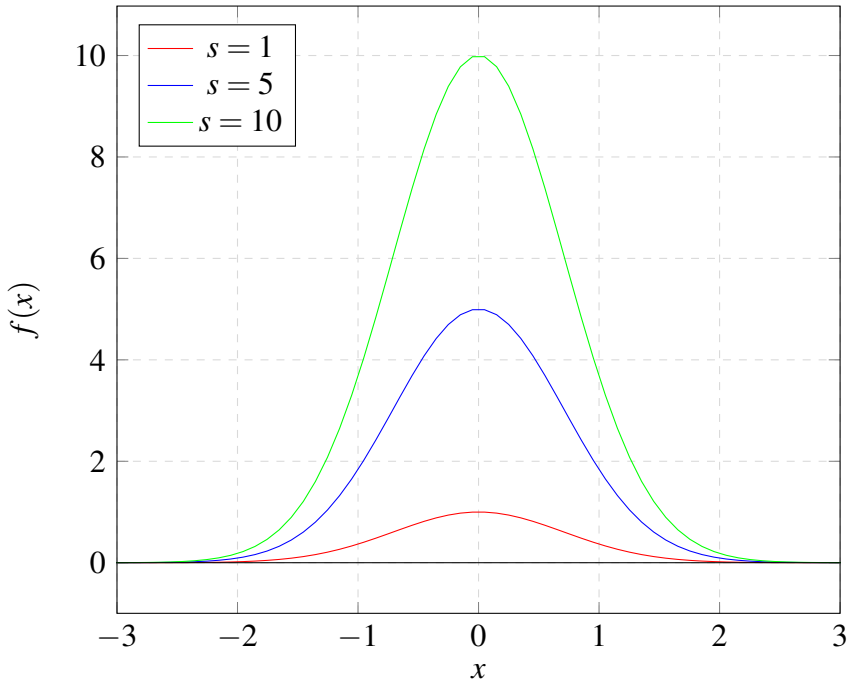

Figure 2.3: Plot of the function $f(x) = s \cdot exp(-\frac{x^2}{\sigma^2})$ for different values of s.

Effect on Gradient

In section 2.1.2, we showed how the new loss function is composed. The normal cross-entropy-loss and the distance function are added together. Therefore, taking the derivative of equation 2.7 results in

$$\nabla_\theta L = \nabla_\theta(\sum_i \delta_{yi} log(f(x)_i) + distance(\theta, \theta_c)) \tag{2.10}$$

$$= \nabla_\theta(\sum_i \delta_{yi} log(f(x)_i)) + \nabla_\theta distance(\theta, \theta_c) \tag{2.11}$$

The derivative of the sum decomposes in the sum of the derivatives. Therefore, the cross-entropy-loss will be added to the gradient the same as before. The new term is the distance function on the right.

First of all, it influences the direction of the gradient, so that the network parameters distance from their checkpoint values. If we take one parameter θ_i and compute the derivative of the distance function with respect to the checkpoint θ_c, we end up with:

$$\nabla_{\theta_i} distance(\theta_c, \theta_i) = \nabla_{\theta_i} exp(-\frac{\sum_j (\theta_{c_j} - \theta_j)^2}{\sigma^2}) \tag{2.12}$$

$$= \nabla_{\theta_i}(-\frac{\sum_j (\theta_{c_j} - \theta_j)^2}{\sigma^2}) \cdot exp(-\frac{\sum_j (\theta_{c_j} - \theta_j)^2}{\sigma^2}) \tag{2.13}$$

$$= -\frac{1}{\sigma^2} \cdot \nabla_{\theta_i} \sum_j (\theta_{c_j} - \theta_j)^2 \cdot exp(-\frac{\sum_j (\theta_{c_j} - \theta_j)^2}{\sigma^2}) \tag{2.14}$$

$$= -\frac{1}{\sigma^2} \cdot 2(\theta_{c_i} - \theta_i) \cdot \nabla_{\theta_i}(\theta_{c_i} - \theta_i) \cdot exp(-\frac{\sum_j (\theta_{c_j} - \theta_j)^2}{\sigma^2}) \tag{2.15}$$

$$= -\frac{1}{\sigma^2} \cdot 2(\theta_{c_i} - \theta_i) \cdot (-1) \cdot exp(-\frac{\sum_j (\theta_{c_j} - \theta_j)^2}{\sigma^2}) \tag{2.16}$$

$$= \frac{2}{\sigma^2} \cdot (\theta_{c_i} - \theta_i) \cdot exp(-\frac{\sum_j (\theta_{c_j} - \theta_j)^2}{\sigma^2}) \tag{2.17}$$

Only the factor $(\theta_{c_i} - \theta_i)$ will affect the direction of the gradient, the other terms will be $>= 0$ for $\forall \theta_i, \theta_{c_i} \in \mathbb{R}$ and only scale the size. Therefore, they can be disregarded and there are three cases:

- $\theta_{c_i} > \theta_i$
  It follows that $\nabla_{\theta_i} distance(\theta_{c_i}, \theta_i)$ will be positive. As we apply the negative gradient to update the parameters, θ_i will get smaller and $(\theta_{c_i} - \theta_i)^2$ will be increase.

- $\theta_{c_i} < \theta_i$
  $\nabla_{\theta_i} distance(\theta_{c_i}, \theta_i)$ will be negative and the distance $(\theta_{c_i} - \theta_i)^2$ will increase the same as above.

- $\theta_{c_i} = \theta_i$
  $\nabla_{\theta_i} distance(\theta_{c_i}, \theta_i)$ will be 0. Therefore, the parameter value θ_i will stay the same.

For $\theta_{c_i} \neq \theta_i$, the gradient of the distance function will successfully increase the distance. For $\theta_{c_i} = \theta_i$ however, it will introduce no gradient at all. This would problematic, as when a checkpoint is created, before the first update this case is true. However, the update is also composed of the cross-entropy-loss, which will likely introduce an non-zero gradient. In the next

iteration, $\theta_{c_i} \neq \theta_i$ will then be true and the distance function works as expected. If Momentum is used, there is another term that prevents the gradient from becoming 0.

Besides the influence of the distance function on the direction of the gradient, we expect an influence size of the gradient. Assume that the direction of the gradient is stable across multiple iterations, which is likely if we use Momentum. After we create a checkpoint, the normal loss function would therefore increase the distance to the checkpoint by itself. As we have seen above, the gradient of the distance function will also increase the distance, therefore both gradients will point in the same direction. For a stable gradient direction of the normal loss function, the distance function therefore increases the gradient size rather than change the gradient orientation. Only when the direction for a parameter changes, the distance term will work against it.

Computational analysis

Although the network performance is one of the most important aspects of a neural network, other factors like training time have to be taken into account. Here, we will analyze how the distance term affects training time. Algorithm 4 shows that at the beginning, the network is trained as usual. Therefore, there is no additional training time in this stage. After we add the checkpoint however, the distance function is added to the loss. In section 2.1.2 we have seen that there is no interaction between the gradient of the cross-entropy-loss and the distance function. The same is true for the foward pass. Therefore, the only added computation time raises from the distance term itself.

The number of parameters of the network stays constant over the epochs. Therefore, the number of mathematical operations in the foward and backward pass of the distance term stays constant. Lets assume the comparison time for two numbers is also constant regardless of their size. Then it follows, that the distance function creates a constant additional amount of time, denotet by d. In addition, let l denote the constant cost of one pass trough the regular training loop. If we pass this loop n times, our inital cost is $O(l \cdot n) = O(n)$. If we add our distance function, we have to add $O(n \cdot d)$ to our cost $O(n \cdot l + n \cdot d)$. As $O(n \cdot l + n \cdot d) = O(n \cdot (l + d)) = O(n)$, the complexity of the algorithm stays the same. However, the distance function will still raise the per epoch cost, although by a constant. If we add more checkpoints, there are again no interactions with other checkpoints or the regular training loop. Therefore, every new checkpoint should add the same additional cost, while keeping the complexity class the same. The actual value of the additional cost will be discussed in the results, section 3.6.

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.
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.

Algorithm 4 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
```

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.

Algorithm 5 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
```

Loss function

Finally, we add our distance term and the regular loss function together. First, the standard loss is computed. Then, we iterate over all checkpoints and add their resulting distance values to the loss.

Algorithm 6 Loss function

```
def loss_distance(model, images, labels, minimum_list):
        loss = 0
        classify_loss = loss_function(model, images, labels)
        loss += classify_loss
        for minimum in minimum_list:
            loss += (distance_factor/len(minimum_list)
             *kernel_function(computedistance(minimum, model), distance_width))

        return loss
```

2.2 Configuration

2.2.1 Library and Training

The code for the network was written in Python 3.7.4 , with the use of Pytorch [18] as the machine learning library. Data that occured during training was logged and plotted with Tensorboard. Training was done on the TCML Cluster of the University of Tübingen [1].

2.2.2 Dataset

CIFAR-10 [16] is a popular dataset for image classification. It 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. [15].

The data is tranformed by using a random crop, random horizontal flip and normalization.

2.2.3 Networks

ResNet

The Residual Network (ResNet) architecture was first proposed by He et al. [10]. 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.

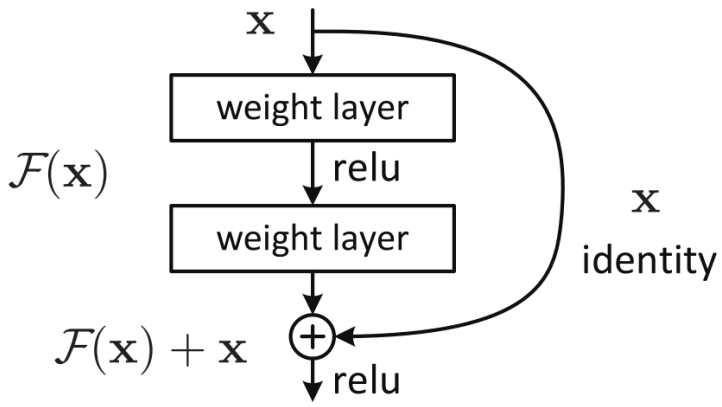

Figure 2.4: Residual Block from [10, Page 2]

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.

MobileNetV2

MobileNet was first introduced by Howard et. al [12] as a lightweight neural network for the use on mobile devices. To reduce the computation effort of the network, they 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 the shape 32x32x1. To get more channels, we would need more kernels. For a total of k output channels, we would need to perform $32 \cdot 32 \cdot k = 1024 \cdot k$ convolutions, where each convolution with the Kernel has $3 \cdot 3 \cdot 3 = 27$ multiplications, resulting in a total of $27648 \cdot k$ 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 $32 \cdot 32 \cdot 3 = 3072$ convolutions of a $3 \cdot 3 = 9$ Kernel, resulting in 27648 multiplications. In the second step, we need $32 \cdot 32 \cdot k = 1024 \cdot k$ convolutions of a Kernel with $1 \cdot 1 \cdot 3 = 3$ multiplications resulting in $3027 \cdot k$ convolutions. This results in a total of $27648 + 3027 \cdot k$ convolutions. Although this might be more for $k = 1$, it scales up way smaller for larger k, because of the number of channels only be dependend on the lightweight pointwise convolution. For example for $k = 10$, depthwise seperable only needs 57918 multiplications while a traditional convolution needs 276480. Figure 2.2.3 shows a graphical illustration of this idea.

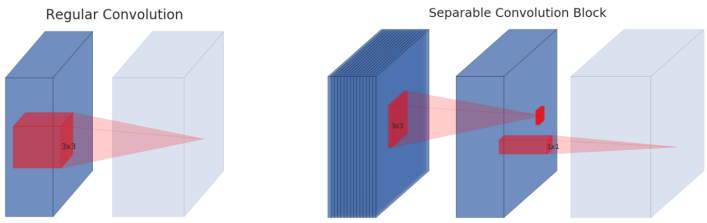

Figure 2.5: Depthwise Separable Convolution from [19, Page 3]
On the left, a regular convolution with a 3x3xDepth kernel is drwan. On the right, this convolution is separated in depthwise convolution with a 3x3x1 Kernel, followed by a pointwise convolution with a 1x1xDepth Kernel. This methods saves computation time over the standard method, while also allowing interactions between channels.

With the second iteration called MobileNetV2 [19], 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.

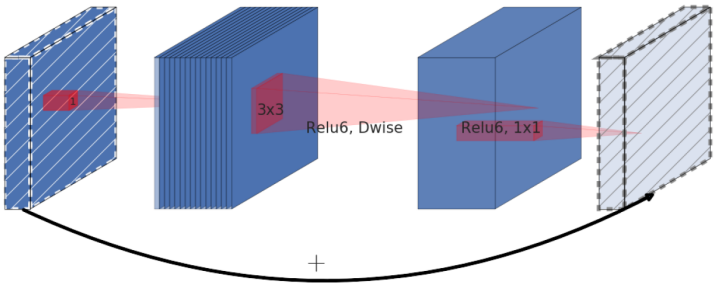

Figure 2.6: Inverted Residual Block from [19, Page 3]
First, a Relu 6 followed by a Pointwise Convolution is applied to map the data to a higher dimensional space. Then a depthwise convolution is applied, followed again by a Relu6. Finally, the data is compressed back to the original size. This time, 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
- Batch size
  128
- Regularization
  An L_2 Regularization as described in chapter 1.2.1 is used with a factor of 1e-3.
- Loss Function
  Cross-Entropy Loss
- 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 600 is used.

3 Results

After the theoretical introduction in chapter 2, this chapter provides some empirical results. In section 3.1, a baseline for further comparisons will be created. Section 3.2 will focus on the effects of the hyperparameters of the distance function. How multiple checkpoints can be used will section 3.3show, followed by the combination with learning rate schedulers 3.4. We will investigate the effects on ensemble methods in section 3.5 and finally study the computational cost in 3.6.

3.1 Baseline

For the baseline, we use the hyperparameters as defined in section 2.2.4. The distance function is used with a width of $\sigma^2 = 1000$ and a strength of $s = 1$. First we train both networks without distance function. After 100 epochs, we add a checkpoint in the case of the network with distance function.

For the validation accuray, we can see no impact of the distance function. Both networks show a standard learning curve, with a initial strong increase in validation accuracy until coming into convergence in later epochs. This results in a validation accuracy of around 90% for MobileNetV2 and 89% for ResNet32 after 600 epochs.

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 gradient of the RBF Kernel. But as the distance increases, its gradient becomes smaller, similar to the gradient of the Kernel. The blue plot follows a smilar curve, but with a much smaller gradient and overall distance. Therefore, the additional term succeeds in the initial goal, which was to distance from the checkpoint. Note that despite the validation accuracy beeing in an area of convergence, SGD even without distance function keeps on walking and never truly stops at a point.

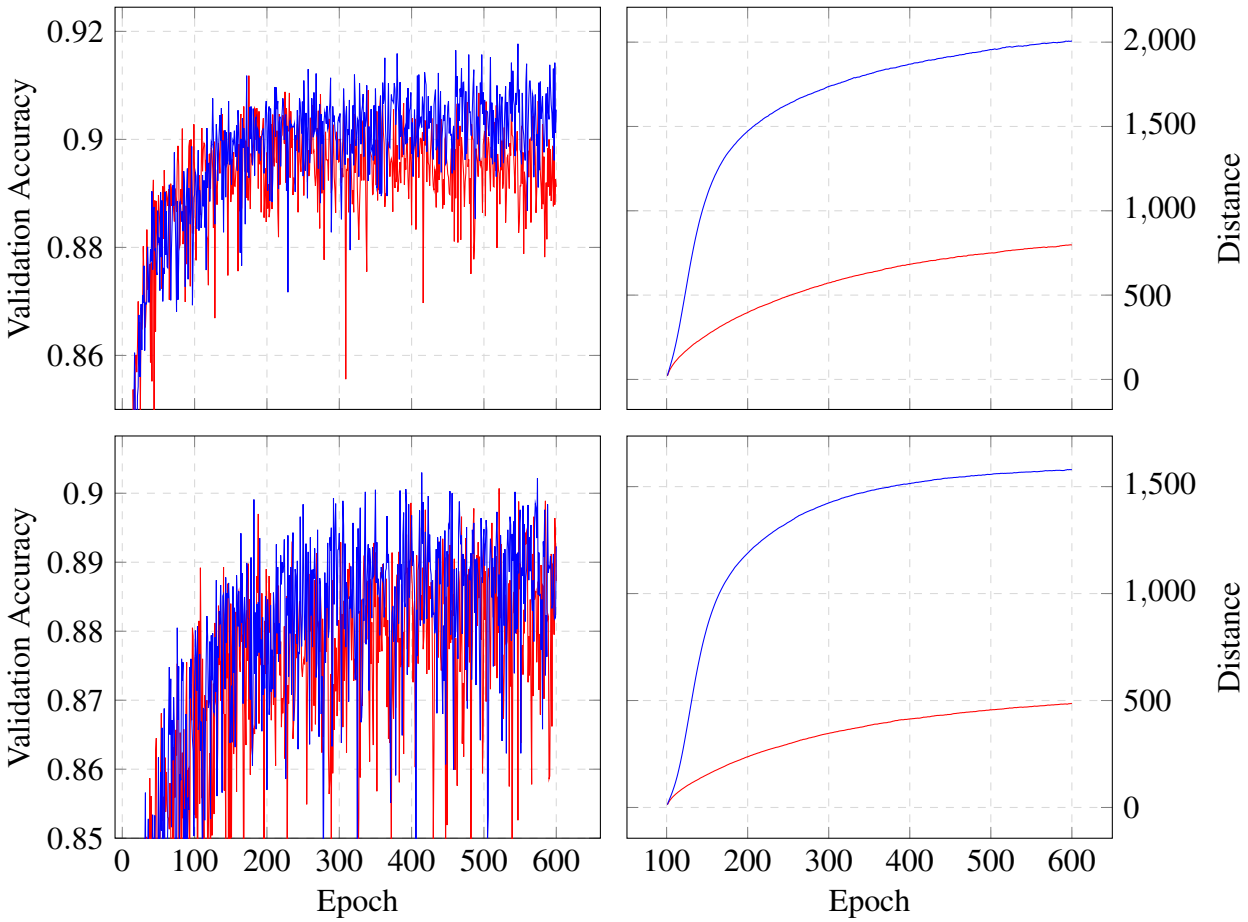

Figure 3.1: Validation accuray and distance to the checkpoint for MobileNetV2 (upper) and ResNet32 (lower) trained without distance function (red) and with distance function (blue).

3.2 Distance function Hyperparameters

3.2.1 strength

Recall section 2.9, where we added strength as a hyperparameter, controlling the influence of the distance function. As the RBF Kernel is bound between 0 and 1, the strength hyperparameter will increase that bound between 0 and the strength value. In this section, we use the same hyperparameters as in section 3.1 but vary the strength parameter. We also add the checkpoint after 100 epochs.

For higher strength values, the validation accuracy receives an initial drop after a checkpoint is added, which is larger the higher the strength parameter value, see figure 3.2.1. This may be due to the increased influence of the strength function. Each distance between the parameters and the checkpoint starts at 0, therefore the values of the exponential function starts at $strength \cdot 1$. A higher strength value will lead to a higher influence of the distanc term on the loss function, and consequently on the gradient. The gradient of the cross-entropy loss will become insignificant, and the optimizer will update the weights without regards to the validation accuracy, hence the drop at the beginning.

The distance plot reflects this behaviour, as we can see an larger increase in distance for larger strength values. After the initial step however, the distance quickly converges. This is due to the fact that after the initial strong decrease, the value of the distance function quickly comes close to 0. Therefore, the distance Kernel becomes insignificant again, and the Cross-Entropy loss is followed. The validation accuracy reflects this behaviour, as after the initial drop the accuracy recovers to the level of before. This provides additional insight in the loss landscape. As the network distances from it‘s current position on the loss landscpae, but is still able to reach high accuracy, there have to be areas of high accuracy everywhere on the landscape. This is in consonance to other research in this field, as discussed in section 1.2.2.

As a result, the value of the strength doesn't matter in long term at least. It merely increases the distance to the checkpoint by defining how long the distance term is important to the loss function. But after that, the normal loss is followed again. As we above suggested, areas of low loss and high accuray can be found everywhere. Therefore, the same accuracy as before can be reached, no matter the value of the strength. On the other hand, it is also unlikely that it will outperform the last best value, as new areas are not generally better than other. That's why the effect of the distance doesn't transfer to the validation accuracy.

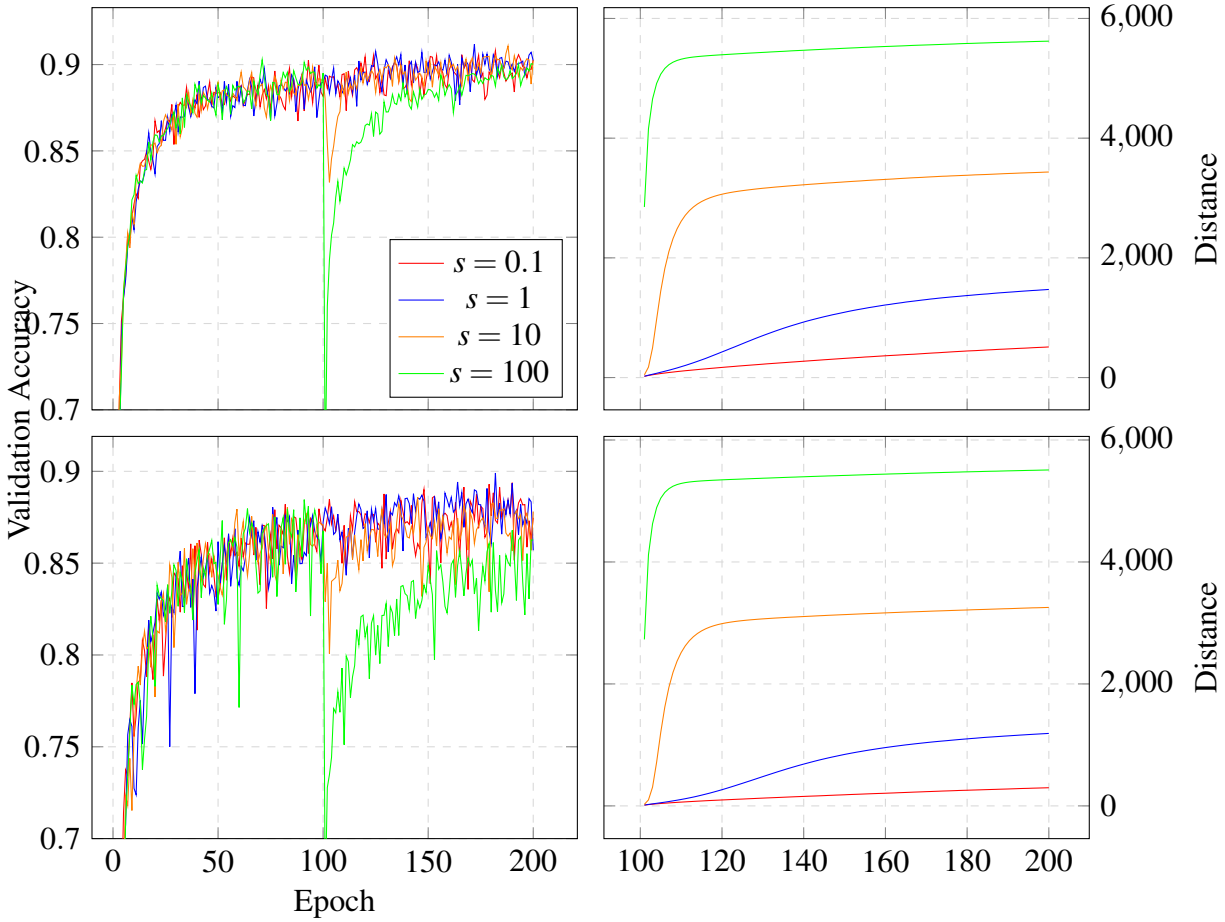

Figure 3.2: Validation accuracy and Distance for MobileNetV2 (upper) and ResNet32 (lower) trained without distance function and different strength values.

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 further from it's checkpoint than for smaller values. That's exactly what can be seen in figure 3.2.2: For an σ^2 of 0.1, the distance follows the one of network trained without distance function, due to the distance function quickly becoming 0 and therefore having no influence. The larger the width, the slower the values of the distance function decrease for further distance. Therefore, the gradient of the distance function will stay quite large, pushing the network further away. This can be seen for higher σ values, where the distance increases further.

Initially however, the distance plot doesn't reflect this behaviour. The curve for $\sigma^2 = 10$ has a higher distance value than for $\sigma^2 = 100$. That's because the gradient of the distance function is higher for small distances, the smaller σ^2. But for larger distances, this effect flips with the gradient of the larger width staying higher for more epochs. Therefore, the curves intersect after additional epochs. The smaller width converges, while the larger stays constant until eventually reaching it's area of convergence. The intersection happens later the larger the width, for $\sigma = 10000$, only after 307 epochs.

This behaviour however has no influence on the validation accuracy, which stays the same for every width, in contrast to the strength. This may be due to the fact, that a larger width does in fact decreases the size of the gradient. A smaller gradient over more epochs will be added to the gradient of the Cross-Entropy Loss. The optimizer therefore nearly keeps on following the normal gradient and descending the valley of low loss, with a small but steady push to increase the distance to the checkpoint. Therefore, even if the network is allowed to move quite freely on the loss surface, it finds areas of low loss and high accuracy in distant areas of the checkpoint, meaning there has to be good local minima everywhere on the landscape.

In particular, $\sigma^2 = 1000$ seems to increase the distance far enough in a small number of epochs. That's why this configuration is used as standard in the following work.

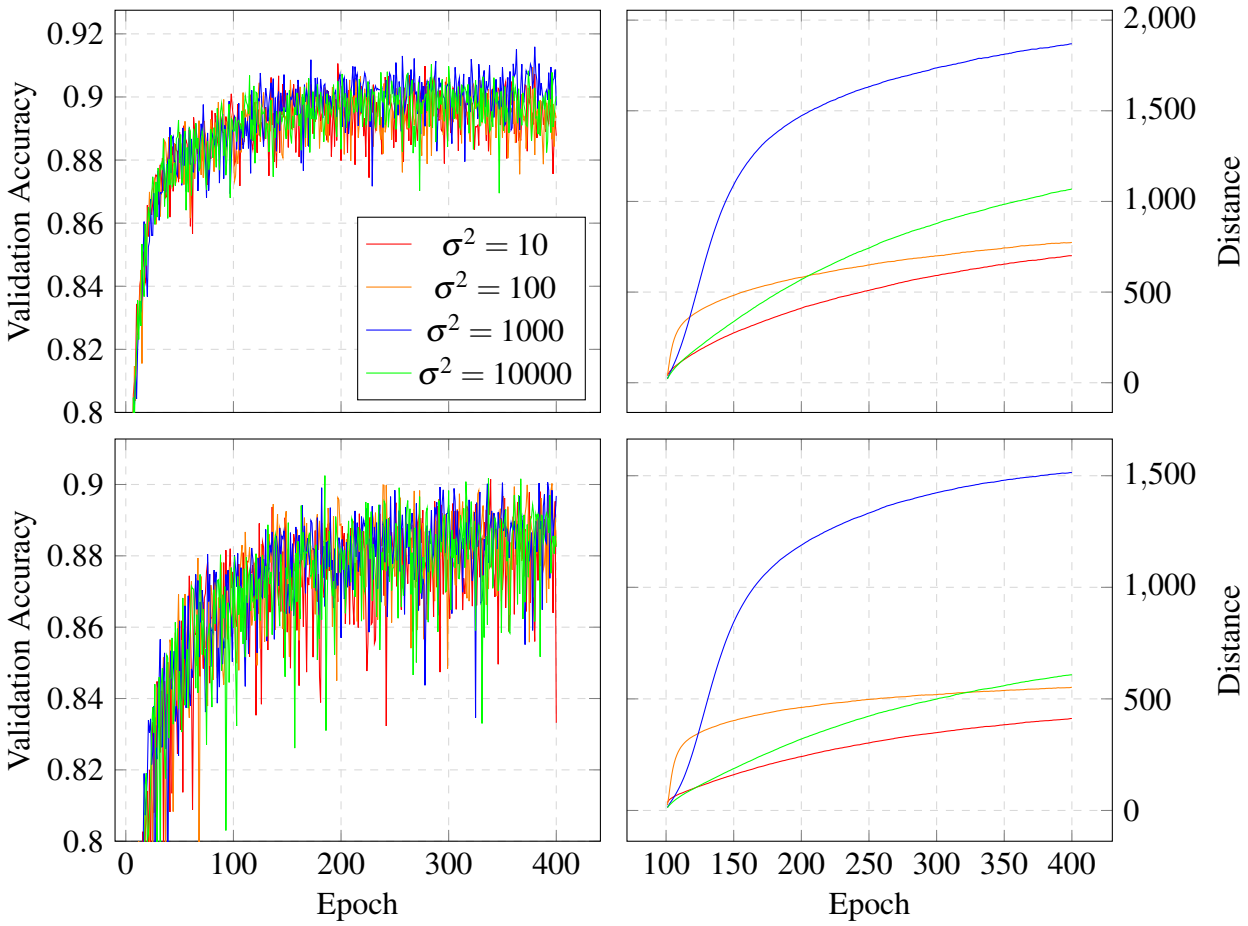

Figure 3.3: Validation accuray and Distance for MobileNetV2 (upper) and ResNet32 (lower) trained with different widths of the distance function.

3.3 Multiple Checkpoints

In chapter 2, we have seen that we can not only incorporate one but also multiple checkpoints in the loss function, on which this section will focus. Here, we add a checkpoint after every 150 epochs that are performed, resulting in a total of 3 checkpoints after 600 epochs. We also perform the same analysis as in section 3.2.1, where we alternate the strength factor.

For the validation accuracy, we can see the same pattern as in section 3.2. For low strength, we experience no effect at all. If we increase the strength there is an initial drop in the accuracy. For MobileNetV2, the accuracy recovers and even tops the baseline, with a small but increasing margin for every new checkpoints that is added, coming to an improvement of 0.8% over the baseline after 600 epochs. In contrast, ResNet32 doesn't seem to benefit from multiple checkpoints, the accuracy even decreases for larger strength values.

The pattern of MobileNetV2 is quite similar to warm restarts, where for every restart the reached accuracy increases. For a in depth comparison, see section 3.4.1.

The influence on the distance to the checkpoints is more interesting however. In general, an additional checkpoint seems to decrease the distance to the others. This is quite counterintuitive at first glance. Suppose for one parameter, we have checkpoint value a and create a new one

at our current value $b < a$. To increase the distance to both, we would just have to walk in direction $c < b$. As we have seen in previous section, this would be sufficient as areas of low error exist everywhere. But why is the distance then decreased for our first checkpoint? The reason might be the L_2 regularization, as it restricts the weights to small sizes. At some point, it might be less costly to decrease the weights again for a smaller L_2, traded off against a bit smaller distance to other checkpoints. This restriction in weight space is probably why new terms lead to a decrease of the distance to existing checkpoints. A network trained without L_2 regularization futher supports this explanation, see figure 3.3. Here, the distance increases further for each new checkpoint that is added.

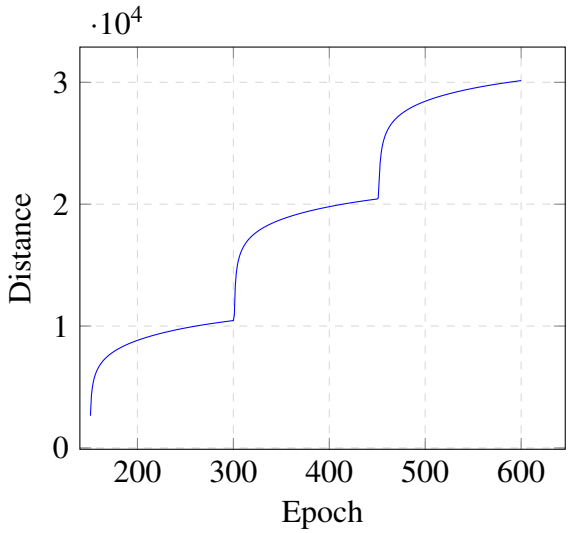

Figure 3.4: Distance to the first checkpoint after 150 epochs for MobileNetV2 trained with multiple checkpoints and $s = 100$, but without L_2 regularization.

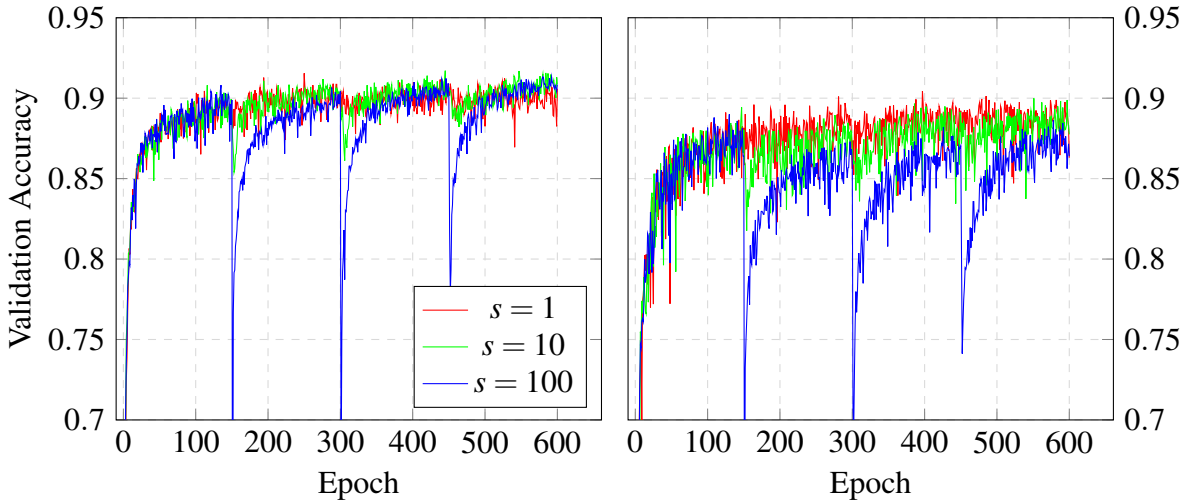

Figure 3.5: Validation accuray and Distance for MobileNetV2 (left) and ResNet32 (right) trained with multiple checkpoints.

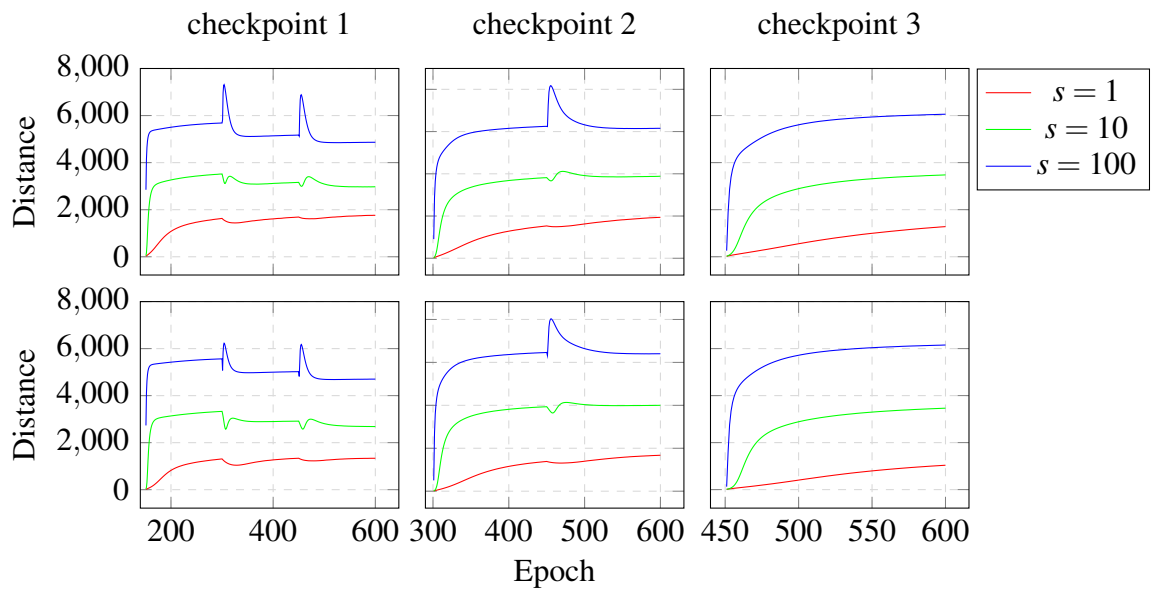

Figure 3.6: Distance plots for MobileNetV2 (upper) and ResNet32 (lower) trained with multiple checkpoints and different strength values.

In short term however, new checkpoint produce quite different influence on the distance to existing ones. For a strength of $s = 1$ we can see a slight decrease which turns into a longer increase until the next checkpoint. For $s = 10$, we see an aprupt decrease with an small peak after. For $s = 100$, there only is a peak at the beginning which quickly diminishes. Noticeable, this pattern repeats for the other checkpoints and therefore seems unlikely to be a coincidence. The size of these hills can be tracked to the size of the strength. Larger strength introduces a larger gradient and therefore step size at the beginning, making the hills larger. This doesn't explain the different orientation of these hills. For the case of $s = 100$, the initial increase of distance seems reasonable. As the value of the old distance function is not 0 when a new checkpoints is created, the current gradient should face in direction away from the last checkpoint. A new checkpoint term in the loss function introduces a gradient boost, as section 2.1.2 discussed. Because momentum preserves the old gradient, it will acclerate in direction away from the first and the second checkpoint, therefore leading to a hill. However by this explanation, the pattern should repeat for the other strength values, which is not the case. Therefore, the true cause of the difference in short term effects remains unclear.

3.4 Learning rate

3.4.1 scheduler

Learning rate Scheduler were introduced in section 1.1.5, where the learning rate was not held constant, but decayed over time. One further addition were warm restarts, where after the decay we set the learning rate back up to the inital and repeat this cycle several times. We use a step decay scheduler with $\gamma = 0.1$ for every 50 epochs, and a warm restart after 150 epochs. For cosine decay with warm restart, we set $\varepsilon_{min} = 0$, $\varepsilon_{max} = 0.01$ and $T_0 = 150$.

For the validation accuracy, schedulers outperform a fixed learning rate as we can see in figure 3.4.1. At the beginning of every cycle, the performance drops due to high learning rate.

But when the learning rate decays again, the accuracy recovers. Furthermore, the maximum accuracy of every cycle slightly increases, until eventually coming to convergence. Here, a cosine decay outperforms a step decay. That's probably due to the smaller learning rate at the end fo each cycle cosine decay can reach. This leads to a maximum accuracy of around 95% for MobileNetV2 and 93% for ResNet32.

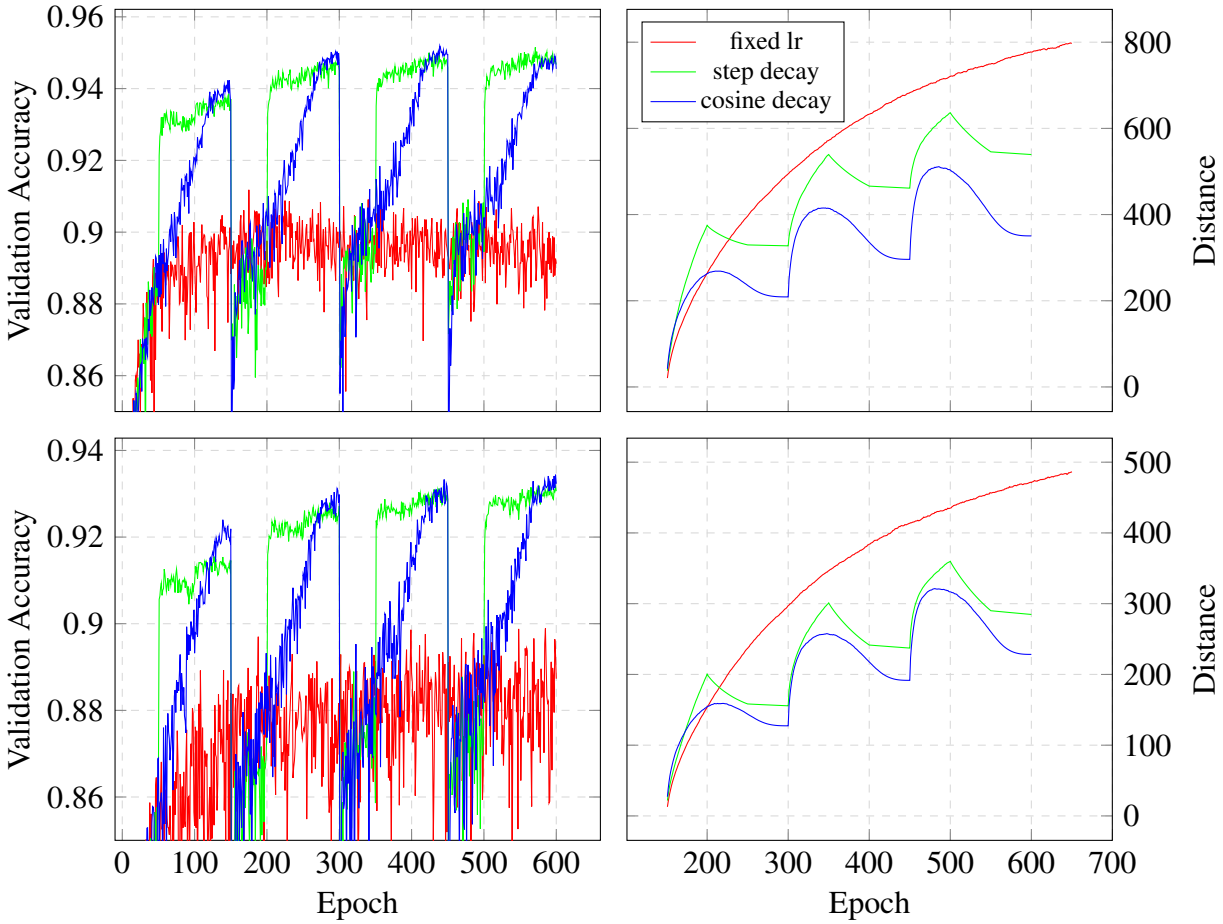

Figure 3.7: Validation accuray and Distance to a checkpoint after 100 epochs for MobileNetV2 (upper) and ResNet32 (lower) trained with learning rate schedulers but without distance function.

The distance plot now also looks a bit different compared to a fixed lr. We can see that after an initial increase in distance to the checkpoint, the distance reaches a local maximum and then actually decreases again until coming into an area of convergence. For each restart performed, the same pattern repeats again. For step decay, we can see hard edges in contrast to smooth curves of cosine decay. The similarity to the alteration of the learning rate gives evidence that the decrease of distance is caused by the learning rate.

One possible explanation is that the decrease in learnig rate could lead to smaller increase in distance by the pure fact, that smaller learning rate leads to smaller updates of the weights and threfeore smaller increase in distance. However, this couldn‘t explain why the distance itself actually keeps decreasing, not just smaller increasing. Only could try to explain this with the L_2 Loss and a similar argument to section 3.3. For high learning rate, a larger Momentum builds up. This leads to parameter values, which are very larger. After the momentum is slowed

down, the L_2 regularization keeps decreasing the weights again until reaching an equilibrium. But larger convergences in the following cycles make this explanation doubtful.

We have seen that whenever a warm restart is performed, the accuracy drops. However with decreasing learning rate, the network recovers and even tops the performance of the previous cycle. This pattern is quite similar to effect of multiple checkpoint of section 3.3. Section 2.1.2 further motivates a comparison: We suggested that a checkpoint would initially just increase the size of the gradient, rather than changing its orientation, given that the orientation is stable over some epochs. Therefore, the same effect on the parameter update, but rather from a larger gradient itself than a larger learning rate, should be achieved.

Figure 3.4.1 shows a comparison of a cosine decay with warm restart for a network trained with distance function, multiple checkpoints and $s = 100$ from section 3.3. As mentioned, the pattern is quite similar, with a drop in validation accuracy after the warm restart or checkpoint. The top accuracy in each cycle also tops the last one for both cases. The effect for MobileNetV2 is with around 1% against 0.8% increase in maximum accuracy between the first and last cycle for cosine decay stronger than for multiple checkpoints. Additionallly, the network with cosine decay gets a much better absolute accuracy. That's probably due to the small learning rate at the end of each cycle, where the network can exploit a small region to fine adjust the parameter values.

In contrast to MobileNetV2, section 3.3 showed that ResNet32 didn't benefit from multiple checkpoints. If the larger update step is responsible for the boost in performance for each cycle, then ResNet32 should benefit the same as MobileNetV2. Because this is not the case, it remains unclear if either the explanation is wrong or if some properties of the ResNet32 architecture prevent the effect.

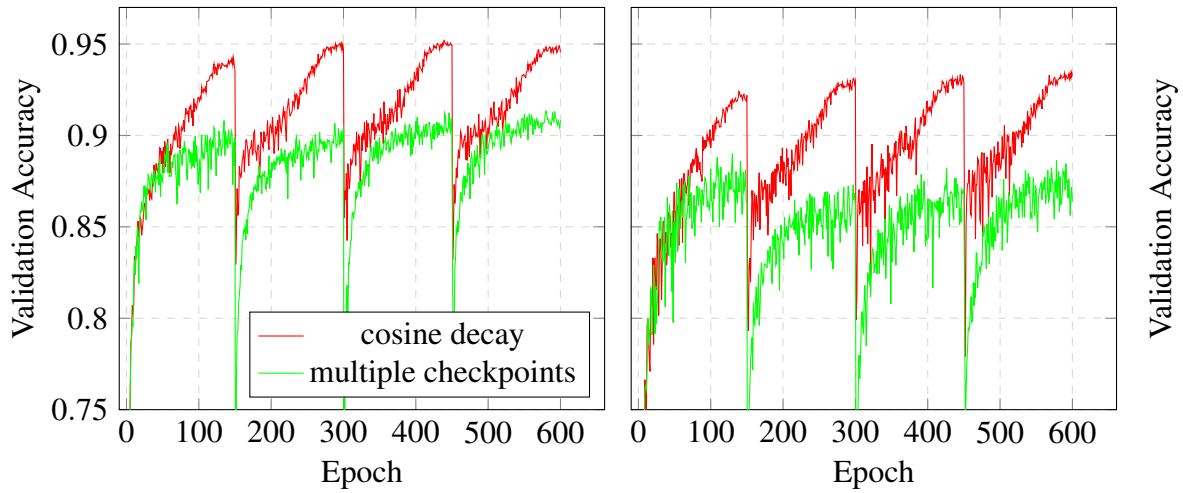

Figure 3.8: Validation accuracy of multiple checkpoints compared to cosine decay with warm restart for MobileNetV2(left) and ResNet32(right).

Would a combination of both techniques lead to an even better result? Here, we combined the cosine decay with multiple checkpoints at each warm restart. We can see that this decreases the top validation accuracy of each cycle. Furthermore, the distance behaves differently when compared to cosine decay, at least in the first cycle. Where normal cosine decay decreases the distance again for small learning rates, the distance function prevents this from happening. Instead, it just normally converges. In later cycles however, the distance kernel becomes insignificant again and the patterns tend to look more similar.

If we compare it to multiple checkpoints of section 3.3, the distance plot resembles more the shape of the cosine decay. Where for a fixed learning rate and multiple checkpoints, each new checkpoint led to an initial decrease and then increase in distance, cosine decay and distance function actually turn this effect around. Here, an inital increase is followed by a decrease. As stated above, this also happens without distance function. Therefore it seems, that the learning rate has a potentially stronger influence on the shape of the distance to one checkpoint than other checkpoints of the distance function itself.

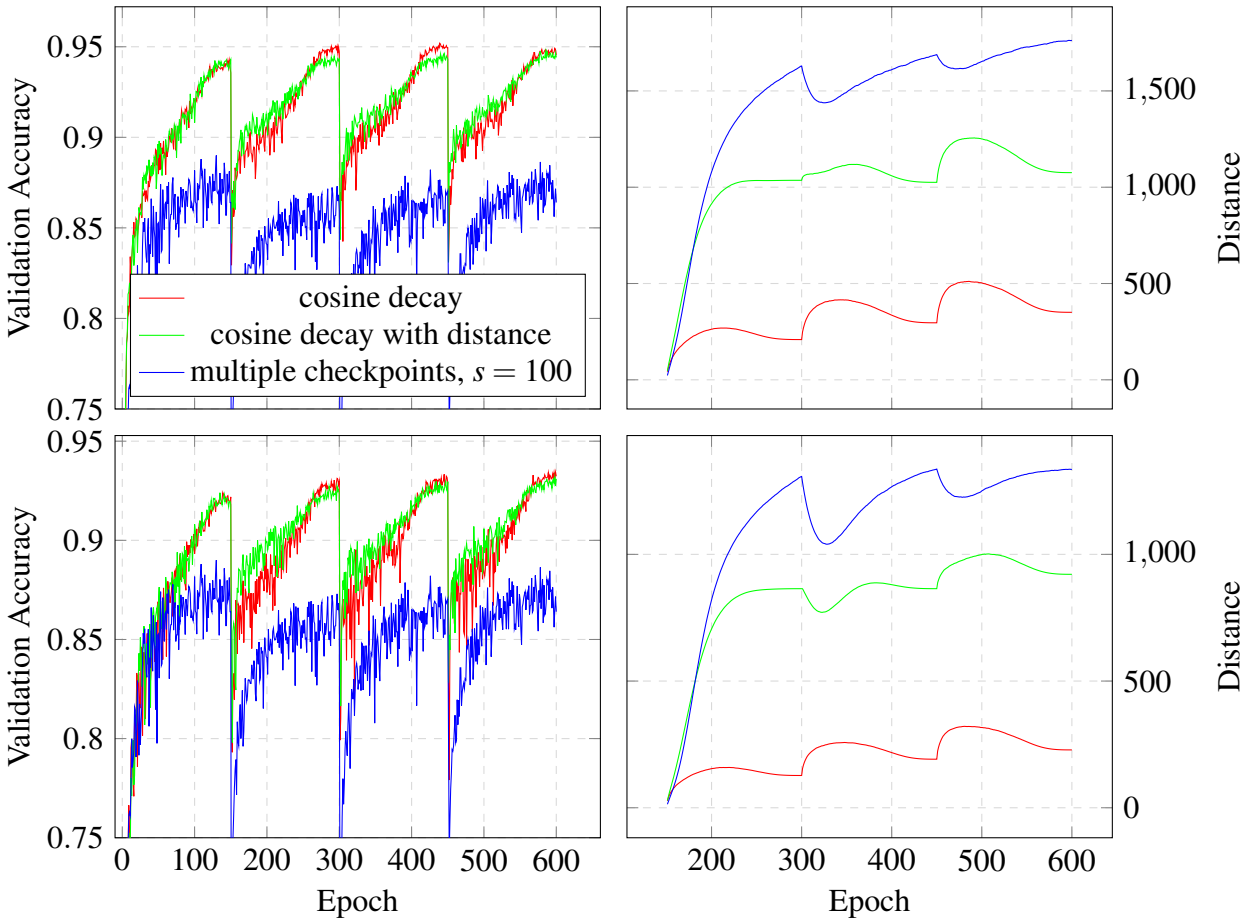

Figure 3.9: Validation accuray and Distance to a checkpoint after 150 epochs for MobileNetV2 (upper) and ResNet32 (lower) trained with learning rate schedulers and without (red) and with (green) distance function with multiple checkpoints.

3.4.2 wrong lr

We might suspect that a learning rate decay should be more robust to a suboptimal inital learning rate. At first, this seems to be case. Where MobileNetV2 without decay only reaches a accuracy of 60% and then decreases, a scheduler reaches a comparable accuracy to the best network. But if we perform a warm restart, the maximum accuracy drops for each cycle, so the network gets worse with increasing cycles. If we add the distance function however, we can see that it stabilizes the training so that the maximum accuracy stays the same for each cycle. This leads to a performance difference of 8% for MobileNetV2 in favour of the distance

function after 600 epochs. The difference is also present from the beginning of each restart and remains stable. However, we loose the effect from above, that each warm restart boosts the performance. Nevertheless, this is a significant improvement, which is stable across both ResNet32 and MobileNetV2.

If we compare the distance to the network with optimal learning rate in figure 3.4.1, there also are some differences. Here, especially MobileNetV2 fails do distance from its checkpoint at all in contrast to the optimal learning rate. With the distance function, the distance again goes on to reduce significantly after the initia increase instead of further increasing. This leads to the conclusion, that the difference in performance might arise from the fact, that the network is stuck in an area of high loss. Therefore, the distance function helps distancing from this area into an area of better performance. However, this interpretation would contrast other findings from both our results and the paper, namely that areas of low loss exist everywhere. In addition, the question why this effect happens for a larger inital learning rate is still open.

For other learning rates, there is no benefit from the distance function. For a smaller learning rate, the network performs the same same as normal, even though a slower convergence at the beginning. For a learning rate of 1, both networks doesn't learn at all. Instead, the validation accuracy stays constant at around 10%.

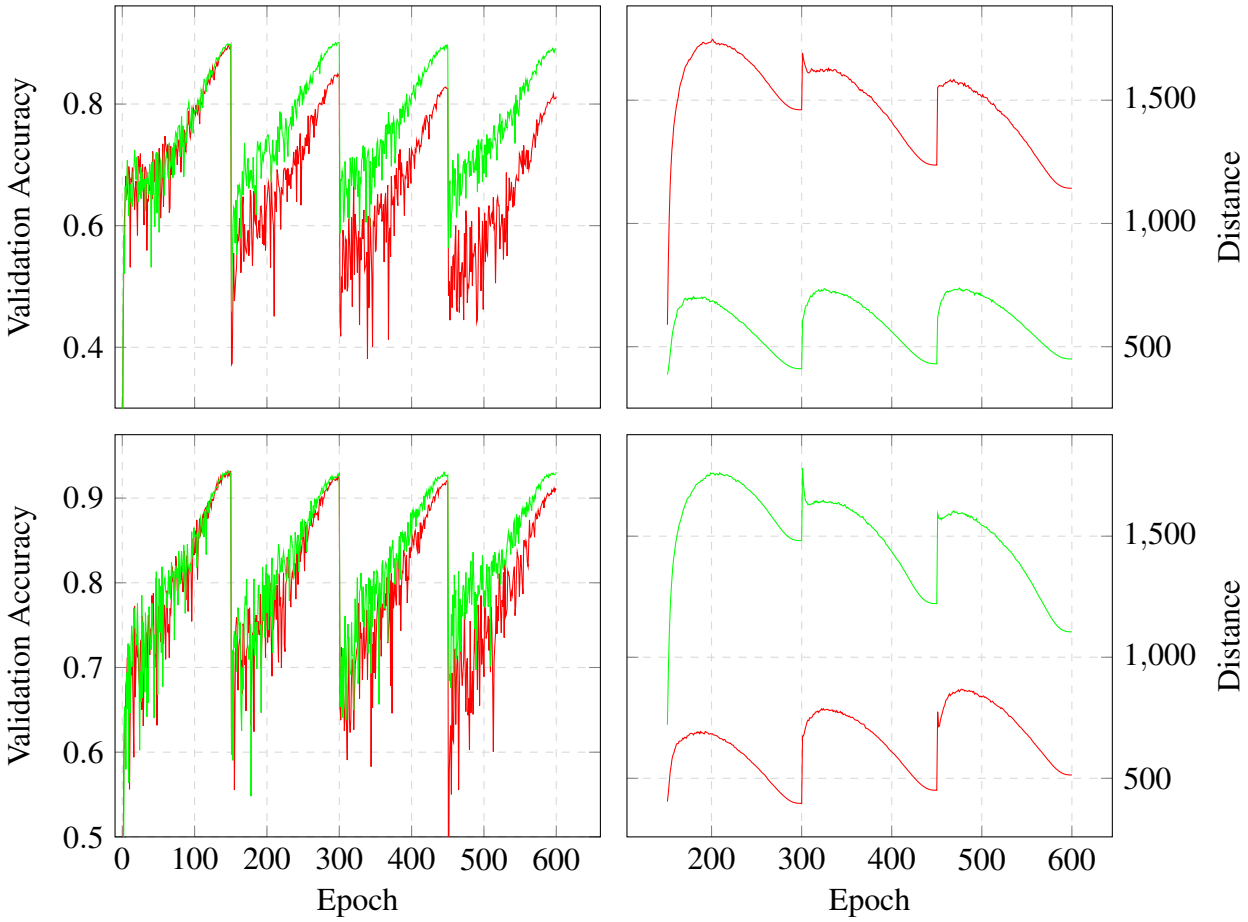

Figure 3.10: Validation accuray and Distance for MobileNetV2 (upper) and ResNet32 (lower) trained with learning rate schedulers and a learning rate of 0.1.

3.5 ensemble methods

Ensemble methods were introduced in section 1.2.4, the idea beeing that the combined prediction of multiple networks would result in better performance. However, the networks need uncorellated errors for that increasement.

In the approach from [17], they took a snapshot of the network at the end of each cycle. This procedure results in a performance boost of around 1% against a standard consine decay from section 3.4.1 without any additional training cost for MobileNetV2. The high learning rate after the warm restart ensures that the each new snapshot distances itself in the parameter space from the old. The hope is, that this distance results in uncorellated errors.

As motivated in section 2.1.1, we try to increase that distance more explicitly with our distance function. We use the same network as in section 3.3 and take new snapshots before adding a new checkpoint. After taking the snapshot, we add it to the ensemble, which consists of all previous snaphsots and the model that is currently trained on. For the ensemble prediction, we use the model averaging technique from 1.2.4.

For MobileNetV2, this leads to a ensemble accuracy of 94%, which is 4% better than the validation accuracy of the single network. Figure 3.5 shows a stepwise increase in ensemble accuracy after 150, 300, 450 and 600 epochs, when a snapshot is added to the ensemble. This shows that each new network adds performance to the ensemble. However, the network doesn´t reach the performance of cosine decay with warm restart and is even similar to a standard network without any decay or distance function. The difference is probably due to the fact, that cosine decay reaches a better performance for each snapshots of the ensemble. The predictions may therefore be better in overall, even if they are not more versatile.

To see if a ensemble would benefit from snapshots that have a further distance between each other and therefore have a lower corellation, we increase the strength parameter. However, the ensemble accuracy doesn't benefit from this. Instead, the accuracy gets even worse for $s = 10$ and $s = 100$.

In order to check if further distance truly means more different predictions, we compare the individual predictions of the networks. Table 3.5 shows the results. If we compare the different strength values of the distance function, we can see that the number of examples for which the predictions of the networks is the same has no difference between the strength values. This suggests that although the snapshots of the ensembles differ in distance to each other, this doens't result in different predictions. Even compared to the baseline, the coherence seems similar. One possible explanation could be the weight space symetry of section 1.1.4, where due to symetry in weights, networks can be distnant in weight space but still have the same predictions for all training examples. Another possible explanation could be the chracteristics of the training set, which may contain some easy and some difficult examples. As a fixed learning rate doesn't exploit the landscape enough, all networks may get the same easy examples right, regardless of their position on the loss landscape. But for hard training examples, all these networks fail.

baseline	cosine	$s = 10$	$s = 100$
7648	9130	7739	7788

Table 3.1: Number of test examples where all snapshots agree in their classification for different configurations of MobileNetV2.

For cosine decay, the coherence is even higher. This seems reasonable, as snaphsots which perform better necessarly need to have more similar predictions, as there are less options where

they are wrong and can therefore disagree. Nevertheless it is remarkable that although the distance between the networks for cosine decay and a fixed learning rate is quite similar, the prediction coherence differs strongly. This further suggests that a distance in parameter space doens't result in different behaviour. Cosine decay probably exploits the local landscape better due to a low learning rate and therefore arrives at more similar snapshots.

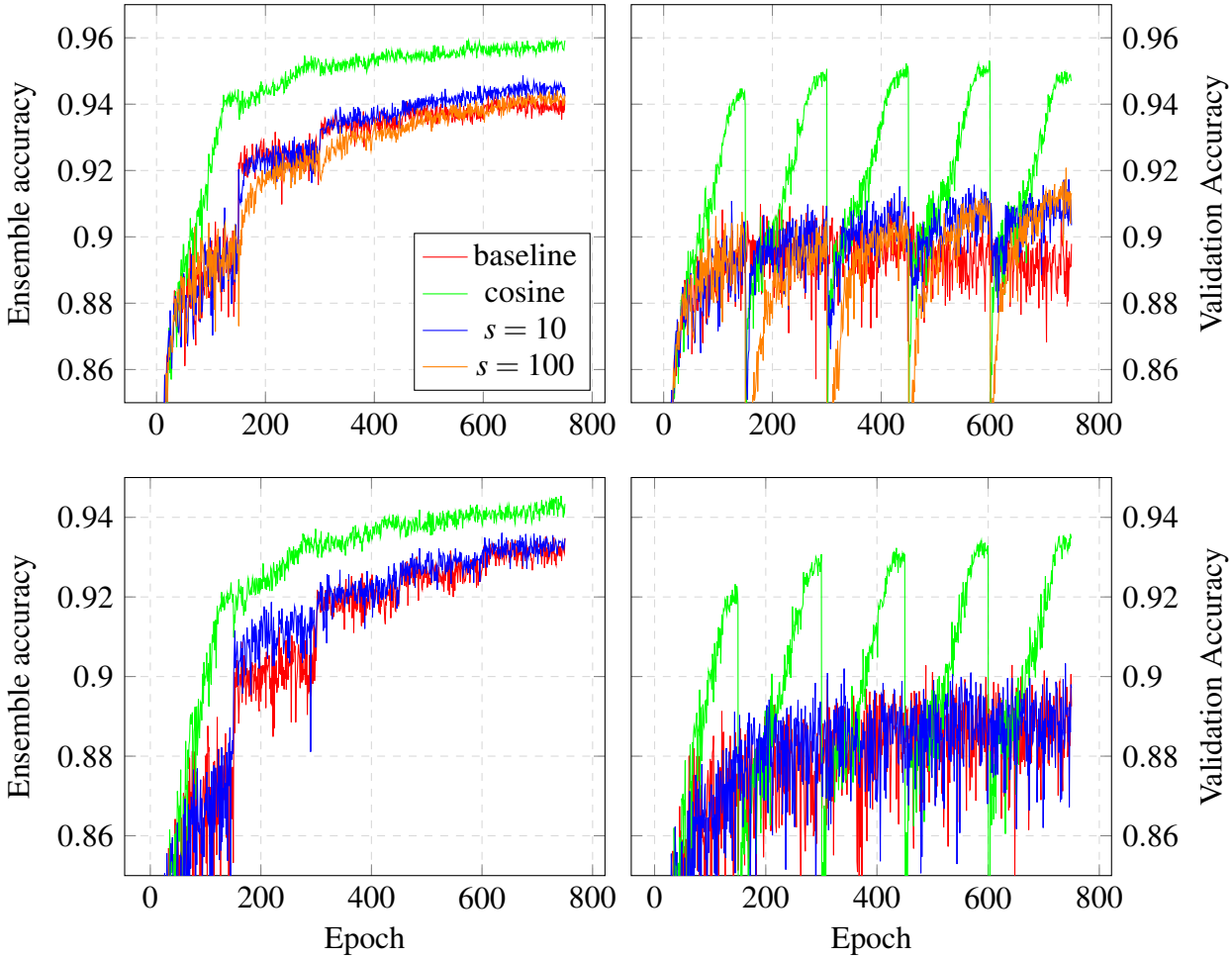

Figure 3.11: Ensemble accuracy of the snapshot ensemble (left) compared to the validation accuracy of the network where snapshots are taken from (right). [fehlt noch ein s=100 für ResNet32]

3.6 Computational Cost

Improvement in performance often arises with an increase in computational cost. In Section 2.1.2 this was discussed in a theoretical setting, suggesting that the additional cost would scale linear with each checkpoint. If we analyze the baseline network without distance from section 3.1, we reach a per epoch training time of 72s for MobileNetV2 and 55s for ResNet32, trained on the TCML Cluster. For the network trained with distance function, we start with the same epoch time, as the analysis expected. That's due to the fact, that we also start training without the distance function. But once we add a checkpoint to MobileNetV2, the epoch time rises by

around 22s to a total of 92s, see figure 3.6. If we add multiple checkpoints, the increase for each stays constant at around 21ms. For ResNet32, the amount that adds for each checkpoint is more variable. Here, the first adds 8s, the second 12s and the third 10s. Nevertheless, the hypothesis of the theretical computational cost seems to get confirmed. MobileNetV2 shows a linear increase for each checkpoint. Even if the results are more variable for ResNet32, it seems unlikely that the increase is superlinear, especially because the last checkpoints adds less time again. The instability is probably due to changes in hardware performance such as high temperature leading to a reduction in clock speed.

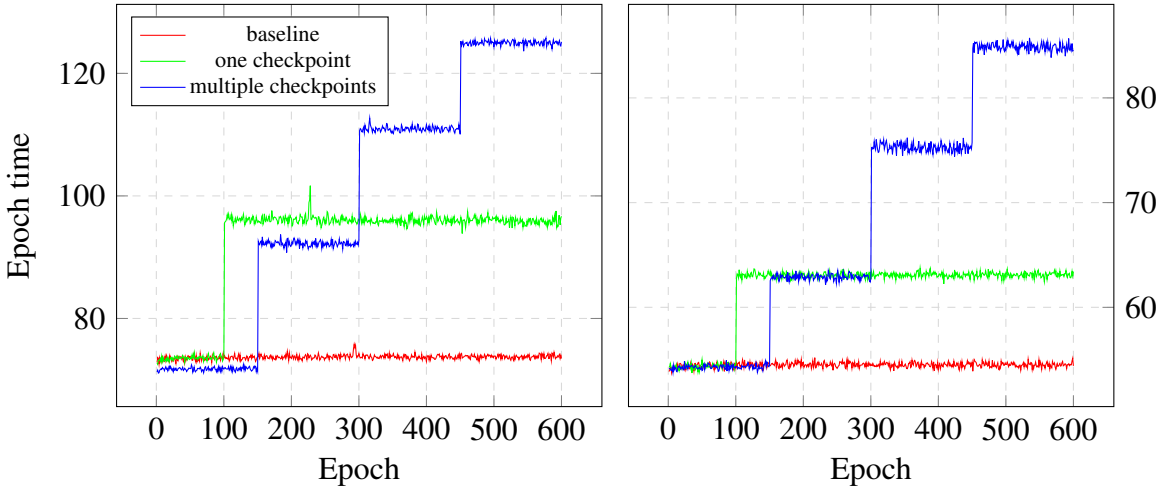

Figure 3.12: Epoch time in seconds for different configurations of MobileNetV2 (left) and Resnet32 (right).

4 Dicussion

In section 2.1.1, we summarized some of the challenges of optimization and described our approach to overcome some of these. The key idea was to push the network away from a checkpoint to escape local minima and explore the loss landscpae further. The following sections described how we tried to realize this goal: We expanded the loss function by a term which measures the distance between the checkpoint and current state and penalizes small distances.

The results showed that the distance function acted successfull at achieving this goal. Whenever we included the distance term, the network distanced further than without, due to the additional gradient. At the beginning, there was a larger increase in distance due to the large gradient of the RBF Kernel of the distance function. When the gradient becomes small enough in later epochs, the distance converges. In section 3.2 we showed, how this distance can be modified. A wider distance function lets the network distance further, but at a slower rate. This reflects the shape of the RBF Kernel, because a larger width factor lets the function become wider, but also have a smaller gradient, see figure 2.1.2. A larger gradient can be created by varying the strength factor. This results in the same effect as above, but at a much faste rate, because of the larger gradient. In summary, the distance behaves like we would predict from the shape of the distance function.

Another interesting fact was that even without a distance Kernel, SGD also increases the distance, even when the validation accuracy seems to converge. This gives evidence for another difference between neural networks and traditional optimization. Where in traditional optimization, the goal normally is to reach global optimum, optimization in neural networks fails to do so. Nevertheless, the good performance of optimizer like SGD shows, that it may not even be necesarry or useful to search for a global optimum. The continous motion of SGD also implies that traditional termination metrics like 0 gradients are not sufficient, as the size of the gradients doesn't decay over time. Therefore, other metrics like early stopping at a good validation accuracy have to be taken into account, as discussed in section 1.1.4.

Recent research on loss landscape suggests that there are cells of local minima which are connected by valleys of low loss. Between these cells however lie areas of high loss. To overcome these areas of high loss and jump to other cells, we used our distance function. Whenever we created a checkpoint, we saw a drop in validation accuracy. The higher the influence of the Kernel, the higher the drop was. Section 3.2 showed some reasons for this from a mathematical viewpoint. Whenever the influence of the distance function is higher, its gradient will also be larger compared to the gradient of the cross-entropy-loss. Therefore, the gradient of the cross-entropy-loss will be neglected, we don't follow the path of low loss any more and the accuracy drops, as a minima is surrounded by areas of high loss.

However, after further epochs, the accuracy recovered to the same level as before. Combined with the distancing, this means that areas of low loss can be found everywhere on the landscape. This is consistent with current literature like in [9]. But in most cases, the network with distance function doesn't top the network without. This means that even if there are areas of low loss everywhere on the landscape, no area seems to be much better than the other. Therefore, distancing is possible, but doesn't seem to be useful. If this is the case, the question if the large

exploration of the loss landscape is useful also arises, because a new area seems to not be able to achieve better performance.

For multiple checkpoints however, there is a small increase with distance function compared to without. Furthermore, the same pattern as in cosine decay arises, where new cycles top the maximum accuracy of the previous. In section 2.1.2 we suggested that this might relate to the simliar influence on the gradient, that the parameter updates just get larger at the beginning. An open question is still why this leads to a boost in accuracy. Here we will argue, that a high learning rate might lead to wider minima and therefore a better generalization.

Consider the case where the networks after some training has a high performance and seems to be in an area of converge. This means that most of the parameters are in a local minima, surrounded by areas of higher loss, though not all of them, as we have seen existing valleys. If we perform a warm restart, the inaccuracy in the gradient from batch methods for example will be amplified by the large learning rate. Consequently, the local minimum of the parameters will be overshoot from both sides and the parameter values will distance further from their local minimum. As the minima are surrounded by a high loss, the accuracy drops as we have seen in the results, section 3.4.1. But if a parameter still returns to the same minimum as before, the minimum is robust in the sense that the surrounding boundaries of large loss are high and wide. For parameters whose local minima is very narrow and another area of low loss is nearby, the values may switch to these dueto high learning rate. Therefore, parameters which have reached good minima will stay in their position, while unstable minima may be left. Here also lies a key difference to the distance function. As the distance term acts the same on all parameters, the distance will be increased to both good and bad parameter values. This could provide an explanation for a smaller boost in accuracy of 0.8% compared to 1% for cosine decay.

One major improvement of the distance function was for a wrong initial learning rate. At first, it seems that a cosine decay should let the network be more robust to a wrong learning rate because it is decayed until 0, irrelevant of the inital learning rate. In section 3.4, this was the case until the first warm restart. But after it was performed, both ResNet32 and MobileNetV2 started diverging for a initial learning rate of 0.01 in a way that the maximum learning rate of each new cycle decreased compared to the previous one. In contrast when adding a distance function, the maximum accuracy stayed constant for MobileNetV2, and decreases significantly less for ResNet32.

A simple explanation would be, that cosine decay with warm restart walks into an area of high loss, and is not able to recover. The distance function then helps to distance from this point and return to areas of lower loss, leading to a stable performance. The distance plot of figure 3.4.2 conformed to this idea, as normal cosine decay was not able to distance from the checkpoint, in contrast to configurations with the optimal learning rate. However, this would contradict both our results and most papers. As the distance function had no impact on the valdiation accuracy in section 3.1, we suggested that this happens due to the fact, that areas of low loss exist everywhere, which would contradict this explanation. Furthermore, it is unlikely that both networks would walk into an area of high loss by chance. Because the effect also only happens for this particular learning rate, the truecuase of the difference remains unclear.

In section 3.5, we saw how the performance could be improved by network ensembling. As section 1.2.4 discussed, there are two major factors which influence the performance of the ensemble. The individual performance of the network and the corellation between the network predictions.

The results showed that although the corellation is an important factor, a low corellation can not recover if the network performs worse. It seems like a low corellation leads to much larger benefits of the ensemble in comparison to the single network. Cosine decay had a much larger corellation and this resulted only in a small increase of 1%, where for our distance function, the ensemble gained over 4% in accuracy against the single network. However, cosine decay still performs much better when compared to our network due to the better single network performance. One further aspect to keep in mind is that a better network performance automatically means higher corellation. Consider a network which performs at 100% accuracy. Then all its snapshots have a corellation of 1, since all of them predict all exampels right. If networks have high but not perfect accuracy, there still has to be a high corellation, since there are not mayn examples where the snapshots predict wrong and can disagree. Therefore, a higher corellation often automatically arises and may rather predict the relative gain to a single network than the absolute performance.

For networks with lower accuracy, we tried to reduce the corellation by using our distance function. Theoretically, the ensemble should then perform better, since we have seen in the other results that a larger distance doesn't harm the accuracy, at least in long term. However, the results showed that even for larger distances, the corellation between the snapshots stayed the same. The implications for the loss landscape remain unclear.

At first, the results seem to suggest that networks with same accuracy predict the same regardless their position on the loss landscape. One explanation for this could be the weight space symetry, as discussed in section 1.1.4. Here, networks perform the same due to the symetry in weight space, which arises from swapping two units. If the snapshots are taken at these points, they perform the same even if their difference is large. However, it seems unlikely that the network converges exactly to this points after distancing from their last snapshot. If this is the case, the question arises if there may only be one truly optimal configuration for network, which can be found at different locations in weight space.

Alternatively, the robust corellation could also be explained by the dataset. Maybe there are examples, which are very hard. Therefore, the network has to exploit the loss landscape very good in order to get correct predictions. As a fixed learning rate fails to do so, these examples remain wrong regardless of the networks position in weight space. As a result, the corellations stays high due to the samne wrong predictions of every snapshot.

No matter the explanation, the results question the underlying idea of snapshot ensembles. A high learning rate increases the distance, but doesn't result in a lower corellation. Therefore, the benefits may be doubtful from a theoretical viewpoint. Nevertheless, in practice our network with suboptimal accuracy achieved a much higher accuracy as ensemble.

One aspect to keep in mind is the additional training cost. Section 3.6 showed that the distance function adds a significant amount to the epoch time. Even though this scales linear with every new checkpoint the cost is very large, especially considering small amount of improvement for most cases. Consequently if training performance is of importance, it seems that other methods like a cosine decay reach better improvement with no additional training cost. Nevertheless, the distance function stabilized the choice of the learning rate for cosine decay. If training time is not important, the distance may be worth to try out, especially because it adds no cost at test time.

Bibliography

[1] Training center for machine learning cluster of the university of tübingen.

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

[3] 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.

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

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

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

[7] 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*, pages 249–256, 2010.

[8] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

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

[10] 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.

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

[12] 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.

[13] 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.

[14] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[15] A. Kolesnikov, L. Beyer, X. Zhai, J. Puigcerver, J. Yung, S. Gelly, and N. Houlsby. Big transfer (bit): General visual representation learning. *arXiv preprint arXiv:1912.11370*, 2019.

[16] A. Krizhevsky, V. Nair, and G. Hinton. Cifar-10 (canadian institute for advanced research).

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

[18] 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.

[19] 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.