Assignment 2 - Convolutional Neural Network

Course: *Advanced Machine Learning*

December 3, 2018

1 Teammembers and Contributions

- **Bernd Menia**, *01316129*: Report, Programming, Testing, Plots
- **Maximilian Samsinger**, *01115383*: Main Programming, Testing

The difference in our contributions is because Bernd just only started working with Machine Learning, while Maximilian has already prior knowledge in the subject. Thus we worked together, but the main programming work for this assignment was done by Maximilian. Apart from that due to the size of the code we will only list the most important parts and explain them in detail with the gathered results.

2 PC Specifications and Original Example

As requested by the exercise we first downloaded the original **CIFAR10** example from the keras-team git (source). Although it wouldn't run right away because apparently some variables weren't initialized in the original code. After adding a few lines to the code it ran just fine though. According to the Keras-team the CNN should get to 75% validation accuracy in 25 epochs and 79% after 50 epochs. We calculated 100 epochs of the code, but we cannot quite confirm the claims. After 25 epochs we only got to a validation accuracy of $\approx 73\%$. Furthermore the validation accuracy reached its plateau at at averaging $\approx 76\%$ after about 34 epochs and didn't change much more, even after running all 100 epochs. Figure 1 shows the differences of the validation accuracy between the original claims, our run of the original example and multiple modifications that we applied to the code. The original example is hereby represented by the black curve.

Since it took a long time to run the examples on our hardware we often only computed 25 epochs to see if the results changed noticeably from the original example. If they didn't change much we stopped execution to save time. For changes that had a positive impact we let them run through all 100 epochs. Finally we also doubled our batch size from 32 to 64 which improved the results a little bit. We also tried to increase the batch size for the first 2 layers up to 128, but results only marginally improved and where almost negligible. However in the latter case the computation time increased by about 50% which made it impracticable and we therefore decreased the batch size again to 64 for each layer which seemed to have the best tradeoff between accuracy and computation time in our opinion.

3 Modifying the Code

Ideally we wanted our CNN to have up to 90% test accuracy, i.e. validation accuracy. To achieve this we altered our code from different viewpoints and checked each time how the accuracy of the CNN changed. Although because on our hardware each epoch took about 3 minutes to complete we often only tested the first 25 epochs to check how the values changed.

3.1 Batch Normalization

The first thing we tried out was to use **Batch Normalization**, that is to normalize the outputs of the layers, i.e. the values get standard normal distributed. Doing this is convenient because we reduce the span of values that each node in the CNN can output. Without Batch Normalization the CNN would possibly have to learn how outlying values behave which could increase the calculation time. Also when we consider multiple metrics for our values it is useful to have them on the same scale. However Batch Normalization does not punish extremely positive or negative values for weights, which proved to be critical in our Ridge Regression exercice. This is where **Kernel Regularization** comes into play. Though we will only look into it in subsection 3.3.

We used the built in Batch Normalization function from Keras to dampen our values, so we don't know exactly how the values get altered. Also when looking at the results we didn't see much of a difference compared to the original example. To not clutter up the plot we therefore didn't include this run through in the plot in Figure 1. However since in our opinion Batch Normalization is good practice we let it in the code with the ret of the alterations. Also we believe that it could have positive effects on the accuracy overall, but we didn't test every permutation of all different changes we made because this would require us to run dozens of different tests, for which we didn't have the time.

3.2 Stochastic Gradient Descent (SGD)

The next thing we implemented is the usage of **Stochastic Gradient Descent (SGD)**. The reason for this is the same as before, we want to minimize the test errors in our CNN. Think of SGD in the following way: Suppose you have a skyline of mountains like depicted by the curve in Figure Y. You are standing on the top, i.e. the start for our CNN and you want to go to the minimal height level, i.e. the minimum test error in our case. Now there are two main questions: **1)** how big should the steps be that we take and **2)** in which direction should we go? If our steps are too big then we are prone to stepping over the minima and going the mountain back up on the other side which is obviously not desired. However this problem is almost unavoidable unless we step exactly onto the minima. To combat this nuisance SGD introduces a decay factor which continuously reduces the step size until we get to our minima (until the distance to our minima is within a given delta???). By trial and error we chose $1e-4$ as the preferred decay.

But still the problem is not yet solved. The second problem that we have to overcome is that if we just take steps in one direction then we could step over the minima which would mean that we would

climb the next mountain and get increasingly further away from our minima. SGD takes into account the last few steps and checks if we get closer to the minima, or further away from it. If we get closer to the minima then the direction doesn't change. In contrast if we get further away then we flip the direction in which we take our steps.

To make our lives easier we also added a momentum factor of 0.9. The higher this factor, the more emphasis lies on the first step sizes in the sense of that we take bigger steps. Speaking from our metaphor point of view this can be thought of as how quick we are taking our steps. Are we slowly strolling down the mountain (low momentum) or are we running (high momentum)?

These properties are good enough to make sure that we will get to the minima if we only consider two mountains, i.e. a quadratic curve. However what happens if we have 3 mountains and the minima is between the second and third mountain? If our step size is not big enough to go over the second mountain then we are stuck between the first two and get to their minima, but we never actually get to the global minima, i.e. between the second and third mountain.

In our code SGD is represented with the following line: *opt = keras.optimizers.SGD(lr=0.01, momentum=0.9, decay=1e-4, nesterov=True)*. We computed 100 epochs with SGD and Batch Normalization and we could ascertain that the average validation accuracy went up by about 2% from $\approx 76\%$ to $\approx 78\%$. There are many different types of adaptive methods that perform similar tasks to SGD, for example **Root Mean Squar Propagation (RMSPRop)** or **Adaptive Moment Estimation (Adam)**. Overall RMSProp and Adam achidve better results regarding training errors, but we chose SGD because it generalizes better and has therefore lower test errors.

3.3 Kernel Regularizer

The problem with outlying values is that they have an unproportional high impact on the CNN as a whole when compared to more standard values. In subsection 3.1 we have already talked about dampening the effect of variables to get normalized values. Adding to that Kernel Regularization is used to dampen the effect of specific values, more specifically outlying values. In principal this is the same process as dampening the values in a linear regression model with the λ value. We used the built-in Kernel Regularizer from Keras to modify our code and added it to each layer of the CNN. By trial and error we chose a value of $1e-4$. We computed 100 epochs of our CNN with BN, SGD and the just added Kernel Regularizer and achieved astounding results. The average validation accuracy went up from $\approx 78\%$ to $\approx 86\%$, which is a $+8\%$ difference.

3.4 Activation Function

Finally we got rid of the **Rectifid Linear Unit (ReLU)** activation function and replaced it with a modified version, **Leaky ReLU**. When training the CNN it can happen that a neuron happen to die (become inactive) and from that point onwards the output of said neuron gets ignored and is therefore useless. Normally when using ReLUs the value of inactive nodes is 0, but Leaky ReLUs allows neurons to have small positive value when they are inactive. This prevents neurons from completely dying so that they

have still an impact and can also possibly recover and become more active again. In practice, we saw that Leaky ReLU already started to converge better after 10 epochs which was enough to convince us to keep Leaky ReLU as our preferred activation function.

PReLU.

4 Discussion

During the course of this assignment we tried out many different modifications to the given code. Figure 1 shows the plot with the most important changes that we made. Solid curves represent the validation accuracy (test accuracy) while the dashed and less opaque curves represent accuracy (training accuracy). Both curves are always given in pairs with the same color and correspond to one group of major changes. For example the black curves represent the original example without any modifications. Batch Normalization and Stochastic Gradient Descent are visible in the red curves. As we can see results improved slightly. Even more so curve for the validation accuracy doesn't jiggle as much anymore which is convenient. Finally by also adding Kernel Regularization we could improve the average validation accuracy from $\approx 76\%$ to $\approx 86\%$, i.e. $+10\%$ which corresponds to a percentual gain of 13.15%. These values are represented by the black- and green curves respectively.

What's interesting to note is that for the original example and also the modification with Batch Normalization and SGD the validation accuracy is higher than the accuracy. This is unusual, but we are unsure why this is the case. After adding the Kernel Regularization the curves swap position and the validation accuracy is lower than the accuracy as it was expected by us.

There are many more things that we could test and improve, such as testing more activation functions and different permutations of possible improvements, i.e. testing every activation function with and without Batch Normalization, with and without Stochastic Gradient Descent and so on. We also tried to remove the last Dropout (0.5) but that proofed to decrease the training and test error and we therefore put it in again.

5 Meep

For two of our largest improvements we where in part inspired by (1). (SGD + Momentum + Nesterov and Batch Normalization) Regarding (3), we wanted to remove Dropout and replace every instance of Convolution + Activation Function with Convolution + Activation Function + Batch Normalization. This was horribly expensive, therefore we only replaced each instance of Dropout(0.2) with Batch Normalization and it worked very well.

Figure 1: Comparison of validation accuracies for different implementations