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 curves in Figure 1.

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 number of kernels 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 number of kernels 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.

Since some initial testing provided some evidence that Batch Normalization increased performance, we used it in all further experiments instead of all but the last dropout layer. However, we did not conduct a 100 epoch run due to time constraints and therefore it will not be separately included in Figure 1.

3.2 Stochastic Gradient Descent (SGD)

Next, we experimented with different optimizers and their parameters. After some trial, error and some guidance from the internet [Rud16], we managed to achieve stellar results with **Stochastic Gradient Descent (SGD)**. We used a large momentum factor of 0.9 and instead of regular momentum, we use Nesterov momentum. Nesterov momentum simply replaces the direction of the momentum with an updated (corrected) direction using the current gradient, but otherwise behaves like the regular momentum updating scheme.

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 $\approx 78\%$. Instead of SGD there are other popular optimizers that we could have used, for example Root Mean Square Propagation (RMSPRop) or Adaptive Moment Estimation (Adam). In general RMSProp and Adam achieve better results regarding training errors, but SGD tends to generalizes better and therefore has lower test errors. We could confirm this in our experiments.

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. This improvement is represented by the green curves in Figure 1.

3.4 Activation Function

During training we can suffer from the “dying ReLU” problem. When using ReLUs negative values get set to zero and in unlucky cases this can cause some neurons to be permanently inactive and not getting updated during back propagation. Leaky ReLUs allows neurons to have small negative value when they would otherwise be 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.

There is also another related implementation of ReLU, **Parametric ReLU (PReLU)**. Similar to Leaky ReLU it creates new parameter a and multiplies said parameter with the negative value of the neuron. Unlike Leaky ReLU, the parameter is learnable and not fixed. Though when considering $a \leq 1$ then the output of the neuron behaves as if $f(x) = max(x, ax)$ [HZRS15]. Due to timing constraints we haven’t yet tested changing our activation function from Leaky ReLU to PReLU and therefore don’t have any data on its impact.

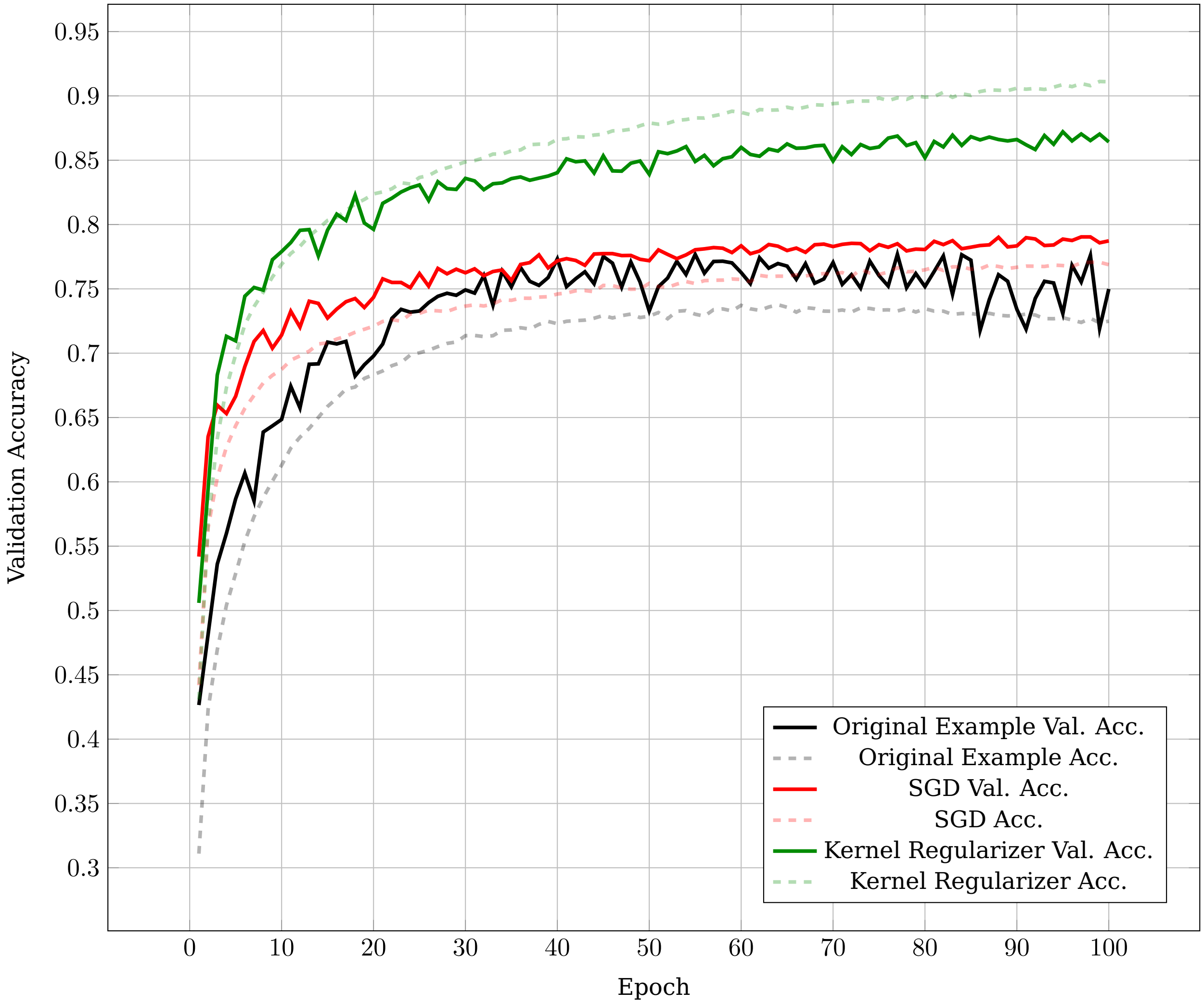
4 Discussion

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. We assume this happens due to the image augmentation step, which creates images that are possibly more difficult to recognize. 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.

Figure 1: Comparison of validation accuracies for different implementations



References

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015.

[Rud16] Sebastian Ruder. An overview of gradient descent optimization algorithms. URL: http://ruder.io/optimizing-gradient-descent/index.html, January 2016.