Deep Learning Lab

Autonomous Intelligent Systems

Exercise 2

Author: Eduardo ALVARADO

Matriculation No. 4454388

19th November 2017

1. Introduction

In the following exercise, we need to implement a CNN in TensorFlow and training it on the MNIST dataset.

This CNN will consist of two convolutional layers (16 filters of 3x3 each, and stride of 1). Each is followed by ReLU function and a max pooling layer. The Fully Connected Layer (after the convolutional layers) has 128 units and softmax makes the classification. For the training, the optimization consists on cross-entropy loss with Gradient Descent. All these experiments have been executed by using **10 epochs** and a **batch size of 50**.

The code with the implementation can be followed in the file: *mnist_solution_final.py*

2. Changing the Learning Rate

In this experiment, we are training our dataset for different values of Learning Rates. The Learning Rate (normally described by λ) is an hyperparameter for Gradient Descent Optimization. It determines, how fast or slow I move towards optimal weights (which minimize our Loss Function). If λ is very large, we could overpass the optimal weight, but if it is too small, a larger number of iterations will be needed to converge to better values. For that reason, is very important to find the appropriate value of λ.

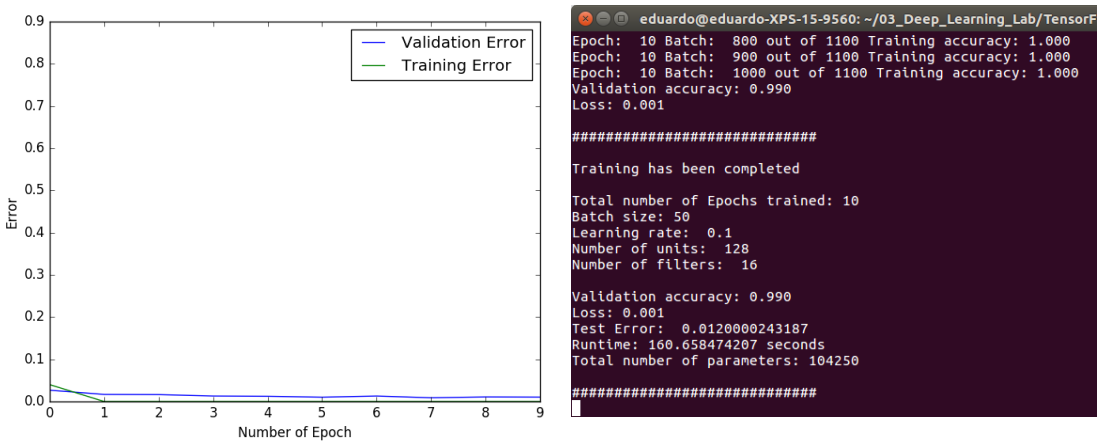

Figure 1: Learning Rate: 0.1

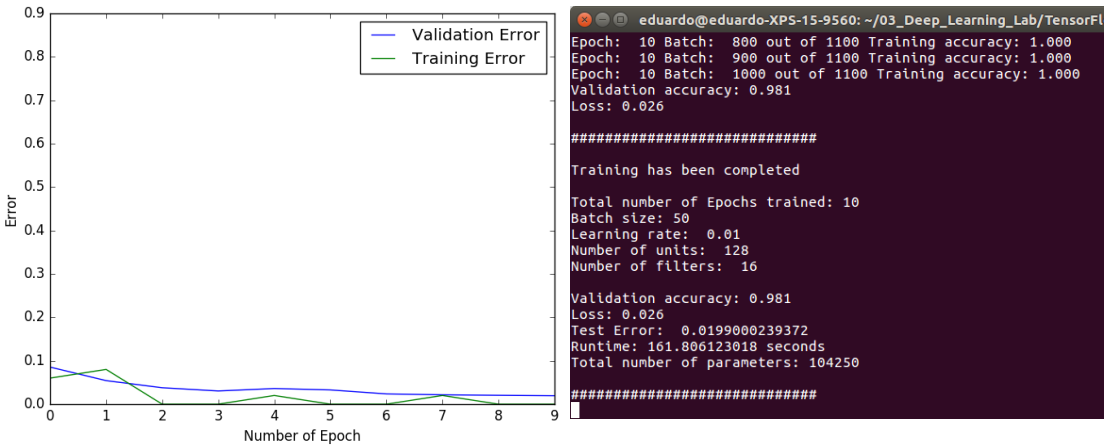

Figure 2: Learning Rate: 0.01

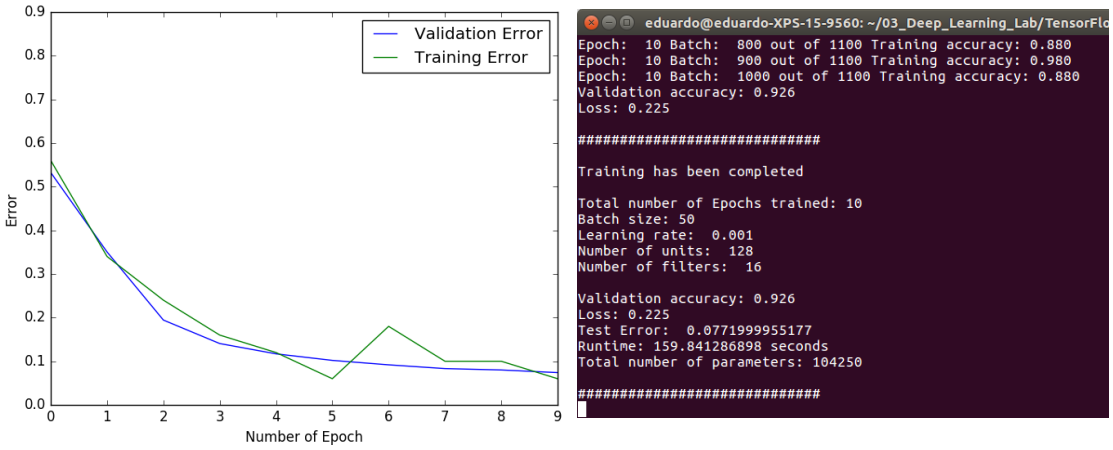

Figure 3: Learning Rate: 0.001

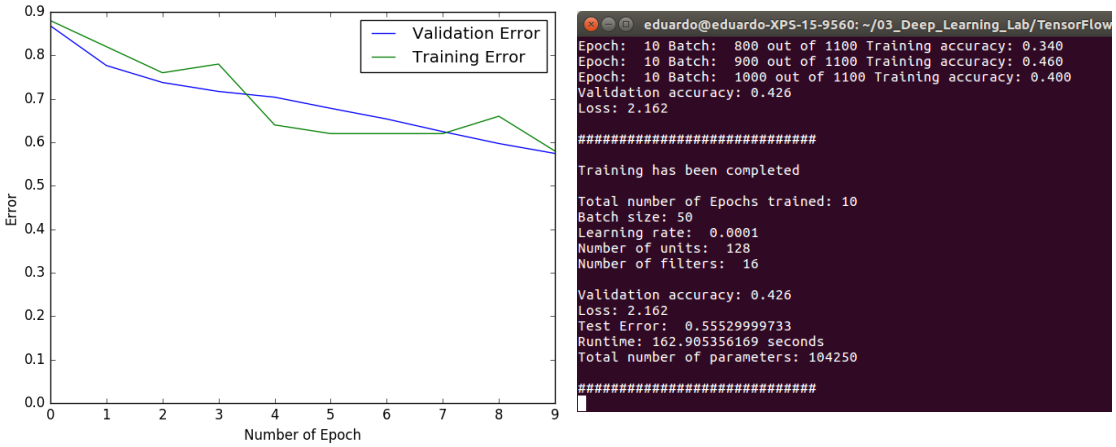

Figure 4: Learning Rate: 0.0001

In our examples we can see such effect. If we decrease our Learning Rate, the validation/training error increases (therefore, accuracy decreases), because we are not able to find an optimal weight for our system. In this case we can conclude, that our experiment works better with an appropriate Learning Rate λ of 0.1. Anyways, we need to be careful and to not set a very large value, since the optimization could overpass through our optimal weight and it would not converge anymore.

2. Runtime

In this part, we will see how the runtime changes when we modify the number of filters. The Figures represent the outcome for each number of filters on CPU.

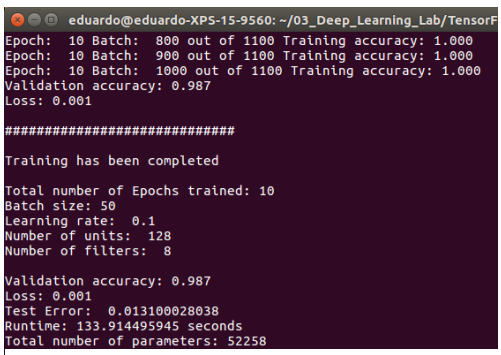

Figure 6: 8 Filters

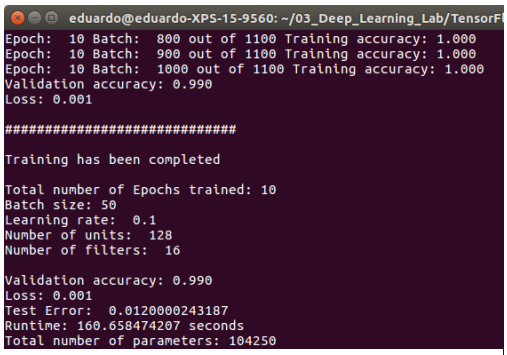

Figure 5: 16 Filters

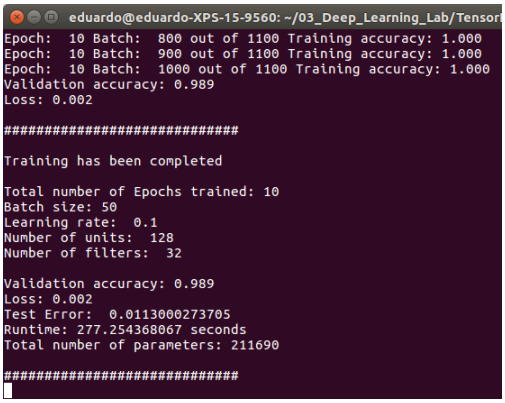

Figure 7: 32 Filters

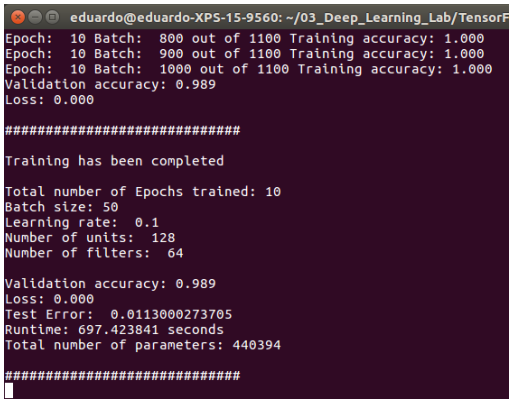

Figure 8: 64 Filters

We can observe, how a larger number of filters increases the training runtime. This is because the number of total parameters that must be trained is more if we have more filters. The following table and graph show this relation (the rest of parameters are the same, and the λ is set to 0.1):

Number of Filters	Number of total parameters	Runtime (seconds) CPU	Runtime (seconds) GPU
8	52258	133.91	48,83
16	104250	160.65	65,97
32	211690	277.25	78,77
64	440394	697.42	144,25
128	953098	-	278,69
256	2199690	-	623,43

Table 1: Data for each filter configuration for CPU and GPU

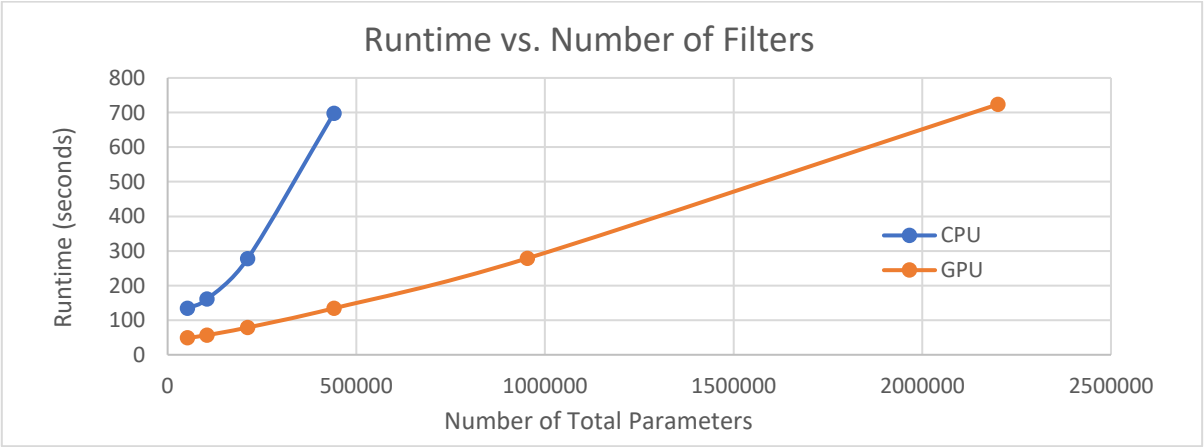

Figure 9: Runtime vs. Number of Filters

For CPU, we can observe, as we said before, that the runtime increases when the number of parameters becomes larger. This increment is even more, when the number of filters reaches large values. That means that the relation is not linear (even becomes exponential), and we can expect longer runtimes progressively when the number of parameters tends to higher values.

This problem is significantly reduced, when we train on GPU. Although also the runtime increases, the dedicated memory helps to reduce it drastically for every filter configuration. For example, the runtime needed for 64 filters on CPU, is approx. the same than the one required for 256 filters on GPU. As we can notice, this is essential for performance-critical applications.