University of Freiburg

Deep Learning Lab

Neurorobotics/Robotics

Assignment 2

Author:
Mostafa Hussein

Supervisor:
Andreas Eitel

November 20, 2017

Abstract

This report is summarizing the task of Assignment 2 which was implementing a convolutional neural network using $tensorflow$ for MNIST dataset classifcation. The report is divided into 3 sections as follows: Section 1 represent a simple explanation of the implementation, section 2 represents the designed CNN and section 3, represents the results.

1 Convolutional Neural Network

This exercise was an implementation of a Convolutional Neural Network (or CNN) to classify the MNIST digits with it. Firstly, MNIST dataset were loaded, and then $tf.placeholder$ nodes were created for saving the dataset. Afterwards, $conv2d$, max_pool_2x2, $weight\_variable$, and $bias\_variable$ functions were defined to initialize the weight and bias variables, as well as, the convolutional and pooling layers. A $deepnn$ function was defined to build the model for classifying digits. Then, the network was trained by optimizing the cross-entropy loss with stochastic gradient descent. Finally, a $softmax$ output was used for the output layer.

2 Convolutional Neural Network Design

The designed convolutional neural network consists two convolutional layers with 16 features, and 3x3 filters with a stride = 1. A 2x2 max_pool layers were used after each $conv\_layer$ to downsample by 2x. The final $fully\_connected\_layer$ after 2 round of downsampling leads our 28x28 image to be down to 7x7x16 feature maps, then this is mapped to 128 features by having 128 units, and a $softmax$ layer was used to do the classification by mapping the 128 features to 10 classes, 1 for each digit. Dropout was used to control the complexity of the model and prevents co-adaptation of features. $GradientDescentOptimizer$ with a $learning\_rate = 0.1$ was selected as the optimization algorithm for stochastic gradient descent, and the optimization is done by minimizing the $cross\_entropy$ loss. Finally, an interation over 30 epochs and a $batch\_size$ of 50 were chosen to train the network, and calculate the $training\_error$ and the $validation\_error$ after each epoch. The results of the designed network are discussed in the next section.

3 Results

The results for the designed neural network is shown below in Figure 1. As shown from the below graph, that along the training over 30 epochs, the training error starts from 3.9% and it decreases along the epochs until it converges to a 0.0%. However, the validation error starts from 2.59% and it keeps decreasing along the training process until it reaches a value of 0.819%. The testset was used to compute the test error after training the network, and it gave out an error of 0.99%.

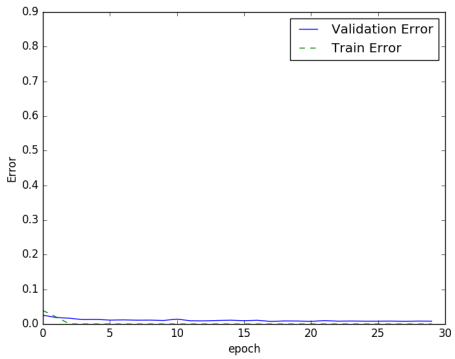


Figure 1: Results of the designed CNN

3.1 Changing Learning Rates

After implementing the CNN, and making sure that everything is going well and a good results were achieved. It was now the time to do some optimization and start tuning hyperparameters to see the effect on the learning curve. Accordingly, different values of *learning_rate* were used. The following values of $\{0.1, 0.01, 0.001, 0.0001\}$ were tried during training and then the learning curves were plotted to see the difference and which one is working best. The results of the training for these rates are shown below in Figure 2.

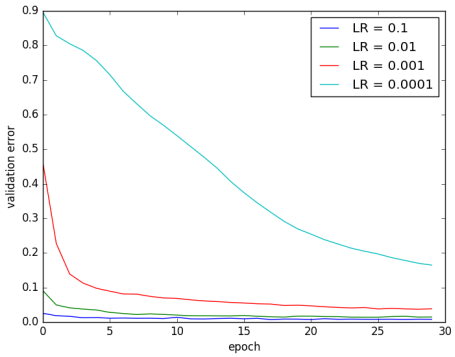
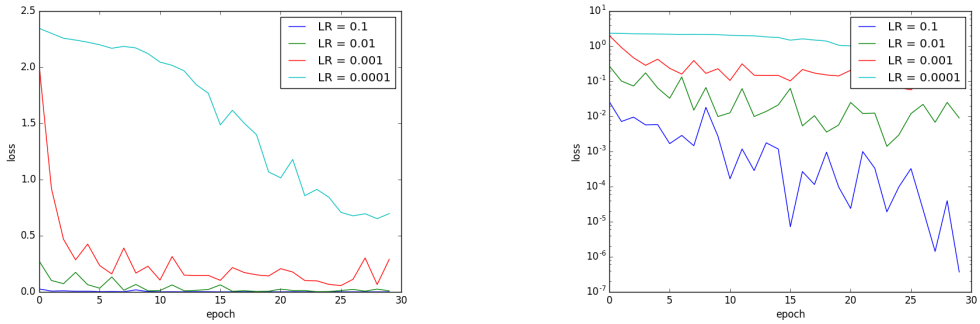


Figure 2: Learning curves for different learning rates

It is obviouvsly shown from the above graph that a learning rate = 0.1 is giving much better accuracy than other learning rates. We can see from the learning curves of the 4 values we have, that a learning rate of 0.1, starts with a validation error of 2.59%, and then converging to 0.819%. However, a learning rate of 0.01 gives an error of 9.07% and reaching 1.48%. As for the learning rate of 0.001, the error starts with 46.1% reaching 3.83%. While the worst learning rate was 0.0001, because it gave out a starting error of 89.66%, and along the epochs, it reaches 16.4% which is considered too high. Moreover on computing the test errors for the 4 learning rates, it gives out that the learning rate of 0.0001 is the worst of them. To clarify, learning rate of 0.1, gave a test error of 0.99%, and learning rate of 0.01 gave a test error of 1.49%, however, for a learning rate of 0.001, the test error was around 3.51%, and finally, for the learning rate of 0.0001, the computed test error was 15.65%, which is again too high for the network.
It is clearly shown also from Figure 3, that the loss function of learning rate = 0.1 is better than the others, as it decreases and converges to 0 so fast, better than learning rate of 0.01 which also converges to 0 but at a slower rate. However, for the other learning rates, the loss is not converging to 0 at all.



(a) Not scaled

(b) Scaled with logarithmic fn.

Figure 3: Loss Function for different learning rates

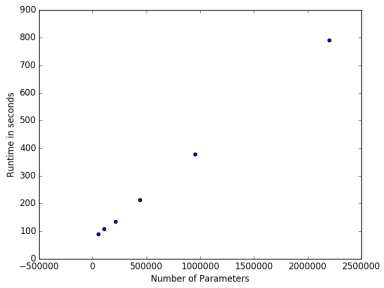
3.2 Runtime and Parameters

A second experiment is conducted on the network to see how will training time be affected by changing the number of units in the layers. Accordingly, the neural network was trained on a GPU with these set of number of filters $\{8, 16, 32, 64, 128, 256\}$, and for each training, the runtime was measured and the total number of parameters was computed. It was so clear, that the number of parameters increases for higher number of filters. Moreover, another test was done on the CPU with number of filters $\{8, 16, 32, 64\}$, to see the difference between the runtime when training on a GPU vs. CPU. There was a great difference between in the runtime. This can be shown below in Table 1.

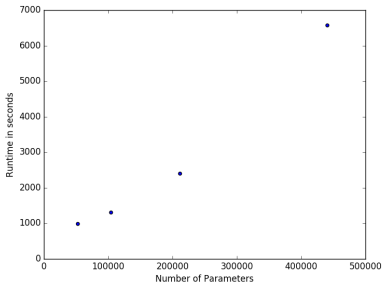
Filter Size	**Number of Parameters**	**Runtime GPU (seconds)**	**Runtime CPU (seconds)**
8	52,258	90.7	993.0
16	104,250	108.4	1314.1
32	211,690	135.9	2405.0
64	440,394	213.3	6571.7
128	953,098	378.5	-
256	2,199,690	792.1	-

Table 1: Number of parameters for different filters

From the results above, GPU is much faster for training than CPU. as we can see for same number of filters, and same network, the runtime is much higher when the network is trained on CPU compared to GPU. A scatter plot was done to show clearly how the graph is changed between CPU and GPU for number of parameters vs. runtime in seconds, and this can be shown below in Figure 4. So after running on the CPU, there is an exponential relation between number of parameters and runtime, which is a total different behavior with the GPU. And from this, we can conclude that GPU is more capable of achieveing better computations than the CPU, and we can then train our network with more parameters faster than doing it on CPU. As an example we can see, that for a runtime of around 800 seconds, GPU can train a network with 256 filters, however, in the same amount of time and maybe more, CPU can train a network with 8 filters only, which is very less efficiently.



(a) Training on GPU



(b) Training on CPU

Figure 4: Network trained with different number of filters on GPU and CPU