KTH
ROYAL INSTITUTE OF TECHNOLOGY

Computer Assignment #7

Student name: *H Hellström, M Tavana, F Vannella, and M Zeng*

Course: *Fundamentals of Machine Learning over Networks (EP3260)*
Due date: *June 4th, 2020*

Question 1

Consider the "MNIST" dataset, and a DNN with J layers and N_j neurons on layer j. Train DNN using SGD and your choices of hyper-parameters, L, and $N_j \in [J]$. Report the convergence rate on the training as well as the generalization performance.

Answer. We chose to use categorical cross entropy as a loss function since this is appropriate when considering multiple classes (10) and only one result per image (each image contains only one digit). However, we did not go with vanilla SGD because it took over 100 seconds to complete a single epoch, instead we went for mini-batch GD. Mini-batch SGD is slower per iteration, but since one epoch requires the entire dataset to be considered, mini-batch requires fewer iterations per epoch than SGD. We used a single hidden layer with 100 nodes, and l2 regularization on the hidden layer weights. For the update, we kept it simple and have no decay or momentum, which means that the update has the basic form:

$$w_{k+1} = w_k - \alpha \nabla_w f(w_k), \tag{1}$$

where α is the learning rate. The hyper-parameters used for question 1 are available in Table 1. The results of the training can be seen in Figure 1, total training time was 133 seconds. The accuracy on the training dataset follows a steady increase while the test accuracy is more erratic, as expected. The generalization performance seems good, since it looks like the test accuracy is oscillating around the training accuracy. The final accuracy on the training data was 95.9%, on the test data it was 95.6%. Finally, we decided to look at a few example of correct and incorrect guesses, which can be seen in Figure 2.

Parameter	Value
J	3
N_j	$\begin{pmatrix}28^2 & 100 & 10\end{pmatrix}^T$
λ (regularization)	0.01
α	0.1
Batch size	100

Table 1: Parameters and DNN dimensions for question 1.

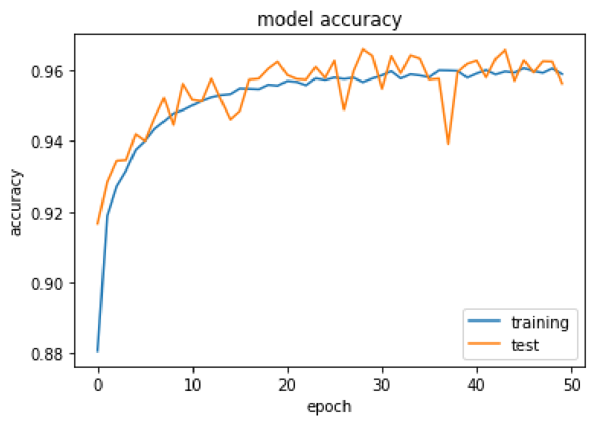

Figure 1: Convergence results for question 1.

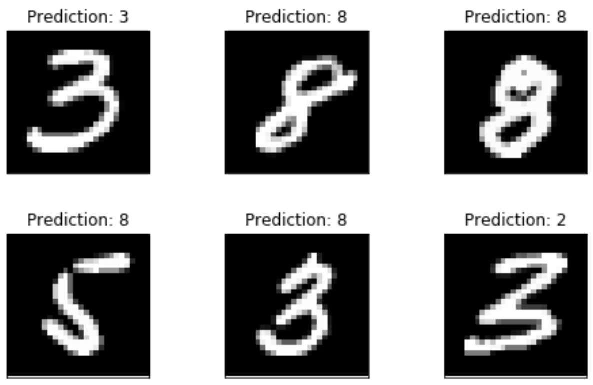

Figure 2: Example of 3 correctly (top) and 3 incorrectly (bottom) classified digits using the question 1 model.

Question 2

Repeat question 1 with mini-batch GD of your choice of the mini-batch size, re-train DNN, ans show the performance measures. Compare the training performance (speed, accuracy) using various adaptive learning rates (constant, diminishing, Adagrad, RMSProp).

Answer. Since we already used mini-batch GD for the first question, we'll focus on the learning rates here. We already used a constant learning rate for question 1, so next we'll do a diminishing learning rate. This is the method used:

$$\alpha_k = \frac{\alpha_0}{1+\delta k} \tag{2}$$

where α_k is the learning rate on iteration k, α_0 is the learning rate on iteration 0, and δ is the decay parameter. We will also take a look at Adagrad in which the learning rate is different with respect to each parameter. Specifically, the weight update takes on the form:

$$\boldsymbol{w}_{k+1}^i = \boldsymbol{w}_k^i - \alpha_k^i \nabla_{\boldsymbol{w}} f(\boldsymbol{w}_k) \tag{3}$$

where the weights and learning rates are not only indexed by iteration k but also the weight index i, so that each weight gets their own learning rate. The learning rate itself looks like:

$$\alpha_k^i = \frac{\alpha_0}{\sqrt{g_k^i + \epsilon}} \tag{4}$$

where $\boldsymbol{g}_k$ is a vector whose elements are the sum of the squares of the gradient w.r.t. w^i up to iteration k. The idea behind this update is to reduce updates of parameters with frequently occurring features and increase the updates of rarely occuring features. If a feature occurs only rarely, the update should be large to make sure its properties are captured even if the gradient is mostly near-zero. The new parameters used are in Table 2 and results are shown in Figure 3. As expected, the total time to train these models are about the same as question 1 with 134 seconds for diminishing learning rate and 141 seconds for adagrad. The convergence rates for constant and diminishing learning rate are very similar, but adagrad experiences a significantly slower rate. At the final iteration, diminishing learning rate reaches an accuracy of 96.5% on the test data, which is 1% higher than constant learning rate, but judging from the plot this was mostly due to luck and there are many epochs for which constant learning rate was better. Adagrad had the advantage of experiencing a much smoother progression on test data accuracy and looks like it is still improving even at epoch 50. The results are better than for constant learning rate with 96.1% accuracy on the test data and could probably be even better with more epochs.

δ	0.1
α_0 (constant and diminishing)	0.1
α_0 (adagrad)	0.01

Table 2: New parameters for question 2.

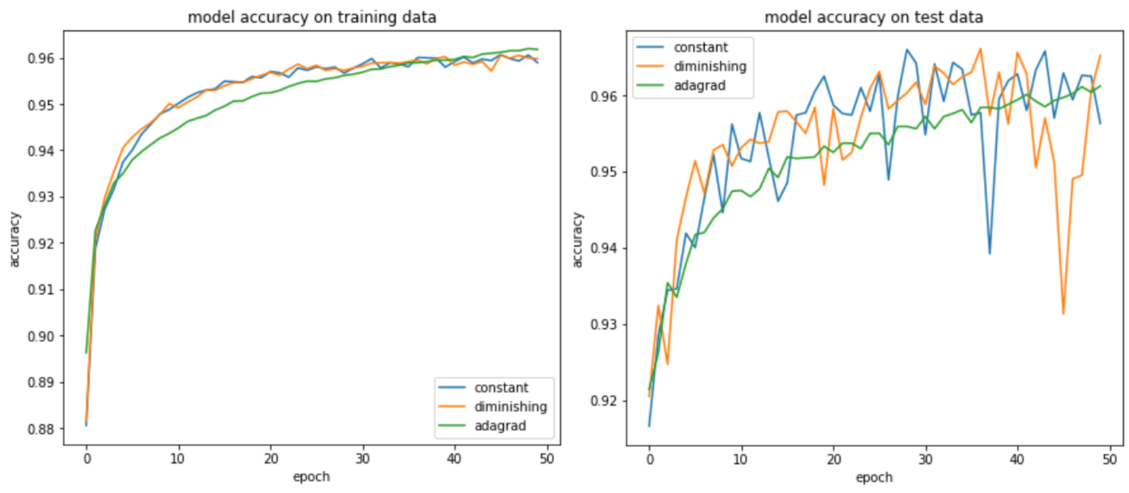

Figure 3: Convergence results for question 2.

Question 3

Consider design of question 1 and fix $\sum_j N_j$. Investigate shallower networks (smaller J) each having potentially more neurons versus deeper networks each having fewer neurons per layer, and discuss pros and cons of these two DNN architectures.

Answer. To perform these tests we reuse all of the hyperparameters from question 1, see Table 1, but the dimensions are changed to form three networks. The dimensions of these three networks can be seen in Table 3. The convergence for these models are shown in Figure 4. Training of Model 1 took 133 seconds, Model 2 took 154 seconds, and Model 3 took 140 seconds. In our experimental setup, using a deeper network was strictly worse. The training takes longer time, the convergence per iteration is slower, and the final accuracy both on training and test data is worse. However, this is probably not a general trend, there could be something specific with our parameters that causes this behavior.

Model 1	$\begin{pmatrix}28^2 & 100 & 10\end{pmatrix}^T$
Model 2	$\begin{pmatrix}28^2 & 50 & 50 & 10\end{pmatrix}^T$
Model 3	$\begin{pmatrix}28^2 & 25 & 25 & 25 & 25 & 10\end{pmatrix}^T$

Table 3: Model dimesions for question 3.

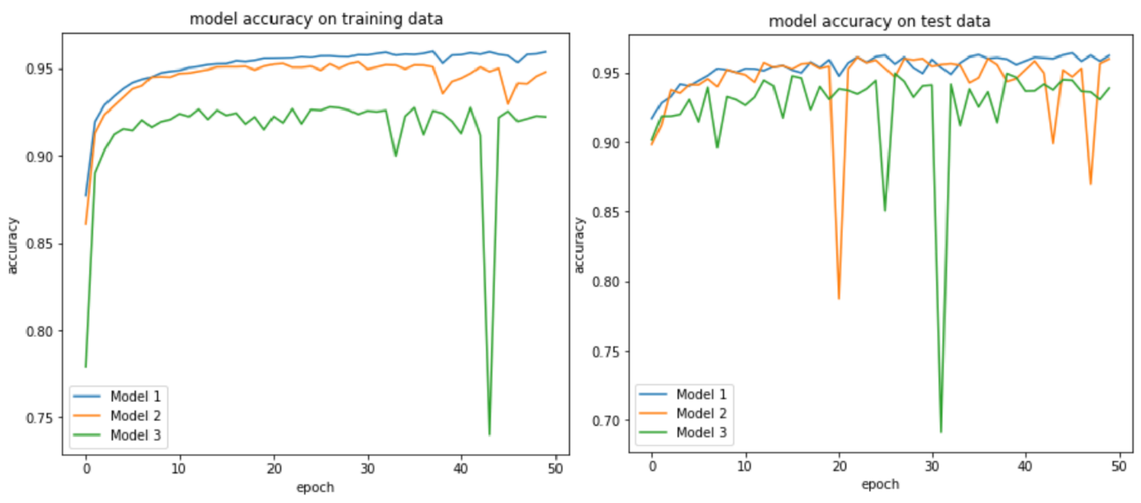

Figure 4: Convergence results for question 3.

Question 4

Split the dataset to 6 random disjoint subsets, one for each worker, and repeat question 1 on a master-worker computational graph.

Answer. For this question we tried two implementations, both the MultiWorkerMirroredStrategy available in tensorflow and the ParameterServerStrategy. The second case is probably better since it is based on distributed SGD like we worked on in the course. However, even though we got this strategy to run without errors (after a lot of troubleshooting) the program seemingly runs endlessly without making progress (As of writing this it has been running for 30 minutes with no output). The strategy is still in an experimental state and does not support some fundamental tensorflow 2.0 features such as eager execution which could be the source of the problem. Regardless, the results here are instead based on the MultiWorkerMirroredStrategy. To be honest, it is not immediately clear to us how this works. The documentation states that "the model parameters are mirrored among all workers and kept in sync with each other by performing identical updates". This description is not detailed enough to understand exactly what is happening.

Since the model weights are identical for every worker, the convergence graph looks like a single worker, but we actually have 6 working in parallel here. The results are slightly worse than question 1 despite having the same parameters, as can be seen in figure 5. Still, we end up with an accuracy on the test dataset of 94.6%.

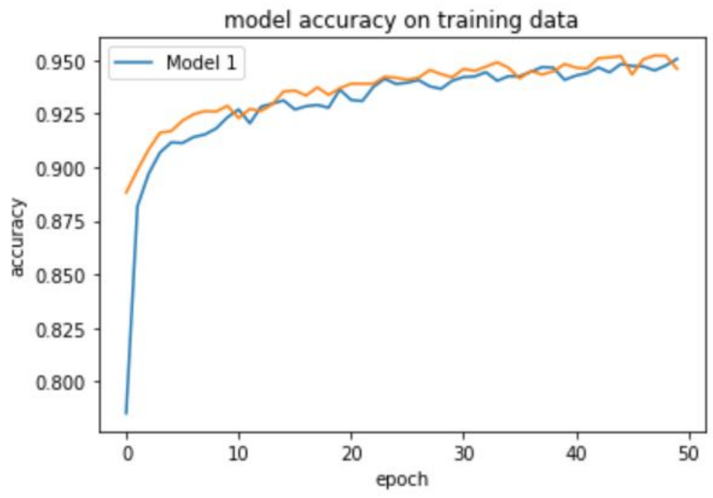

Figure 5: Convergence results for question 4.

Question 5

To promote sparse solutions, you may use l_1 regularization or a so-called dropout technique. Explain how you incorporate each of these approaches in the training. Compare the training performance and size of the final models.

Answer. l_1 regularization is implemented by adding a penalty term to the cost function. The term is given by taking the sum of the absolute value of all weights in the hidden layer. For our setup in question 1 with 100 nodes in the hidden layer we get the term:

$$\lambda \sum_{i=0}^{100} |w^i|. \tag{5}$$

Note that if the derivative is taken of this term with repect to any w^i, we only get a lambda term left. This means that as long as w^i is non-zero the change in weights (with respect to the penalty term) is going to be constant. Thus, l_1 has the property of pushing weights towards being zero, promoting sparsity in the weight vector. For this question, we use $\lambda = 0.05$, other hyperparameters are identical to question 1.

Unlike l_1 regularization, dropout is not deterministic and works with random selection. For each mini-batch, a randomly selected subset of hidden layer neurons are simply ignored, besides that training happens as normally. When one mini-batch has gone through training, a new random subset is chosen to be ignored. However, training in this way causes the weights to be very large since they compensate for being fewer during the training stage. Since the nodes are only removed during training and all nodes will be present for inference, this can lead to a problem. So after training is complete, the weights are scaled down with a fixed factor. If 20% of the nodes are chosen to be ignored for each mini-batch, then the weights are scaled down by 20% after training is complete. The new hyperparameter introduced by dropout is this percentage. For our simulations, we used a dropout rate of 20%.

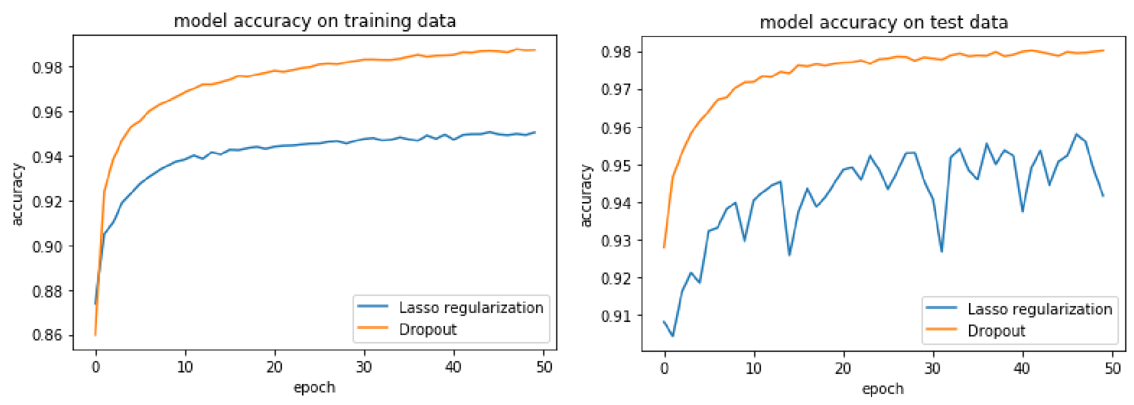

Figure 6: Convergence results for question 5.

The results of our simulation showed very good results for dropout, with a final accuracy on the test data of 98.1%, see Figure 6, outperforming every other model in this computer assignment. The l_1 regularized model performed significantly worse with a test accuracy of 94.1%, but in return we have a sparser weight vector which could be communicated more cheaply. To measure the size, we simply took the mean of the absolute value of every weight in the hidden layer. For the l_1 regularized model we got an average weight size of 0.002 and for the dropout model we got a mean of 0.06. Looking at the weight vector itself, this difference seems to be because the l_1 regularized model is sparse, with several weights of a size of 10^{-7} or less. By omitting these weights a model with smaller datasize could be obtained.

Question 6

Improving the smoothness of an optimization landscape may substantially improve the convergence properties of first-order iterative algorithms. Batch-normalization is a relatively simple technique to smoothen the landscape. Using the materials of the course, propose an alternative approach to improve the smoothness. Provide numerical justification for the proposed approach.

Answer. We will begin this answer by giving an explanation for batch normalization. The basic idea begins with normalization of input data. As we know, it is standard practice to normalize the input data to take on values between zero and one. Using this as a starting point, batch normalization is meant to normalize the activation of hidden layers as well. Do show how this is done practically, imagine a batch of datasamples as a matrix. If we have 100 neurons in our hidden layer, and 50 samples in our mini-batch, then the activations of the hidden layer can be seen as a 100x50 matrix. Each element representing the output of one hidden layer neuron for one data sample in the mini-batch. Then, we can define the mean for batch normalization as:

$$\mu_j^B := \frac{1}{50}\sum_{i=0}^{4} 9x_{ij}, \tag{6}$$

where the subscript $j \in [0, 99]$ refers to one neuron of the hidden layer and the subscript i refers to one sample of the mini-batch. The variance σ^B is defined in an analogous fashion. The batch normalization is then performed on the output of each hidden layer neuron for each sample to yield:

$$\hat{x}_{ij} = \frac{x_{ij} - \mu_j^B}{\sqrt{(\sigma_j^B)^2 + \epsilon}}. \tag{7}$$

Performing this operation on the outputs of the hidden layer and then propagating $\hat{x}_{ij}$ instead of x_{ij} during training smoothens the optimization landscape. The drawback is that we have to calculate x_{ij} for every sample in the mini-batch before we can start propagating further in the network and that we are forced to use a somewhat large mini-batch. Of course, if the batch-size is 1 this does nothing and the optimization landscape cannot be improved.

As for suggesting our own solution to smoothen, we tried an idea of input data smoothing. Using something like a Gaussian blurring filter on the input image to smoothen sudden transitions in pixels. However, the results were not noticably better than just the vanilla version, so instead we found an alternative in literature, which is layer normalization. Layer normalization is an interesting alternative to batch normalization if the feature-dimension (in our case 100) is larger than the mini-batch dimension (in our case 50) is to normalize over the neurons instead of over the batch. This also allows for pure SGD with mini-batch size 1, since the normalization is independent of the number of samples in the batch. Tthe update is very similar to batch normalization, just change the summation index from batch to neuron. As an example, the mean would then be:

$$\mu_j^L := \frac{1}{100}\sum_{j=0}^{9} 9x_{ij}. \tag{8}$$

To illustrate the benefits of normalization numerically, we chose to retrain a model with the parameters from question 1 but now include a layer normalization layer after the hidden layer. As can be seen from the convergence results in Figure 7 the model with normalization layer ourperforms the vanilla version on a majority of epochs. The final result for vanilla was an accuracy of 96.5% on the test dataset and 97.1% for the layer normalization version.

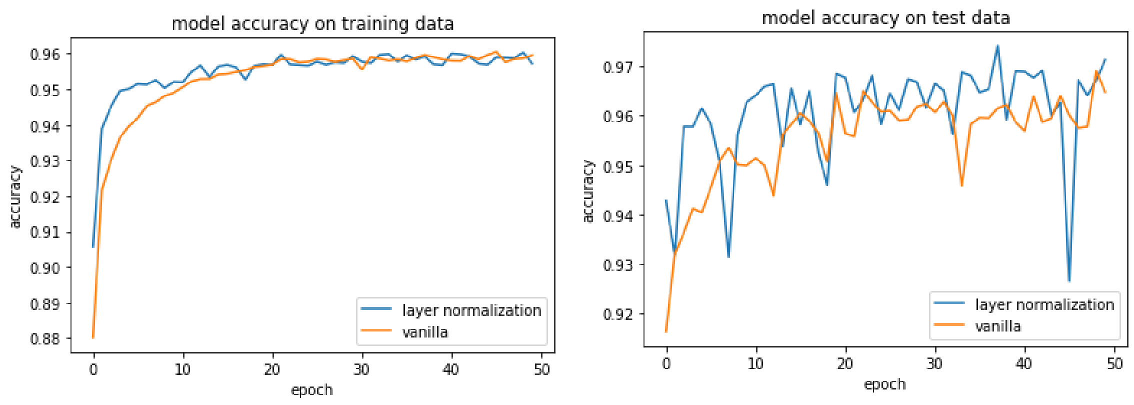

Figure 7: Convergence results for question 6.