Machine Learning - MSAI 349

Homework 3

November 16th, 2021

Professor
Dr. David Demeter

Students
Aleksandr Simonyan
Dimitrios Mavrofridis
Donald Baracskay
Xingbang Liu
Ana Cheyre

Answer for Question 1:

For question 1, we have created a simple Feed Forward Neural Network with one hidden layer. The hyperparameters used are: batch_size=1, epochs=30, stopping criteria > 90%, number of hidden_nodes = 16 accuracy on validation dataset during the training., learning rate = 1e-7. Batch size is the number of samples that will be propagated to the model. We are using stochastic gradient descent. Mini-batch gradient descent has a batch size larger than 1, but less than the sample size. The batch gradient descent takes all the samples in one batch to train the model. Small batch size saves memory and the model trains fast, but the result is not accurate. The bigger batch size is accurate but runs slow. Epoch is the number of cycles needed to finish the training. If we increase the number of epochs too much, the model could be overfitted. A way to determine if the model is overfitting is to check the accuracy in the validation set and test set. If the accuracy in the validation set and test set are dropping, the number of epochs should be decreased. The learning rate decides how much the weight should be updated during learning. Further, it decides how fast the model adapts to the problem. The higher the learning rate, the fewer epochs required, and the more likely the model will diverge. The smaller the learning rate is, the more epochs are required, the slower the model converges. We chose to set the stopping criteria as accuracy higher than 90%.

(i) Learning curves for the training and validation datasets,

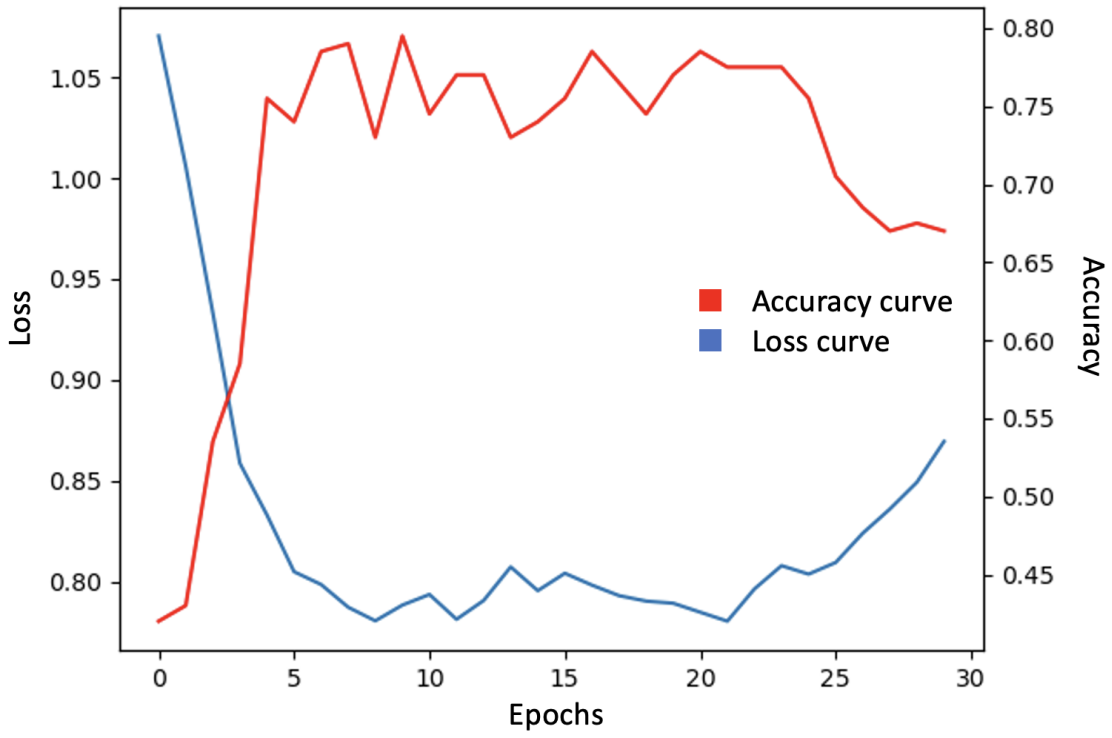

Chart 1: Learning curves for training and validation dataset using simple feedforward NN

The blue chart represents the learning curve for the training dataset, the smaller it is the more accurate the model is predicting on the training dataset. The red curve represents the accuracy of the validation dataset. We can see that there is huge volatility in both curves because we are using simple Stochastic Gradient Descent, which does not have momentum and does not use any regularization at all.

(ii) Final test results as a confusion matrix and F1 score,

F1_good score: 0.543

F1_neutral score: 0.771

F1_bad score: 0.642

19	27	2
3	96	5
0	22	26

Confusion Matrix

(iii) Short discussion of the hyper-parameters that you selected and their impact

We have experimented with different learning rates, the number of hidden nodes, and bias in our linear layers. We identified that the learning rate **1e-7** as one that produces good results, but at the same time learns quickly. Also, we compared how the accuracy changes when we remove the bias. After we remove the bias, we on average see a slight decrease in the accuracy. We have implemented one hidden layer (to keep the model simple) and used 16 nodes on our hidden layer. In general, with the increase of the number of nodes and number of epochs our accuracy increased (to a certain point of course). However, we decided to proceed with relatively simple architecture to easily reimplement it for question 4.

(iv) A description of why using a neural network on this dataset is a bad idea

Even though neural networks can approximate almost any function, one of their major drawbacks is that they are not interpretable. Especially, for the problems that require a certain ethical perspective, such as in classifying whether a person will receive insurance. We need to tell exactly why a person's application was rejected/approved and we cannot do this with neural networks. In addition, the problem that we are facing here is relatively not complex. There are much less computationally expensive algorithms that will perform relatively (or potentially better) as well as neural networks.

Answer for Question 2:

For question 2 we have created two models, one Feed-Forward neural network with seven layers, Leaky Relu as an activation function and adam optimizer, and the other one is CNN. For both models, we made an option to call dimensionality reduction and regularization. For reference, please check the *classify_mnist* function in the starter3.py file which has the following parameters: train, test, valid, regulate, use_dim_reduction, and finally use_cnn. By changing use_cnn to True, the model used to train on the data will be a custom-made CNN model. Additionally, the same functionality applies for the regulate and use_dim_reduction attributes which will activate and deactivate the appropriate functions.

The difference between this network and the one implemented in Part 1, is that here we implemented a deep feedforward neural network, which has multiple layers while in Part 1 we used only one. A NN with a single active layer can learn how to solve linearly separable problems, on the other side, multiple layers can create any arbitrary shape to separate the input data and should be able to solve any problem (at least theoretically).

The accuracy obtained with Deep Feedforward was 95% while with KNN was 92%, this difference is mainly due to the backpropagation algorithm, the network improves every time we train it with the training data, where the KNN algorithm only needs to receive the training data once for it to work. Anyway, the difference is not that significant and KNN is a much simpler algorithm to implement and very good results can be obtained.

	Deep Feedforward NN	KNN
Accuracy	95%	92%

Table 2: Accuracy of Deep Feedforward and KNN models

18	0	0	0	0	0	0	0	0	0
0	19	0	0	0	0	0	0	0	0
0	0	25	0	0	1	0	0	0	1
0	1	1	19	0	0	0	0	0	0
0	0	0	0	25	0	0	0	0	0
0	0	0	0	0	13	0	0	0	0
0	0	0	0	0	0	24	0	0	0
0	0	0	0	0	1	0	17	0	0
0	1	1	0	0	0	0	1	19	0
0	0	0	1	1	0	0	0	0	11

Table 3: Confusion Matrix using deep feedforward

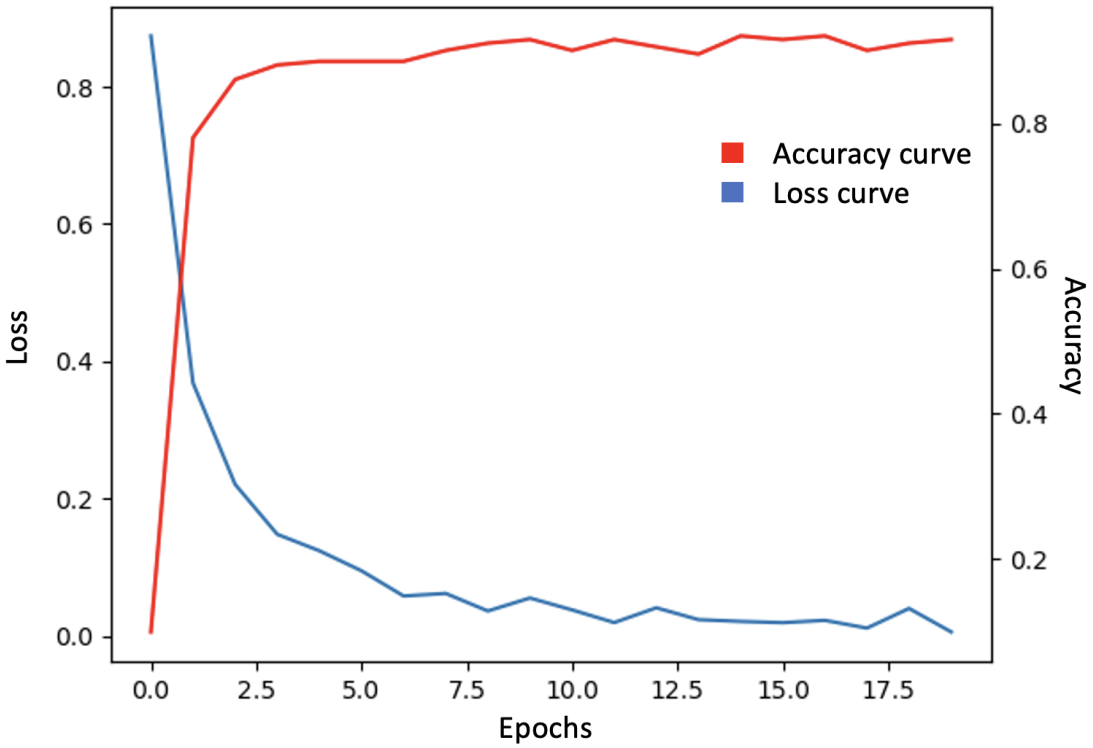

Chart 2: Learning curves for training and validation dataset using deep feedforward NN

The Adam optimizer helps us optimize the parameters much more efficiently compared to simple SGD. The dimensionality reduction we performed removed 257 dimensions. We achieved pretty good results with FeedForward which was equal in general to 95%. We have also implemented CNN with 2 Convolutional layers and 2 Maxpool operations. The accuracy for CNN was even higher, sometimes even reaching 99% percent. For CNN to work we needed to convert our data to 4-dimensional space (2000,1,24,24) after the dimensionality reduction. CNN is obviously much stronger for classifying images than FeedForward.

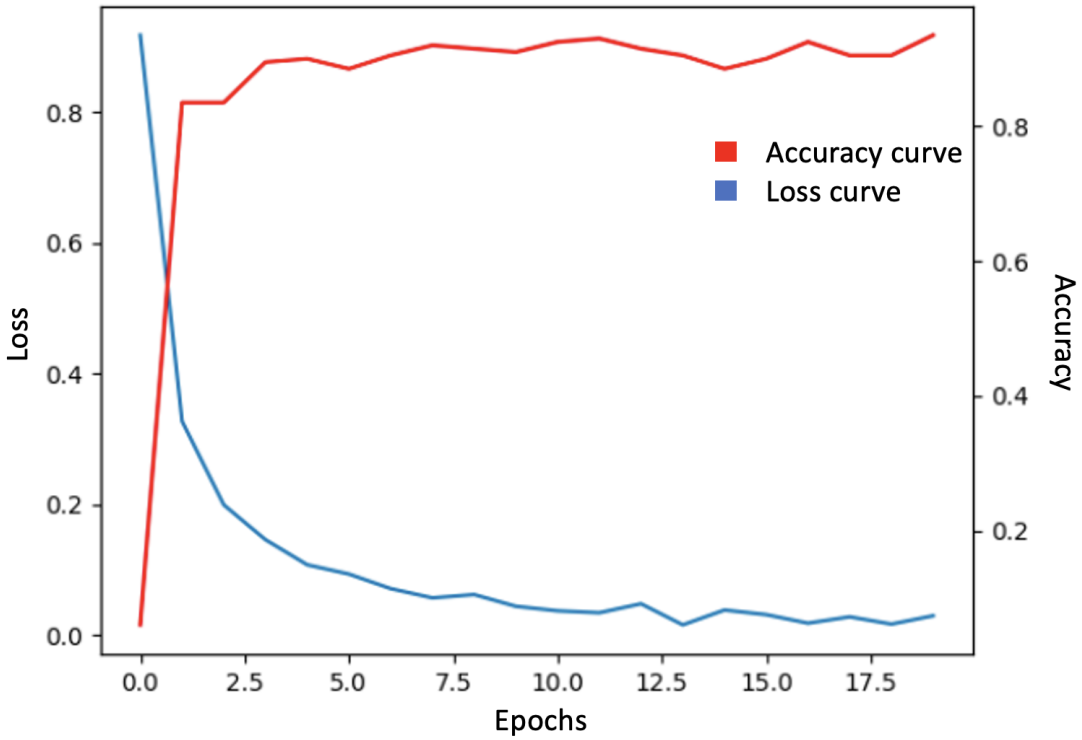

Chart 3: Learning curves for training and validation dataset using deep CNN

Answer for Question 3:

We added L2 regularization to the FeedForward Network and the CNN that we implemented for question 2.In addition, we added a Dropout Layer for CNN to reduce overfitting. As a result, we received better and more accurate results. One can see how the learning curves changed drastically. We can see that the validation accuracy curve in chart 5 is much smoother than the one in chart 3. The L2 regularization added to our cost function, allowed us to reduce overfitting. We decided to go with L2 because L2 makes a set of predictions smoother and has high volatility in our prediction.

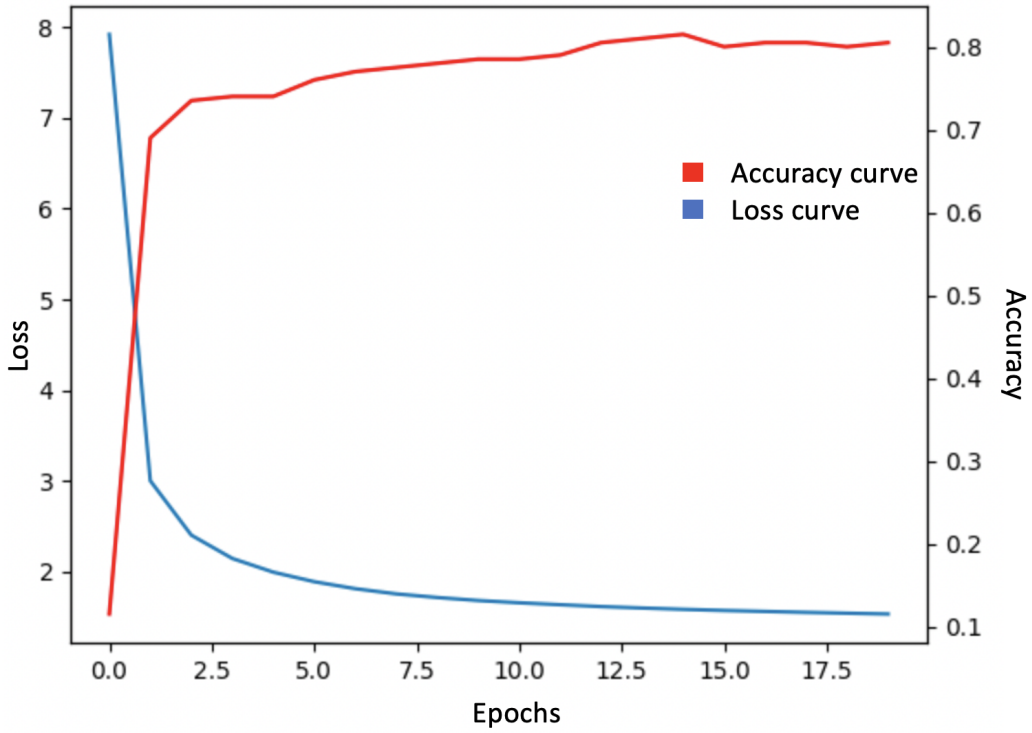

Chart 4: Learning curves for training and validation dataset using deep feedforward NN with regularization

The regularization did provide an improvement This improvement is mainly due to the model being able to better generalize and thus perform better on other data. Of course, there was a trade-off between regulation and accuracy. When we us small l2_weight (such as 0.01), the accuracy does increase but overall smoothness does change significantly, when we set up high l2_weight, such as 0.1, the validation lines become much smoother. However, the overall accuracy starts to decline. So, different weights should be tested in the range of 0.1 to 0.01 to get the best results.

Answer for Question 4:

The simple feed-forward neural network was implemented from scratch in *feed_forward_from_scratch.py.* We have been able to implement forward propagation, backpropagation and calculate the steps needed for weight optimization. We have been using the NumPy package. We have simulated our network with the one we created for question 1. Overall, we are receiving slightly less accuracy, however, our predictions that we are making, in general, are about 60% accurate. The reason behind the difference in accuracy can be a lack of bias (we did not use it for question 4). We have faced many issues. However, during our transformation, sometimes our model fails

because loss returns nan values. The probable reason behind this is that because of the bad initialization we receive very small numbers during our backpropagation, thus NumPy converts them to NaN (this hypothesis needs to be further verified). Another problem we had was that the output layer had extremely small values. If we feed these values to softmax, the softmax function would output almost equal probabilities for each node. The reason is that the exponential function will always output 1 since the value from the output layer is extremely close to 0. However, those very close probabilities are still enough to get decent accuracy. It is because a very small difference in probabilities, (the correct class has a slightly higher probability) is enough to make a correct classification. Thus our overall accuracy makes sense. However, the problem with getting NaN for some random initialization needs to be further investigated.