Homework 4
600.482/682 Deep Learning
Fall 2020

9 October 2020

Due Fri. 10/09 11:59pm.
Please submit a zip containing a report (LaTeX generated PDF), 1 python Jupyter Notebook (one for 1a), and the rest of the code (jupyter notebook or script) to Gradescope with entry code M63JEZ

1. We learnt that a two-layer MLP can model polygonal decision boundaries. You are given data sampled from a "two-pentagons" decision boundary. Data points within the pentagons are considered "true" (labeled as 1) while data points outside are considered "false" (labeled as 0). The data can be downloaded here: https://piazza.com/class_profile/get_resource/kcyxkhb0nuj7d1/kfie17c5kbz3jz.

   Please load the data via *np.load()* function and check that you obtain a numpy array with a shape of $(60000, 3)$. The first two colomns are x, y coordinates and the third one represent the labels.

   The vertices of the 2 polygons are (500, 1000), (300, 800), (400, 600), (600, 600), (700, 800) and (500, 600), (100, 400), (300, 200), (700, 200), (900, 400). We have provided some visualisation code for you in the ipython notebook. The decision boundary can be modeled with a total of $2 \times 5 + 2 + 1$ neurons with threshold activation function.

   (a) We have set up an MLP with threshold activation by analytically obtaining weights that can model the above decision boundary. Implement the "AND gate" and "OR gate" to predict all points within the 2 polygons as the positive class. Next, modify the code to predict only the points in polygon 1 as the positive class. Attach the visualisations to your report. In your report, visualize the prediction result for both cases.

   Note: We reverse-engineered the weights from the known decision boundaries but in the real world, we do not know the decision boundaries. In a more realistic scenario, we would thus need to discover (learn) the decision boundaries from training data which we will explore next.

   (b) Build an MLP using sigmoid activation functions replacing the "AND/OR" gates. You do not need to do the remaining sections in a jupyter notebook and can work with your preferred editor. You may use any Deep Learning package, although **we strongly recommend Google Colaboratory and PyTorch** for ease of debugging and modeling flexibility. Whenever we provide code in subsequent assignments, it will be optimized for this setup. [1]

   Model hyperparameters:

   i. Use 2 hidden layers, and 1 softmax output layer with cross-entropy loss
   ii. Use 10 nodes in the first hidden layer, 2 nodes in the second hidden layer, and 1 output node (binary classification), which mirrors the capacity of the model in 1a)
   iii. Initialize the weights using Xavier Initialization
   iv. Use a learning rate of 0.0001

[1] https://pytorch.org/. If you have not programmed with Deep Learning packages before, give yourself some time to go through some online tutorials. There are many on the internet so its hard to recommend a specific one, it would really depend on your level of comfort.

v. Do not use any regularisation for this problem

Train this model for at least 500 epochs and use gradient descent to optimize the parameters. You should expect an accuracy of above 90%. **Please use the first** 50000 **points as your training set and the remaining** 10000 **as your test set. You may also want to normalize the data before training.**

Visualize your test output predictions. Train your model using 5 random seeds and report the mean and standard deviation of the train and test accuracy at the end of 500 epochs. Are you able to find a solution that performs almost as good as the "manual" solution in (a)? In your report, please attach the visualization of the prediction for the test set; the training loss and testing accuracy change over epochs.

Solution: I created an MLP with the architecture shown below. The model obtained more than 99% accuracy in all runs, so it was a good approximation to the manual solution. I ran the model with random seeds that were a multiple of 5, between 0 and 20 inclusive. For each run, the test accuracy was: [0.9928, 0.9912, 0.9926, 0.9939, 0.9938], which has a mean of 0.99286 and std of 0.00098. The train accuracy was: [0.99378, 0.99398, 0.9941, 0.99464, 0.99498], with mean of 0.994296 and std of 0.00045.

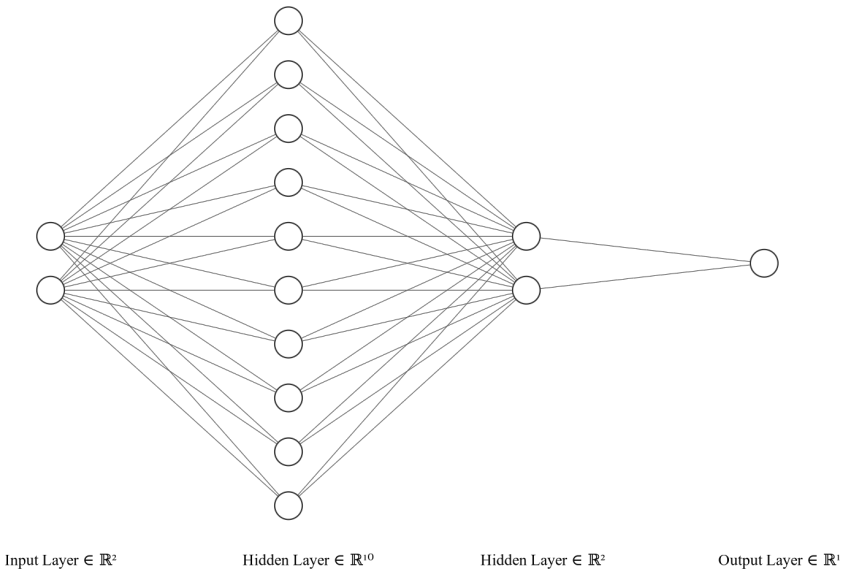

Figure 1: MLP architecture for Q1 part b. I used sigmoid activation functions after the hidden layer with 10 nodes and the one with 2 nodes.

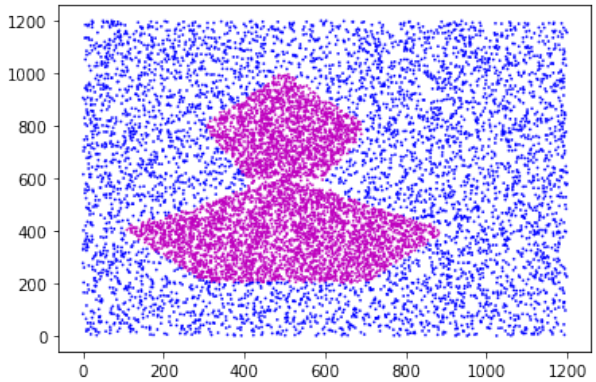

Figure 2: Visualization of the data points after training

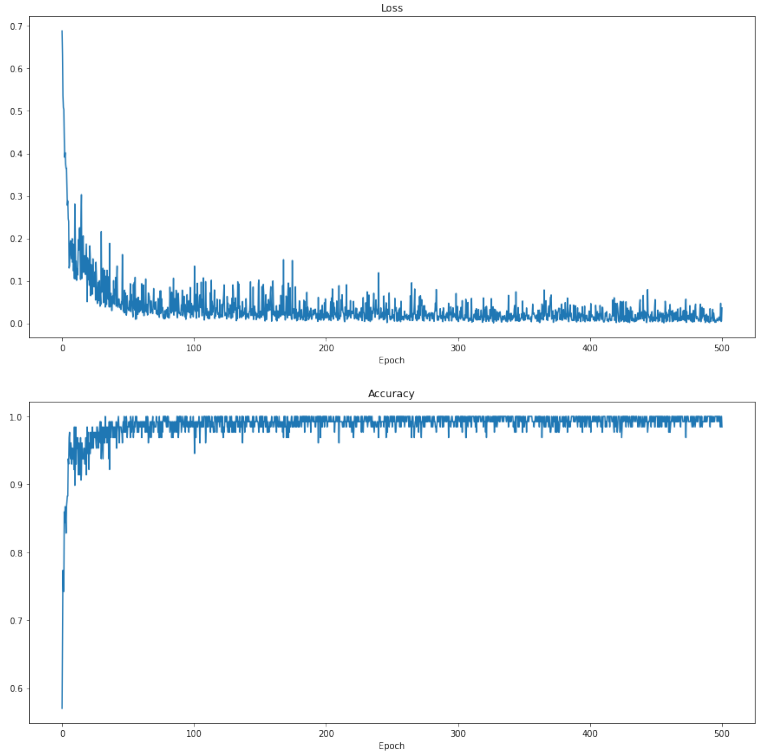

Figure 3: Loss and Accuracy per epoch of training

(c) Build an MLP with a larger capacity (increase the depth and width) and see whether you can achieve higher classification accuracy. Report what depth and width you used, and the mean and standard deviation of the train and test accuracy at the end of 500 epochs. In your report, please attach the visualization of the prediction for the test set; the training loss and testing accuracy change over epochs.

Solution: I used 4 hidden layers this time. Hidden layers of size 10, 15, 10, and 2. I used random seeds between 0 and 20. The train accuracy of each seed was: [0.99404, 0.99552, 0.99408, 0.99624, 0.99346] with mean 0.0.9946 and std of 0.0010. The test accuracy of each seed was [0.9906, 0.99552, 0.9926, 0.9926, 0.995] the mean is 0.99326 and the std is 0.0018. The accuracy wasn't improved over the initial version.

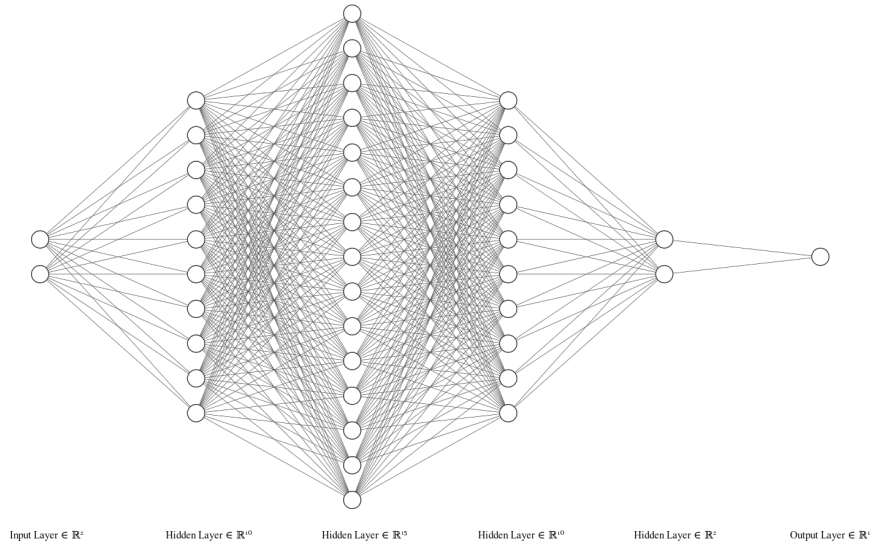

Figure 4: MLP architecture for Q1 part c. I used sigmoid activation functions after the hidden layers.

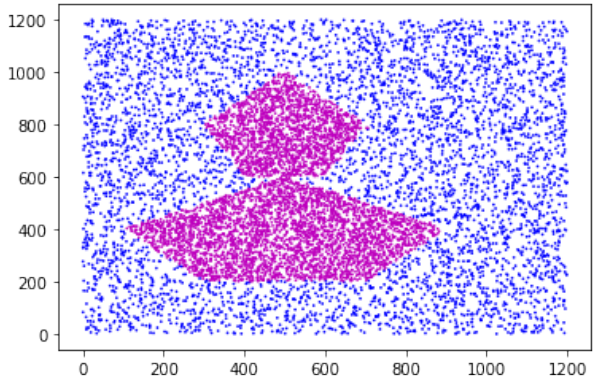

Figure 5: Visualization of the data points after training

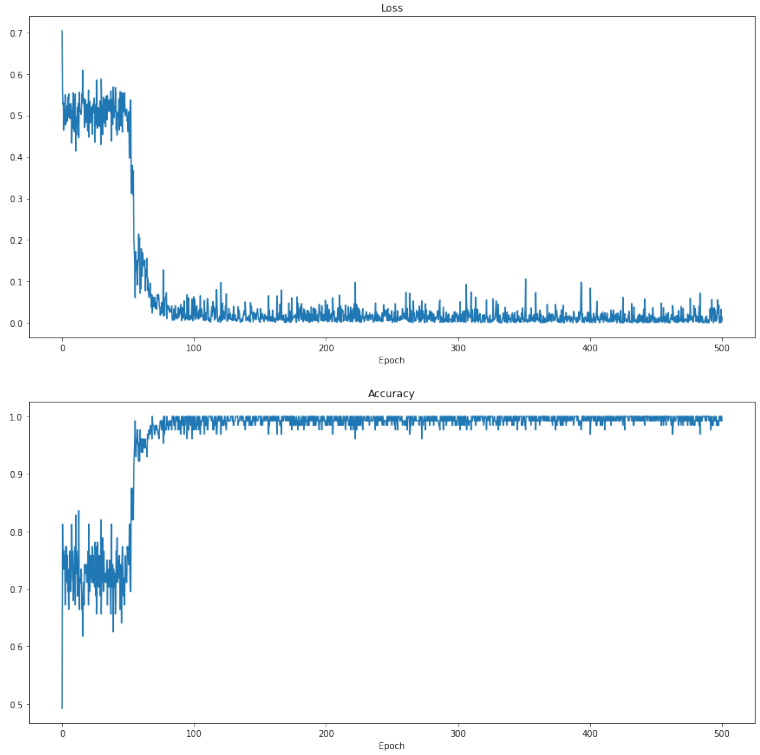

Figure 6: Loss and Accuracy per epoch of training

(d) In part (b) we had fixed the hyperparameter choices, but in part (c) you adjusted those choices for the MLP in terms of depth and width of the network. Would you consider your test results to be valid and generalizable? Please briefly explain.

Solution: The model trained best when the initial hyper parameters. Although it eventually obtained a good accuracy for the deeper and wider network, it took a lot longer to reach that accuracy. We can see from the plots accuracy stalled for a bit for the larger model before reaching peak accuracy, whereas it reached peak accuracy almost immediately for the smaller model. This is most likely because of vanishing gradients, since we used sigmoids rather than RELU activations. Had I built an even bigger model, it probably wouldn't of even trained at all. This means the architecture cannot generalize to other more complicated data that requires more layers.

2. In this problem, we will explore convolutional neural networks (CNNs). We will be working with the FashionMNIST dataset. [2] This dataset only has a train-test split, therefore we will be using the last 10000 training instances as the validation set. Again, we strongly recommend that you use Google Colaboratory and PyTorch[3] although other packages are probably fine as well.

   (a) Design a small CNN (e.g. 3-4 hidden layers) using the tools we learnt in class such as

      i. convolution and pooling layers

      ii. activation functions (other than sigmoid)

   Briefly describe your architecture in your report and explain your design choices (e.g. I used ReLU activation because...) **Please design the network architecture by yourself. This is not about performance.**

   Report the train, validation, and test accuracy of your best model (if the best model was obtained at epoch 50, report the train validation test accuracy at epoch 50). Visualize your training loss, validation and testing accuracy w.r.t the training epoch and attach this plot to your report. For reference, if you achieve more than 85% accuracy on the test set, move on to the next subproblem.

   Solution: I used the architecture shown below. It has two convolution layers and two fully connected layers. I used 4 layers to extract more features. Both the convolution layers had filters of size 3x3. I could not use any bigger filters because otherwise, the output would be too small. I didn't want to pad, because the images were only 28x28, so I didn't want to add noise. After each convolution, I added a max pool of size 2x2. Again, I couldn't go any bigger because the output would be too small. I used RELU activations after each hidden layer because it was the simplest non linearity that wasn't a sigmoid, which would suffer from vanishing gradients. It's simplicity makes the network train faster. The best model had 30 3x3 convolutions, followed by a 2x2 max pool, followed by 20 3x3 convolutions, followed by another 2x2 max pool, followed by a fully connected layer of size 80, and finally the output layer of size 10. I used a cross entropy loss. For training, I normalized all the data, and used batch size 128. I used Adam with learning rate 0.005. The learning rate was empirically chosen, but I chose Adam because it generally converges a lot faster than SGD. I trained for 20 epochs, at which point the network started to over fit.

   I tested the model with 5 random seeds that were multiples of 5 between 0 and 20. This list records the test accuracy for each random seed: [79.720001, 81.699997, 82.189995, 89.720001, 63.570000]. Notice that they cluster around 80% but there is a lot of variation, especially with the last two random seeds. The best model was the random seed of 15 and had train accuracy of 96.389999%, validation accuracy of 90.519997%, and test accuracy of 89.720001%. The sensitivity to initial configuration along with the train accuracy being so much more than the test and validation accuracy is evidence of over fitting. This can be fixed with regularization. **Note: I didn't include test accuracy plots because you should only evaluate on the test set after a model is trained. Technically, you should only evaluate it only with the best model, but I will compromise and evaluate it for all the models.**

[2] The full FashionMNIST dataset can be downloaded from the official website here: https://github.com/zalandoresearch/fashion-mnist. Or you may use the PyTorch API for loading the FashionMNIST dataset: https://pytorch.org/docs/stable/torchvision/datasets.html#fashion-mnist.

[3] https://pytorch.org/get-started/locally/

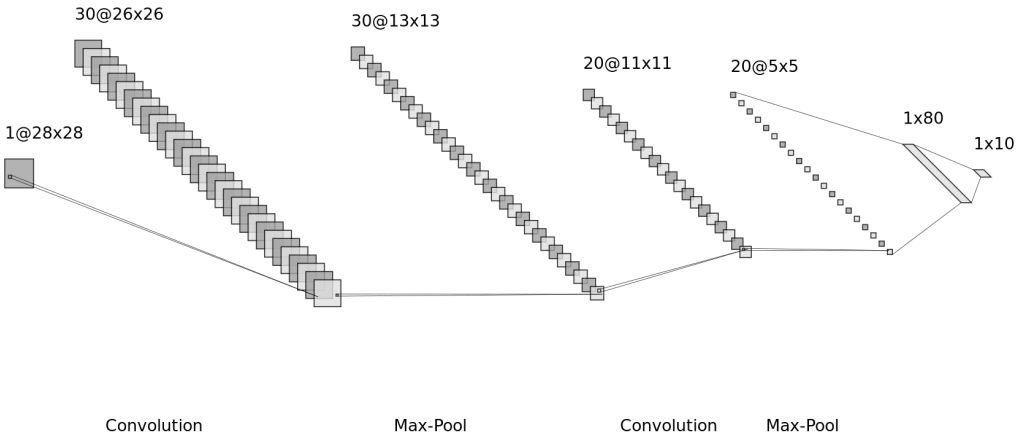

Figure 7: Architecture for Q2 a

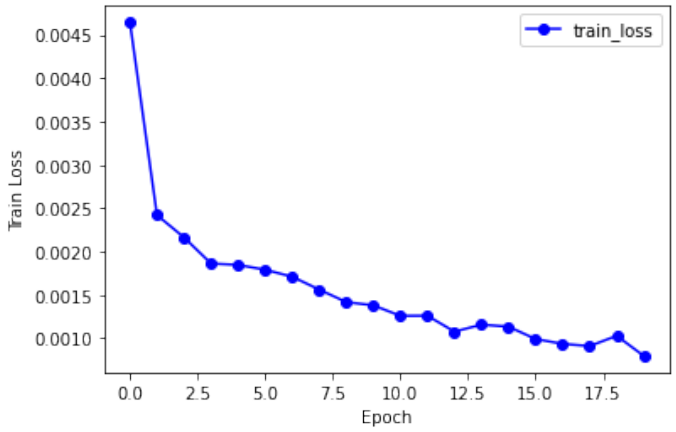

Figure 8: Training loss per epoch of training. Random seed = 15

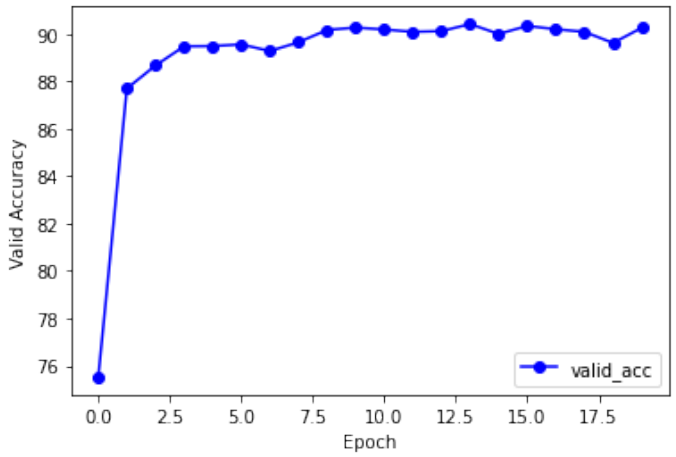

Figure 9: Validation accuracy per epoch of training. Random Seed = 15

(b) Now, design an improved architecture and try to improve on your model's performance.

Implement at least two of the following:

i. dropout
ii. batch normalization
iii. data augmentation
iv. different optimizers (other than vanilla stochastic gradient descent)

Try to improve your accuracy over the model in 2(a). Report the train, validation, test accuracy of your best model. Visualize your training loss and testing accuracy w.r.t the training epoch and attach this plot to your report.

Solution: I implemented dropout and batch norm. I added two batch norm layers. One was after the second convolution layer and before the max pool. The second was after the first fully connected layer. I added a dropout layer after the conversion of the 2D features into a 1D vector of size 500.

Like last time, I tested the model with 5 random seeds that were multiples of 5 between 0 and 20. This list records the test accuracy for each random seed: [89.779999, 90.229996, 89.900002, 89.729996, 80.099998]. The best model was the random seed of 5 and had train accuracy of 94.089996%, validation accuracy of 90.369995%, and test accuracy of 90.229996%. We can see that the model still performs relatively poorly when seeded with 20, but much better than it did initially. There is some evidence of overfitting since the train accuracy is higher than the test and validation accuracy, but that is probably due to training too long. Overall, using batch norm and drop out make the model perform consistently better, and reduce variation in accuracy.

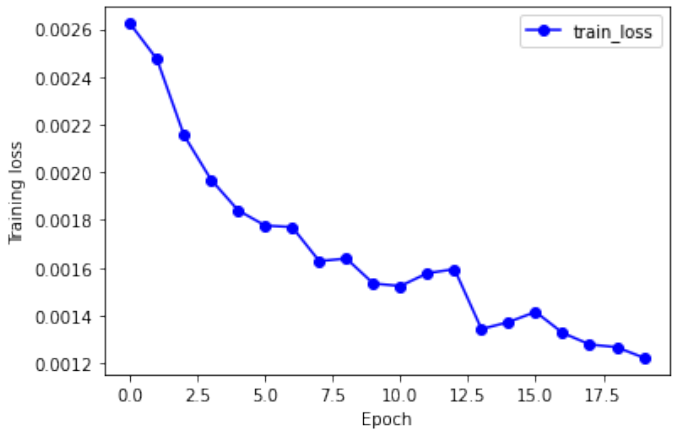

Figure 10: Training loss per epoch of training. Random seed = 5

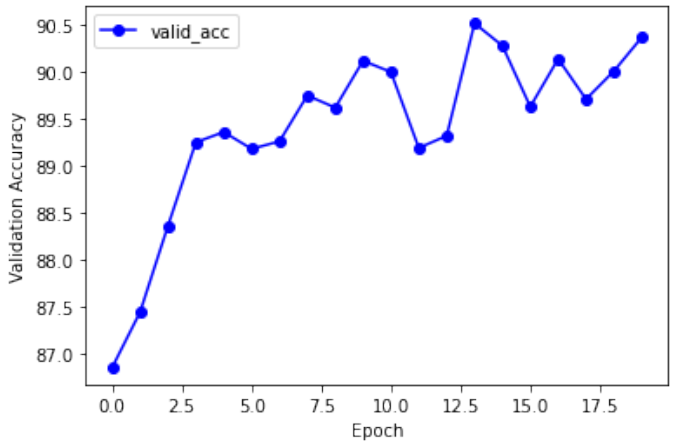

Figure 11: Validation accuracy per epoch of training. Random Seed = 5

(c) Search the internet for models that others have built on FashionMNIST. This can be from research papers or even something like Kaggle. Briefly describe one technique or architecture difference which you found interesting that was not yet covered in class (remember to cite your sources). It can be an extension of an existing method, and you don't need to cover the paper's results for that method. As a rough guide, anything from approximately 50-100 words of prose is sufficient.

Solution: I found an architecture here: https://www.pyimagesearch.com/2019/02/11/fashion-mnist-with-keras-and-deep-learning/. They use a model called Mini VGG-Net. The interesting thing they did was use multiple convolution layers and activations before pooling. My architecture convolution layers (disregarding regularization like batch norm and dropout) was essentially convolution, max-pool, RELU, convolution, max-pool, RELU. Theirs is convolution, RELU, convolution, RELU, max-pool. So not only were there two convolutions before the max-pooling, but the RELU was directly after the convolution. In mine, the max-pool is before the RELU.