Multilayer perceptron

For the multilayer perceptron task we did our own implementation based on the online update mode presented in the lecture slides. We chose this method over batch as it was easier to implement and has one less hyper-parameter to optimize while it shows the same accuracy performance. For the activation function we used the sigmoid function and for the loss we used the mean square error. Due to problem of sigmoid saturation we normalized the data to 0 to 1 scale instead of 0 to 255. For weight initialization Xavier initialization was used to keep weight variance consistency, break symmetry and avoid them blowing up or vanishing, biases were initialized to 0.[1,2] Optimal performance was obtained by starting with a learning rate of 0.1 and dropping it by half every 10 epochs. We repeated random initialization several times. Best results for the normal data can be seen bellow.

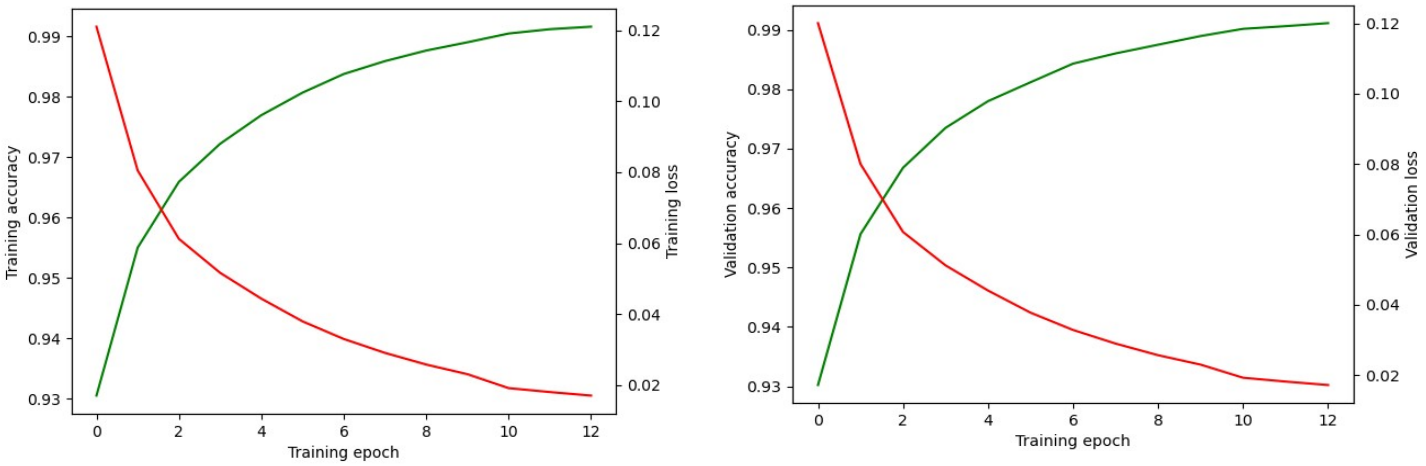

Fig 1. Accuracy and loss graph for the training and validation set

These graphs are completely identical which is surprising but we realized that we used the given csv file for the training set, while we generated the validation set from the pngs which we assume is a subset of the csv train set and thus would be expected to preform the same way as the set it was taken from. On the test data accuracy achieved is: 97.93%

We repeated the process on the permuted set and best results can be seen bellow.

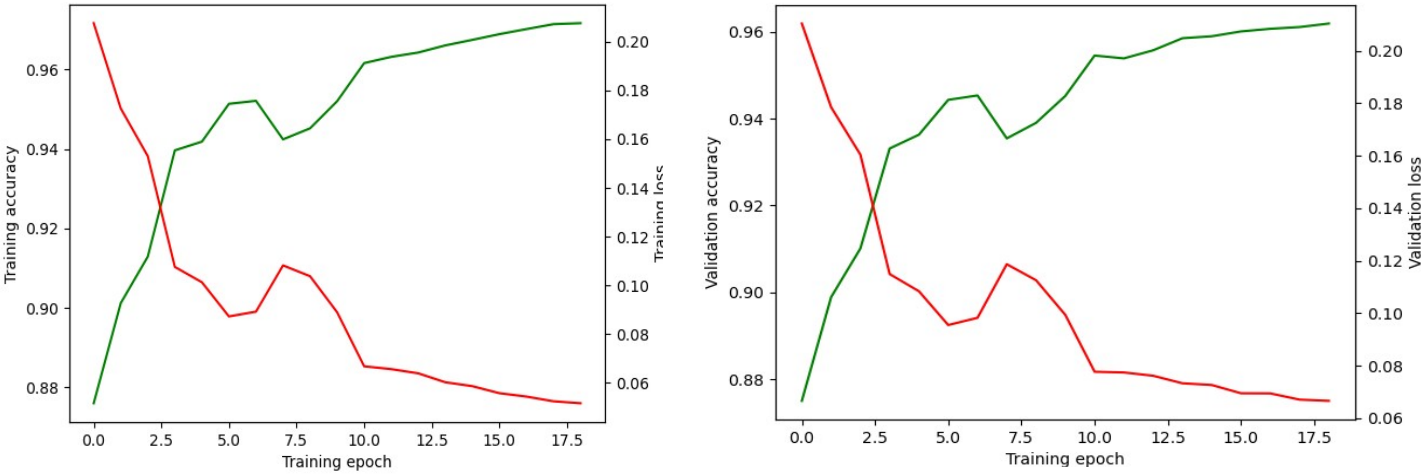

Fig 2. Accuracy and loss graph for the training and validation set of the permuted data

The graphs show same trends, while the validation data has slightly less accuracy, which overall demonstrates a satisfactory and consistent performance. On the test data accuracy achieved is: 96.71%.

We can see that the permuted and normal data-set show approximately the same performance, while the small difference can be attributed to using csv trainset on the normal data which turned out to be a larger set. This is to be expected as the main underlying mechanism of the MLP is computing dot-products which is invariant to change in the order within the data vectors, as long as it is done in the same way for both sets which here is the case. Changes in the curves shapes can be attributed to the fact that we permuted the train-set order every trial which means that samples were visited in different order and since we did an online update this is to be expected.

Convolutional neural network

We implemented the convolution neural network using the pytorch framework. After several smaller scale trials we concluded that the architecture presented in the code gives the best results given the constraints. We used SGD again as the optimizer so that the performance can be somewhat comparable to the MLP. Results for the normal dataset are shown below.

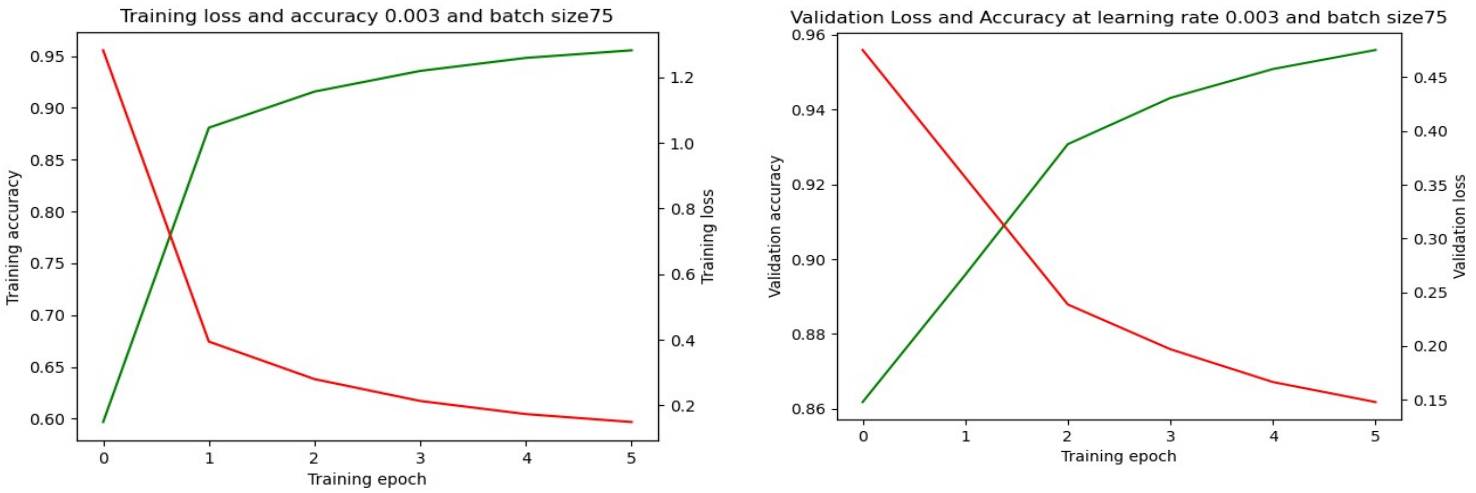

Fig 3. Accuracy and loss graph for the training and validation set

In short similar trends are again observed with the validation and the train sets, with the main difference being the steepness of the curves. Final test accuracy is: 96.06%.

We repeated the process on the permuted set and best results can be seen bellow.

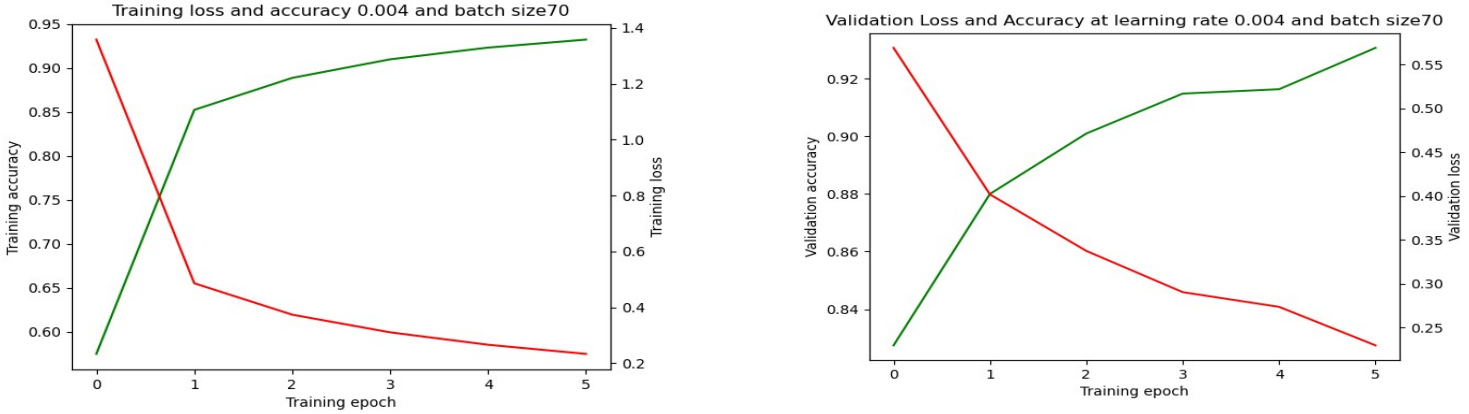

Fig 2. Accuracy and loss graph for the training and validation set of the permuted data

Graphs yet again show similarities with the only difference being steepness of change. Final test acccuracy is: 0.9394.

Compared to the non-permuted dataset CNN has a slight drop in performance. This is expected as CNN unlike MLP exploit spatial relationship which in this case is disrupted by permutation. The reason the performance drop is so small is that these pictures are not complex and are relatively small in size, and a higher performance drop would be expected if we were to test it on more complex images while the MLP should stay consistent.

References

1.https://towardsdatascience.com/weight-initialization-in-neural-networks-a-journey-from-the-basics-to-kaiming-954fb9b47c79
2. https://cs231n.github.io/optimization-1/#gd

Results for the SVM assignement

For a cross-validation done by `GridSearchCV` from `sklearn`, the following accuracies were got:

`params`	mean accuracy (`mean_test_score`)
{'C': 0.001, 'gamma': 0.1, 'kernel': 'linear'}	0.9153
{'C': 0.001, 'gamma': 0.1, 'kernel': 'poly'}	0.9494
{'C': 0.001, 'gamma': 0.1, 'kernel': 'rbf'}	0.1135
{'C': 0.001, 'gamma': 0.01, 'kernel': 'linear'}	0.9153
{'C': 0.001, 'gamma': 0.01, 'kernel': 'poly'}	0.9494
{'C': 0.001, 'gamma': 0.01, 'kernel': 'rbf'}	0.1135
{'C': 0.001, 'gamma': 0.0001, 'kernel': 'linear'}	0.9153
{'C': 0.001, 'gamma': 0.0001, 'kernel': 'poly'}	0.9494
{'C': 0.001, 'gamma': 0.0001, 'kernel': 'rbf'}	0.1135
{'C': 0.001, 'gamma': 1e-06, 'kernel': 'linear'}	0.9153
{'C': 0.001, 'gamma': 1e-06, 'kernel': 'poly'}	0.9007999999999999
{'C': 0.001, 'gamma': 1e-06, 'kernel': 'rbf'}	0.1135
{'C': 0.001, 'gamma': 1e-07, 'kernel': 'linear'}	0.9153
{'C': 0.001, 'gamma': 1e-07, 'kernel': 'poly'}	0.1135
{'C': 0.001, 'gamma': 1e-07, 'kernel': 'rbf'}	0.1135
{'C': 0.001, 'gamma': 1e-08, 'kernel': 'linear'}	0.9153
{'C': 0.001, 'gamma': 1e-08, 'kernel': 'poly'}	0.1135
{'C': 0.001, 'gamma': 1e-08, 'kernel': 'rbf'}	0.1135
{'C': 0.1, 'gamma': 0.1, 'kernel': 'linear'}	0.9153
{'C': 0.1, 'gamma': 0.1, 'kernel': 'poly'}	0.9494
{'C': 0.1, 'gamma': 0.1, 'kernel': 'rbf'}	0.1135
{'C': 0.1, 'gamma': 0.01, 'kernel': 'linear'}	0.9153
{'C': 0.1, 'gamma': 0.01, 'kernel': 'poly'}	0.9494

params	mean accuracy (mean_test_score)
{'C': 0.1, 'gamma': 0.01, 'kernel': 'rbf'}	0.1135
{'C': 0.1, 'gamma': 0.0001, 'kernel': 'linear'}	0.9153
{'C': 0.1, 'gamma': 0.0001, 'kernel': 'poly'}	0.9494
{'C': 0.1, 'gamma': 0.0001, 'kernel': 'rbf'}	0.1135
{'C': 0.1, 'gamma': 1e-06, 'kernel': 'linear'}	0.9153
{'C': 0.1, 'gamma': 1e-06, 'kernel': 'poly'}	0.9504999999999999
{'C': 0.1, 'gamma': 1e-06, 'kernel': 'rbf'}	0.6777
{'C': 0.1, 'gamma': 1e-07, 'kernel': 'linear'}	0.9153
{'C': 0.1, 'gamma': 1e-07, 'kernel': 'poly'}	0.7070000000000001
{'C': 0.1, 'gamma': 1e-07, 'kernel': 'rbf'}	0.9097
{'C': 0.1, 'gamma': 1e-08, 'kernel': 'linear'}	0.9153
{'C': 0.1, 'gamma': 1e-08, 'kernel': 'poly'}	0.1135
{'C': 0.1, 'gamma': 1e-08, 'kernel': 'rbf'}	0.7877
{'C': 1, 'gamma': 0.1, 'kernel': 'linear'}	0.9153
{'C': 1, 'gamma': 0.1, 'kernel': 'poly'}	0.9494
{'C': 1, 'gamma': 0.1, 'kernel': 'rbf'}	0.1135
{'C': 1, 'gamma': 0.01, 'kernel': 'linear'}	0.9153
{'C': 1, 'gamma': 0.01, 'kernel': 'poly'}	0.9494
{'C': 1, 'gamma': 0.01, 'kernel': 'rbf'}	0.1135
{'C': 1, 'gamma': 0.0001, 'kernel': 'linear'}	0.9153
{'C': 1, 'gamma': 0.0001, 'kernel': 'poly'}	0.9494
{'C': 1, 'gamma': 0.0001, 'kernel': 'rbf'}	0.1135
{'C': 1, 'gamma': 1e-06, 'kernel': 'linear'}	0.9153
{'C': 1, 'gamma': 1e-06, 'kernel': 'poly'}	0.9494
{'C': 1, 'gamma': 1e-06, 'kernel': 'rbf'}	0.9513
{'C': 1, 'gamma': 1e-07, 'kernel': 'linear'}	0.9153
{'C': 1, 'gamma': 1e-07, 'kernel': 'poly'}	0.9007999999999999

`params`	mean accuracy (`mean_test_score`)
{'C': 1, 'gamma': 1e-07, 'kernel': 'rbf'}	0.9048
{'C': 1, 'gamma': 1e-08, 'kernel': 'linear'}	0.9153
{'C': 1, 'gamma': 1e-08, 'kernel': 'poly'}	0.1135
{'C': 1, 'gamma': 1e-08, 'kernel': 'rbf'}	0.9048
{'C': 10, 'gamma': 0.1, 'kernel': 'linear'}	0.9153
{'C': 10, 'gamma': 0.1, 'kernel': 'poly'}	0.9494
{'C': 10, 'gamma': 0.1, 'kernel': 'rbf'}	0.1135
{'C': 10, 'gamma': 0.01, 'kernel': 'linear'}	0.9153
{'C': 10, 'gamma': 0.01, 'kernel': 'poly'}	0.9494
{'C': 10, 'gamma': 0.01, 'kernel': 'rbf'}	0.1135
{'C': 10, 'gamma': 0.0001, 'kernel': 'linear'}	0.9153
{'C': 10, 'gamma': 0.0001, 'kernel': 'poly'}	0.9494
{'C': 10, 'gamma': 0.0001, 'kernel': 'rbf'}	0.1135
{'C': 10, 'gamma': 1e-06, 'kernel': 'linear'}	0.9153
{'C': 10, 'gamma': 1e-06, 'kernel': 'poly'}	0.9494
{'C': 10, 'gamma': 1e-06, 'kernel': 'rbf'}	0.9522999999999999
{'C': 10, 'gamma': 1e-07, 'kernel': 'linear'}	0.9153
{'C': 10, 'gamma': 1e-07, 'kernel': 'poly'}	0.9477
{'C': 10, 'gamma': 1e-07, 'kernel': 'rbf'}	0.9579000000000001
{'C': 10, 'gamma': 1e-08, 'kernel': 'linear'}	0.9153
{'C': 10, 'gamma': 1e-08, 'kernel': 'poly'}	0.15660000000000002
{'C': 10, 'gamma': 1e-08, 'kernel': 'rbf'}	0.9304
{'C': 1000, 'gamma': 0.1, 'kernel': 'linear'}	0.9153
{'C': 1000, 'gamma': 0.1, 'kernel': 'poly'}	0.9494
{'C': 1000, 'gamma': 0.1, 'kernel': 'rbf'}	0.1135
{'C': 1000, 'gamma': 0.01, 'kernel': 'linear'}	0.9153
{'C': 1000, 'gamma': 0.01, 'kernel': 'poly'}	0.9494

params	mean accuracy (mean_test_score)
)0, 'gamma': 0.01, 'kernel': 'rbf'}	0.1135
{'C': 1000, 'gamma': 0.0001, 'kernel': 'linear'}	0.9153
{'C': 1000, 'gamma': 0.0001, 'kernel': 'poly'}	0.9494
{'C': 1000, 'gamma': 0.0001, 'kernel': 'rbf'}	0.1135
{'C': 1000, 'gamma': 1e-06, 'kernel': 'linear'}	0.9153
{'C': 1000, 'gamma': 1e-06, 'kernel': 'poly'}	0.9494
{'C': 1000, 'gamma': 1e-06, 'kernel': 'rbf'}	0.9522999999999999
{'C': 1000, 'gamma': 1e-07, 'kernel': 'linear'}	0.9153
{'C': 1000, 'gamma': 1e-07, 'kernel': 'poly'}	0.9494
{'C': 1000, 'gamma': 1e-07, 'kernel': 'rbf'}	0.9575999999999999
{'C': 1000, 'gamma': 1e-08, 'kernel': 'linear'}	0.9153
{'C': 1000, 'gamma': 1e-08, 'kernel': 'poly'}	0.9007999999999999
{'C': 1000, 'gamma': 1e-08, 'kernel': 'rbf'}	0.9333

Where the optimized parameter values were:

```
SVC(C=10, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma=1e-07, kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

Accuracy on the test set with the optimized parameter values: 0.9995