# MNIST Digit Classification Report

## Introduction

In this assignment digits from the MNIST digit dataset were classified using deep learning techniques. Various neural networks (NN's) were implemented with varying degree of complexity:

a) 1 linear layer, followed by a softmax
b) 1 hidden layer (128 units) with a ReLU non-linearity, followed by a softmax
c) 2 hidden layers (256 units) each, with ReLU non-linearity, follow by a softmax
d) 3 layer convolutional model (2 convolutional layers followed by max pooling) + 1 non-linear layer (256 units), followed by softmax.

Architectures A-D were implemented using tensor flow in Part 1 of the assignment and without tensor flow, using other python packages, in Part 2 of the assignment. Models were trained in batches to allow for stochastic gradient descent optimisation and were evaluated using the cross-entropy loss function. The report below is a summary of results in terms of test and training errors.

## Key observations

In Part 1, accuracy and convergence improved with the increasing complexity of model; CNNs performed best. However, this was at the expense of computational speed. Results were stable.

In Part 2, models required significantly more epochs to train before producing satisfactory results. Results were also less stable and in part 2D changes to functions (cross entropy forward) had to be made to prevent numerical underflow. Results were sensitive to the learning rate and batch size (used for performing gradient descent). With a clear trade-off between batch-size and speed.

Overall, Higher accuracy was achieved using models implemented with tensor flow, as opposed to those implemented without it (Part2).

## Report

### 1) Training and testing errors.

Below is a summary of training and testing errors every 10 epochs (throughout training).
It should be noted that : *error = 1- accuracy,* and accuracy is reported below, together with cross entropy loss.

Part 1A:
Iteration (epoch) 0: Accuracy 0.897(train) 0.901(test), cross-entropy loss 0.344(train) 0.336(test)
Iteration (epoch) 10: Accuracy 0.918(train) 0.911(test), cross-entropy loss 0.283(train) 0.303(test)
Iteration (epoch) 20: Accuracy 0.921(train) 0.914(test), cross-entropy loss 0.272(train) 0.301(test)
Part 1B:
Iteration (epoch) 0: Accuracy 0.935(train) 0.935(test), cross-entropy loss 0.213(train) 0.209(test)
Iteration (epoch) 10: Accuracy 0.992(train) 0.974(test), cross-entropy loss 0.027(train) 0.082(test)
Part 1C:
Iteration (epoch) 0: Accuracy 0.258(train) 0.256(test), cross-entropy loss 1.934(train) 1.932(test)
Iteration (epoch) 10: Accuracy 0.992(train) 0.976(test), cross-entropy loss 0.025(train) 0.099)test)
Part 1D
Iteration (epoch) 0: Accuracy 0.977(train) 0.977(test), cross-entropy loss 0.073(train) 0.072(test)

In Part 1, as expected, the more sophisticated the model the less iterations are required for solutions to converge to higher accuracy. Model A was trained for 25 epochs, Model B for 20, Model C for 15 and Model D for 10, the were sufficient for solutions to reach accuracy well above 90%. It should be noted however that the total run time on the more complex models exceeded that of the simpler ones even with less iterations.

```
 Part 2B:
 Iteration 0. Accuracy is: 0.876327 (train) and 0.885700 (test)
 Iteration 10. Accuracy is: 0.914345 (train) and 0.918500 (test)
 Iteration 20. Accuracy is: 0.919509 (train) and 0.921200 (test)
 Iteration 30. Accuracy is: 0.922800 (train) and 0.921800 (test)
 (for results after each epoch refer to code output)
```

```
Part 2C:
 Iteration 0. Accuracy is: 0.462309 (train) and 0.482000 (test)
 Iteration 10. Accuracy is: 0.817582 (train) and 0.821800 (test)
 Iteration 20. Accuracy is: 0.877109 (train) and 0.883900 (test)
 Iteration 30. Accuracy is: 0.893618 (train) and 0.898400 (test)
 Iteration 40. Accuracy is: 0.901655 (train) and 0.905000 (test)
 Iteration 49. Accuracy is: 0.907327 (train) and 0.910600 (test)
 (for results after each epoch refer to code output)
```

```
Part 2:D
 Iteration 0. Accuracy is: 0.112345 (train) and 0.113500 (test)
 Iteration 10. Accuracy is: 0.864455 (train) and 0.866700 (test)
 Iteration 20. Accuracy is: 0.910091 (train) and 0.909400 (test)
 Iteration 30. Accuracy is: 0.933727 (train) and 0.933800 (test)
 Iteration 40. Accuracy is: 0.949200 (train) and 0.947900 (test)
 Iteration 49. Accuracy is: 0.958818 (train) and 0.955500 (test)
```

In Part 2 Models require more epochs to train and converge at accuracies below the ones obtained from Part 1. TensorFlow is designed specifically for use with deep learning, tensors are used as data structures and has a powerful optimising compiler. Due to which, for instance, matrix dot products can be found many times faster than in numpy. Furthermore in Part 2 numerical issues of underflow, and possibly other sources of accumulated inaccuracies can be seen. Thus are many reasons for the divergence of results between the two parts. However, results with an accuracy of above 90% are still satisfactory.

## 2) Final training and testing errors table

Training and testing errors (defined as: 1- accuracy) values at the end of the optimisation can be seen in the table below.

| Experiment | P1:A | P1:B | P1:C | P1:D | P2:B | P2:C | P2:D |
|---|---|---|---|---|---|---|---|
| Train Error | 0.078 | 0.002 | 0.005 | 0.002 | 0.076 | 0.093 | 0.041 |
| Test Error | 0.086 | 0.023 | 0.022 | 0.01 | 0.077 | 0.89 | 0.044 |

- Results from Part 1 are generally better and converge faster.
- In most instances, test and train errors are close indicating that model stoped trains before overfitting which is good.
- In Part 2C the training error is marginally larger than test error indicating slight under-fitting.

## 3) Confusion Matrix

This hold be  a 10 by 10 table with true labels along one dimensions and predict labels along the other dimension, For each label combination this would then include the count of correct vs erroneous predictions.
For part 1 Confusion matrix Not Available because of how code was written I have found it hard to convert my model output (Y predicted) from a tensor to an array.  Never the less, code to make a confession matrix from an np.array is included as a function.
Learning my lesson part 2 is implemented differently so it is possible to implement the confusion matrix function and obtain the matrix shown below.

Part 2:B (horizontal axis : predicted labels, vertical axis: true labels)

```
Confusion Matrix:

[[ 962    0    2    2    0    4    7    1    2    0]
 [   0 1109    2    2    0    2    4    2   14    0]
 [   7    7  914   15    9    5   14   12   40    9]
 [   3    1   22  916    0   29    2   12   18    7]
 [   1    2    3    1  911    0   13    2    8   41]
 [  10    3    3   30    9  779   16    6   29    7]
 [  10    3    4    2    9   16  908    3    3    0]
 [   2    6   22    8    6    1    0  946    2   35]
 [   8    6    7   23    9   27   10   10  869    5]
 [  11    6    2    9   29    7    0   24    7  914]]
```

Part 2:C (horizontal axis : predicted labels, vertical axis: true labels)

```
Confusion Matrix:

[[ 958    0    3    2    0    5    9    1    2    0]
 [   0 1110    2    2    0    2    4    1   14    0]
 [  12    8  901   17   14    1   15   16   42    6]
 [   3    0   21  909    0   34    1   15   20    7]
 [   1    1    5    2  912    1   12    2    7   39]
 [  14    3    6   48    9  747   17    9   32    7]
 [  16    3    5    0   17   15  897    1    4    0]
 [   6   10   26    6    8    0    0  939    3   30]
 [   8    9    8   26   10   31   13    9  846   14]
 [  11    8    3   11   47   13    1   24    4  887]]
```

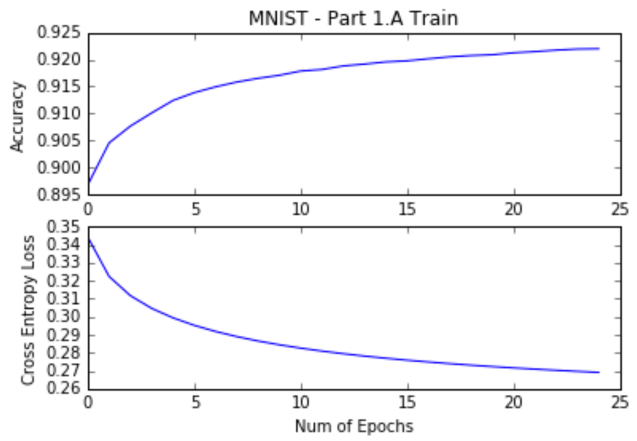
Part 2:D (horizontal axis : predicted labels, vertical axis: true labels)

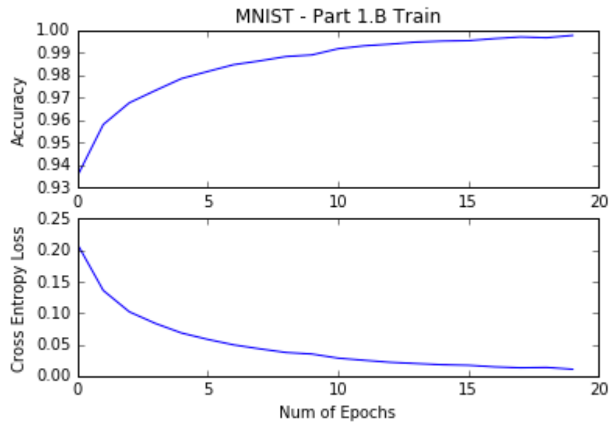
```
Confusion Matrix:

[[ 965    0    1    1    0    6    4    1    1    1]
 [   0 1113    3    2    0    1    3    2   11    0]
 [  10    4  974   12    4    0    9    8    7    4]
 [   0    0   10  971    0    8    0    7    9    5]
 [   1    1    2    0  941    0   11    2    3   21]
 [   6    1    0   10    2  845   10    1   12    5]
 [   7    3    0    0    9   15  920    0    4    0]
 [   3    7   16    5    3    0    0  974    1   19]
 [   4    5    2   12    8   12   10    7  907    7]
 [   9    6    0    7   20    9    0    8    5  945]]
```

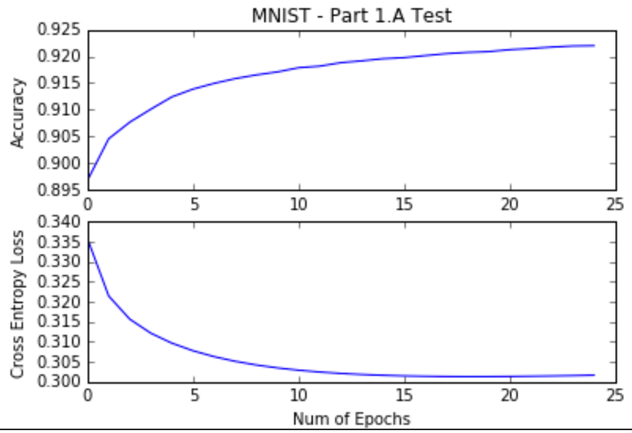
It is interesting to observe the different error types made by the different models. Most correct predictions are of the digit 1, perhaps due to its characteristic shape. Whereas digit 5 appears to be universally the hardest to classify. Overall, the most complex model performs best.

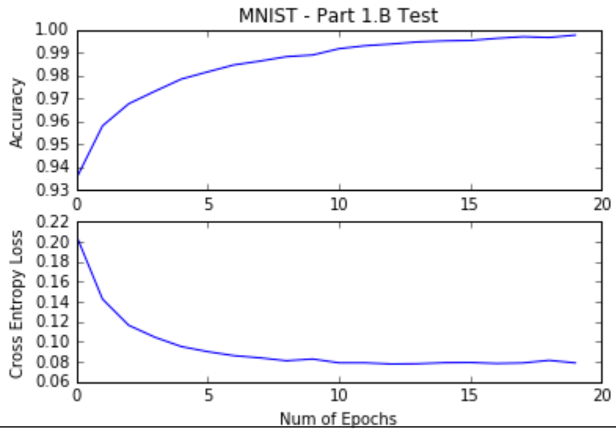
## 4) Plots

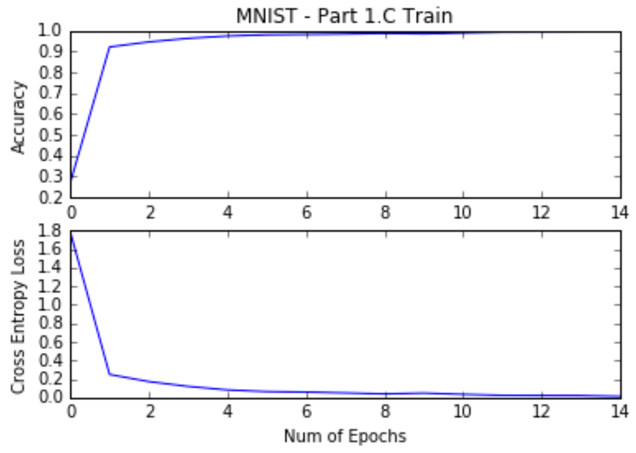
PART 1: Test and Train accuracy as well as cross-entropy loss against epochs.

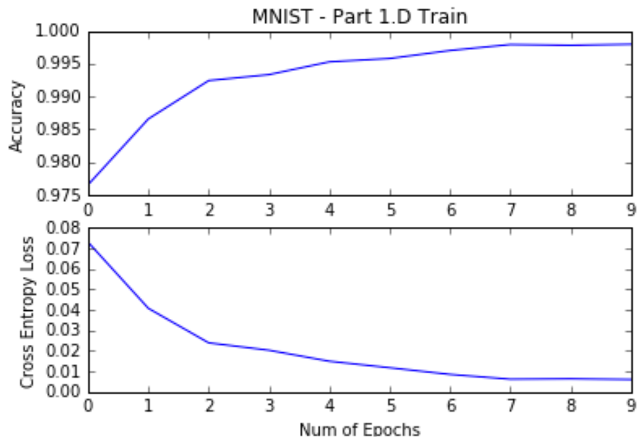


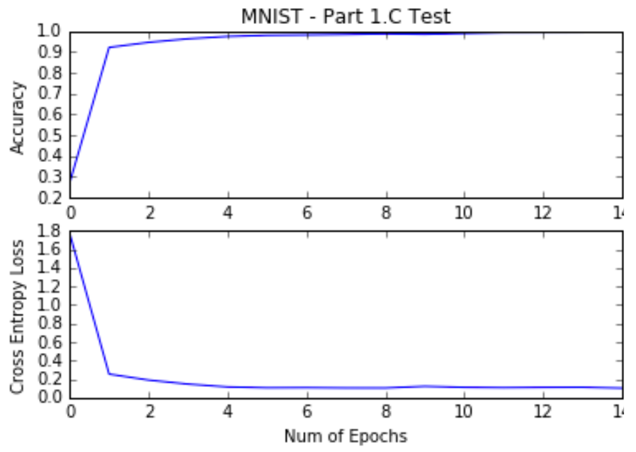


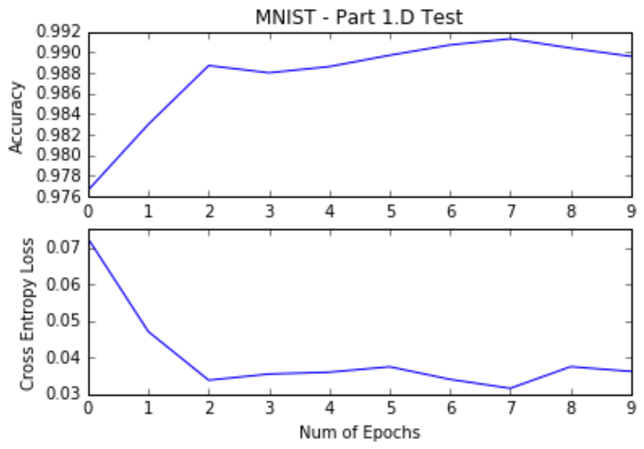






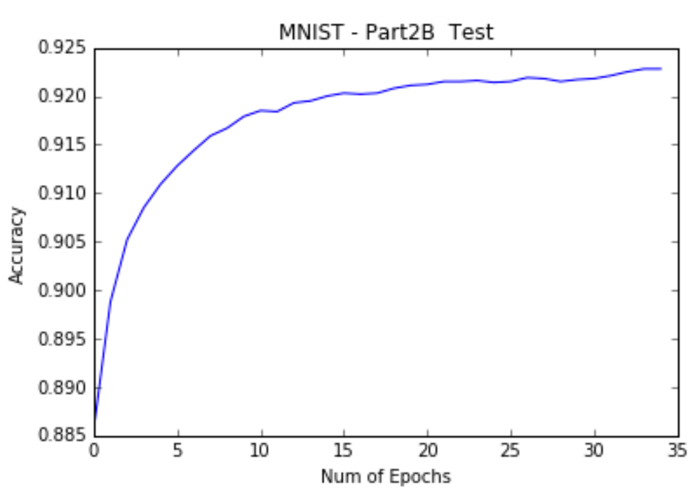


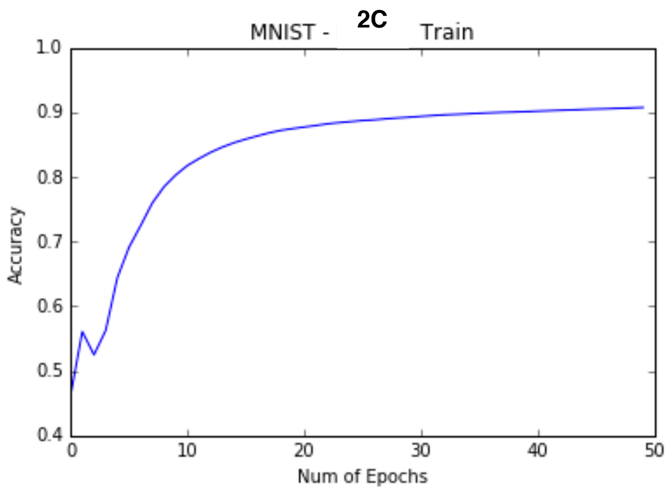


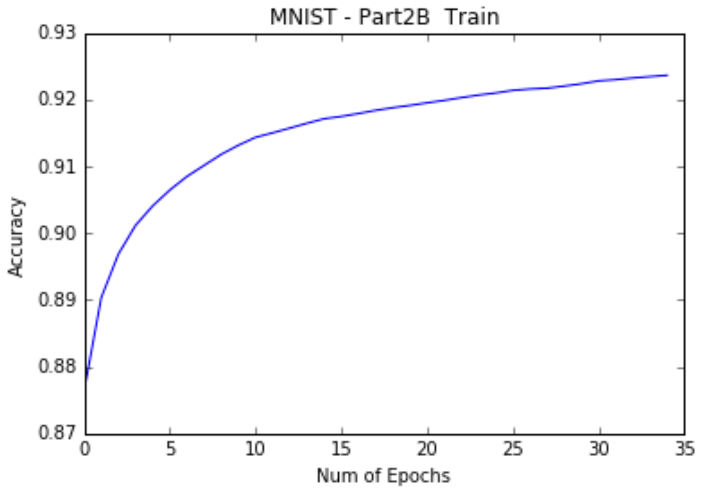


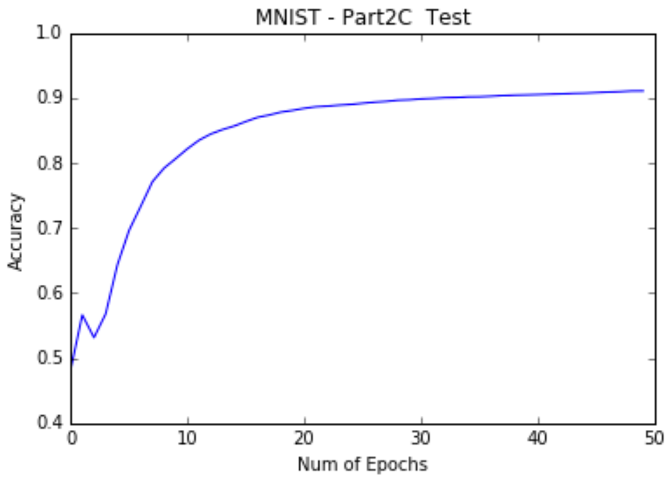
Advanced ML ICA 1 Klaudia Ludwisiak

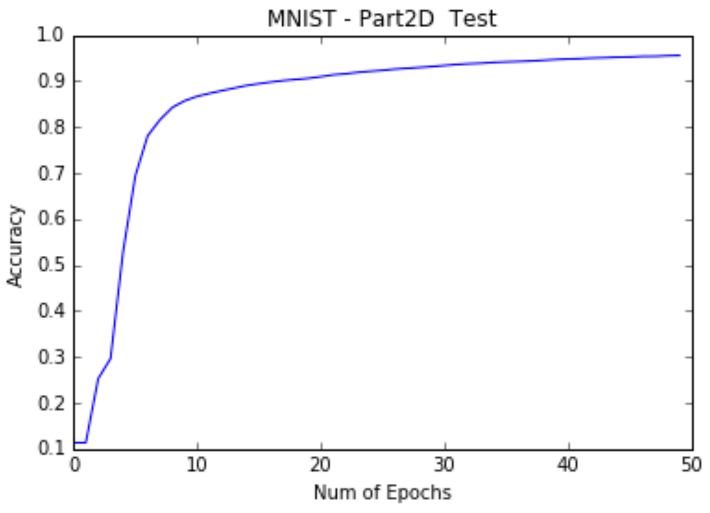
PART 2: Plots of Test and Train accuracy.

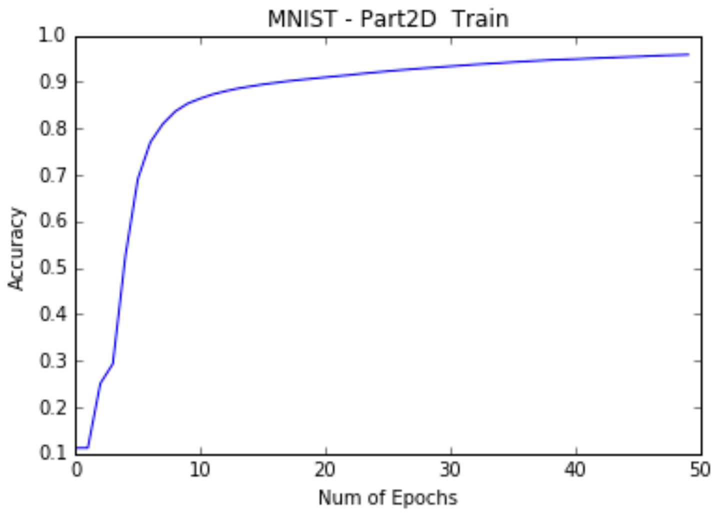












**5) Final weights and bias terms for models**

Again due to the use of tensors and to how the code was written it was hard to obtain final weights for Part 1. Final weights found in part two are printed below.

Part 2:B

```
 After 35 epoch iterations, Final weight array is:

[[ 0.016 -0.006 -0.005 ..., -0.008  0.003 -0.002]
 [ 0.015 -0.021 -0.003 ..., -0.009  0.     0.006]
 [-0.011  0.011  0.009 ..., -0.009 -0.003  0.005]
 ...,
 [-0.016  0.014  0.002 ...,  0.014 -0.009  0.   ]
 [ 0.011 -0.002 -0.01  ...,  0.    -0.014 -0.009]
 [ 0.006 -0.009  0.009 ...,  0.006 -0.015 -0.003]]

 After 35 epoch iterations, Final bias array is:

[-0.468  0.394  0.141 -0.317 -0.036  1.564 -0.131  0.767 -1.648 -0.27 ]
```

Part 2:C

```
 After 50 epoch iterations, Final weight array is:

[[ 0.087 -0.14   0.026 ..., -0.103 -0.064  0.082]
 [ 0.376 -0.444 -0.103 ..., -0.281  0.136  0.17 ]
 [-0.007  0.009  0.01  ..., -0.012 -0.012  0.029]
 ...,
 [-0.054 -0.049  0.017 ...,  0.302 -0.114  0.098]
 [-0.066  0.02   0.081 ..., -0.135  0.092 -0.046]
 [-0.164  0.152 -0.096 ..., -0.033  0.128  0.131]]

 After 50 epoch iterations, Final bias array is:

[-0.175  0.277 -0.041 -0.109  0.036  0.249  0.002  0.132 -0.351 -0.031]
```

Part 2:C

```
After 50 epoch iterations, Final weight array is:

[[ 0.099 -0.141 -0.093 ..., -0.054  0.133  0.033]
 [ 0.079 -0.187 -0.286 ..., -0.34   0.092  0.063]
 [-0.081  0.071  0.101 ...,  0.07  -0.07  -0.1  ]
 ...,
 [-0.015  0.004 -0.021 ..., -0.006  0.012 -0.014]
 [ 0.075 -0.033  0.043 ...,  0.022 -0.075 -0.046]
 [ 0.118 -0.029  0.082 ..., -0.035 -0.034  0.002]]

After 50 epoch iterations, Final bias array is:

[-0.444  0.501 -0.03  -0.072  0.131  0.385 -0.009  0.265 -0.627 -0.09 ]
```