Project 3: Backpropagation Neural Network (BPNN)

Owen Queen, COSC 522

Please see the Jupyter notebook file *run_project3.ipynb* for the implementation of the code for this project.

Task 1

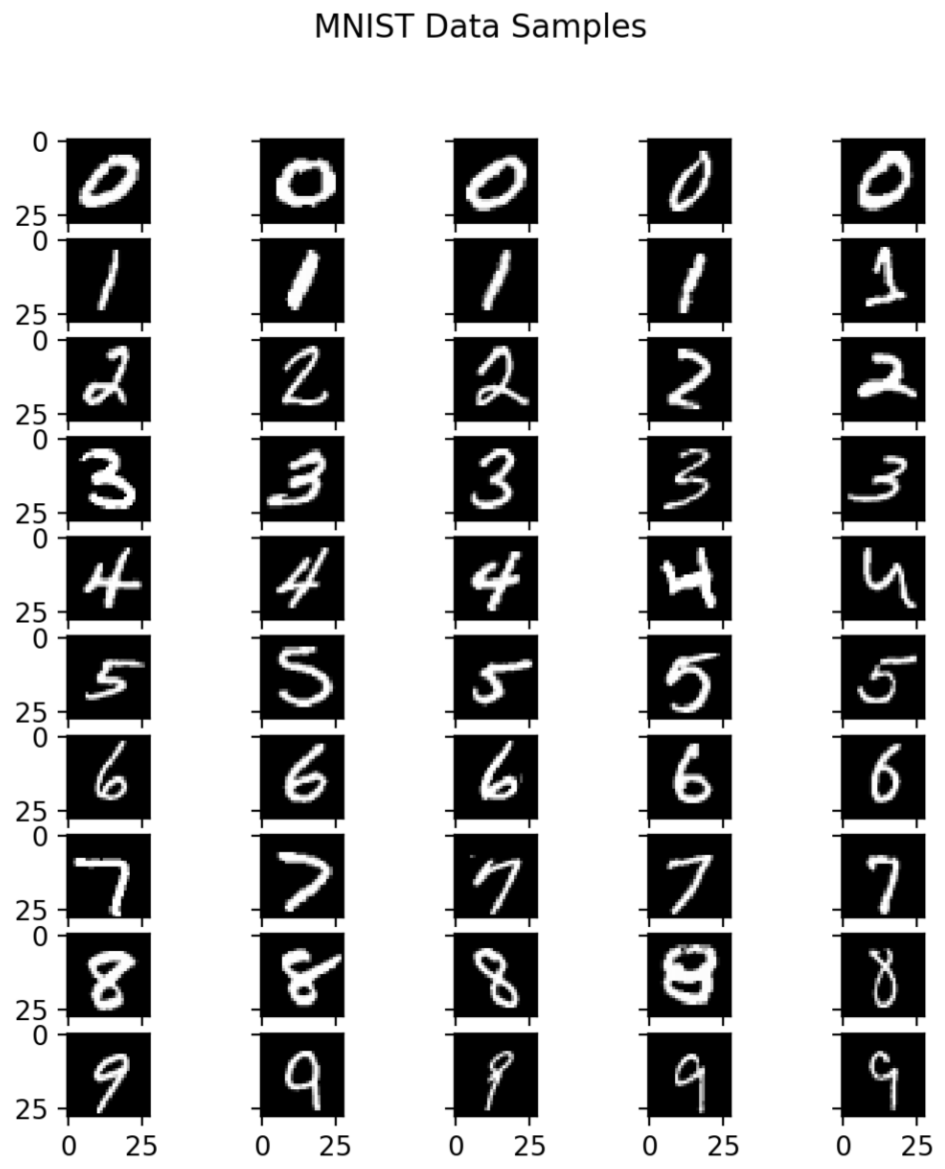
1. "In your report, state the reason why Nielsen uses the one hot label (i.e., the 10-dimensional unit vector) for the training labels but preserves the validation and test labels as integers."

   Nielsen uses this convention because this is how the output layer of the neural network will give the labels. The raw output from the output layer of the neural network will be continuous values in an n-dimensional vector (where n = number of classes), so we choose the largest value to correspond to the label the network chooses. Therefore, the index with the largest value becomes 1 and all of the other indices become 0. This works for the training data because it lends to an easy comparison at the output layer of the network. However, it makes more sense to use scalars as the output for the testing data in order to directly compare the output of the network to determine the accuracy of the model. Since the polished model's output will be a scalar, the testing data should also be a scalar for an easy comparison in the *evaluate* function.

2. "All three functions, load_data(), load_data_wrapper(), and vectorized_result() are efficiently designed. However, the function also utilized fixed numbers or fixed filenames directly which would largely limit the reusability of the code."

   The appropriate functions have been modified in the file named *mnist_loader.py*.

3. "After calling load_data(), generate a 10x5 subplots where you randomly select 5 images from each class in the training set to display."



Task 2
"Bayesian-based classification. Use load_data() to read in the data."

Table of model results on MNIST dataset:

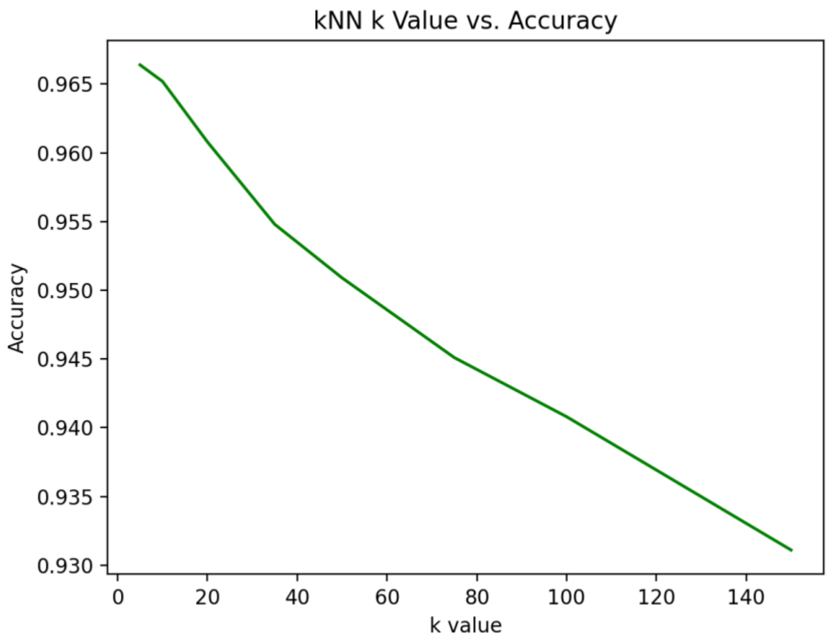
Model	Overall Accuracy	Run Time (sec)
kNN (k = 50)	0.9509	2064.87
MPP Case 1	0.8200	1.09660
kNN (k = 50) + PCA (tolerance = 0.15)	0.9599	227.118
MPP Case 1 + PCA (tolerance = 0.15)	0.8188	0.549371

Bonus

"Comprehensive study of k in kNN and error rate in PCA - Output two figures, overall_accuracy vs. k without PCA, overall_accuracy vs. error rate with k fixed."

kNN:

k value	**Overall Accuracy**	**Run time (seconds)**
5	0.9664	2100.23
10	0.9652	1991.76
20	0.9608	1978.54
35	0.9548	1971.89
50	0.9509	2064.87
75	0.9451	1980.63
100	0.9408	1992.59
150	0.9311	1953.25



Please note that there are some discrepancies in the runtimes for these models, as the CPU load was not controlled. All models were run on my local Mac OS.

From these results, we can conclude that low k values work especially well when classifying these data with kNN. My hypothesis is that this works well because handwritten digits, aside from boundary cases, are very segregated samples. From looking at the figure in Task 1, part 3, it seems as if the digits are always somewhat related to each other, containing the same sorts of general features. Therefore, in a higher dimensional space, these vectors will cluster together very well. Because the data cluster together well, a low k value would make the most sense for classifying. We use increasing k values to get a picture of the neighborhood of a data point, but if a neighborhood contains very few outliers, then we don't need the k value to be too large.

In general, it seems as if heterogenous data – those data sets that are easily separable – would lead to better accuracy with a lower k value. While a low k value would capture a smaller neighborhood, a large k value might tend to misclassify boundary cases, which is undesirable. This is why I believe that we experience the phenomena in the above table.

PCA with kNN (k = 20):

Tolerance	**Error Rate**	**Eigenvectors Kept**	**kNN Accuracy**	**kNN Run Time**
0.01	0.00997	331	0.9616	1169.55
0.025	0.02489	235	0.9624	853.370
0.05	0.04978	154	0.9631	554.690
0.1	0.09984	87	0.9655	316.450
0.15	0.14966	59	0.9673	218.218
0.2	0.19646	44	0.9688	136.847
0.25	0.24419	34	0.9699	114.507
0.3	0.29944	26	0.9673	95.6958
0.4	0.39161	17	0.9597	62.1684
0.5	0.49024	11	0.9347	64.6199
0.6	0.59145	7	0.8779	49.3977
0.7	0.66742	5	0.7662	47.3956

I chose to use k = 20 here because it had a relatively high accuracy in the trials where we varied k. While the lower values did have a higher accuracy, we want to guard from the possibility of overfitting on this model, which tends to happen if the k value is either too high or too low. Therefore, I chose a k value that still had a high accuracy but was not so small as to overfit to training data.

From these trials, we can see that higher dimensions typically lead to higher accuracy and slower run times. However, it is worth noting that even keeping a very low number of dimensions – down to 26 – yields a similar accuracy to those that have much higher dimensions. The upside of using this low dimension is the runtime, which decreased to 1% for the 0.3 tolerance dataset compared to when all eigenvectors were kept. Therefore, when processing data – especially images – it is worth checking whether the dimensions can be reduced because this will make training and testing much easier.

Task 3

1. "On stochastic gradient descent (SGD)."
    a. What is SGD? What is the difference to gradient descent (GD)? Is it an online learning approach or batch learning approach or neither? When would SGD become online learning? When would SGD become batch learning?

       Stochastic gradient descent (SGD) is a modification of the gradient descent algorithm (GD) that improves the speed of computation. The definition of GD requires us to compute the gradient based on the error of each sample's classification on our network; this is very slow when you have large datasets. SGD modifies this method by taking a random sample of the training data and computing the gradient based on that sample (called a mini-batch), which is usually much smaller than the entire training dataset.

SGD is not necessarily an online learning approach. SGD updates the model's internal parameters after every member of its mini-batch has been processed. If SGD had a mini-batch size of 1, then it would be an online learning approach, updating the internal weights and biases after every sample. However, if SGD has a mini-batch size of greater than 1, it is a batch-learning algorithm, updating the internal parameters after a group of samples has been processed.

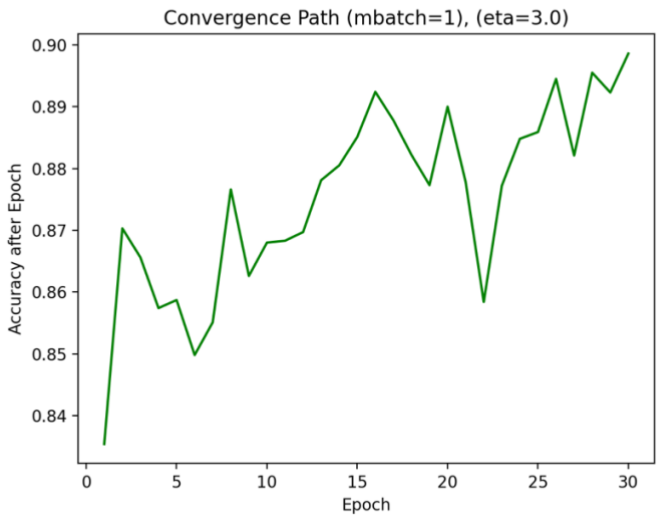
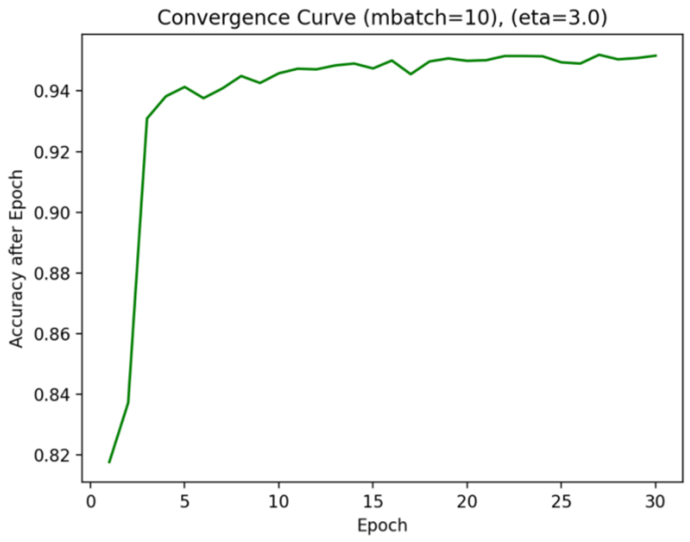
b. "Instead of printing the accuracy at the end of each epoch, modify SGD() such that it also output a figure of accuracy vs. epoch (i.e., the convergence curve)."

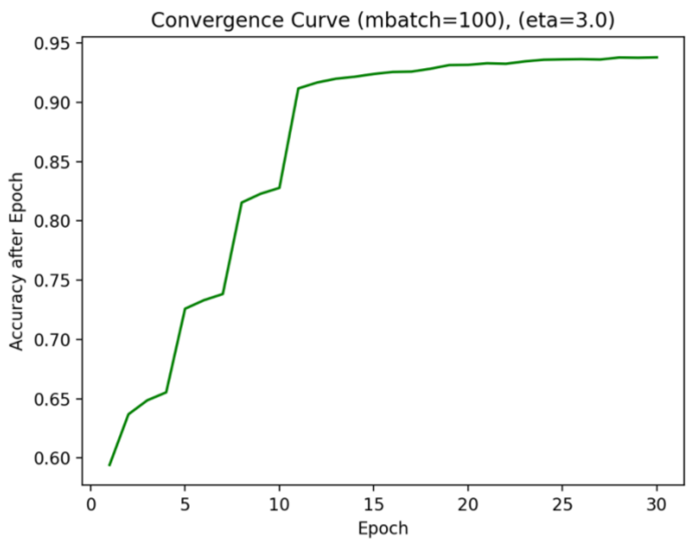
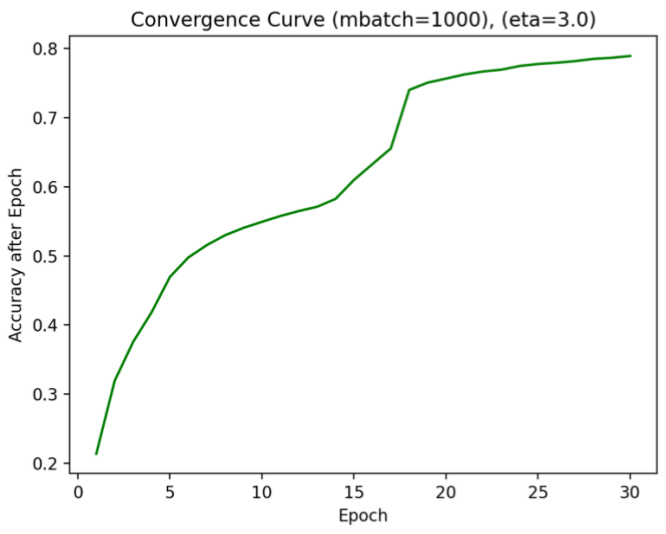
This feature has been implemented in *network_oq.py*.

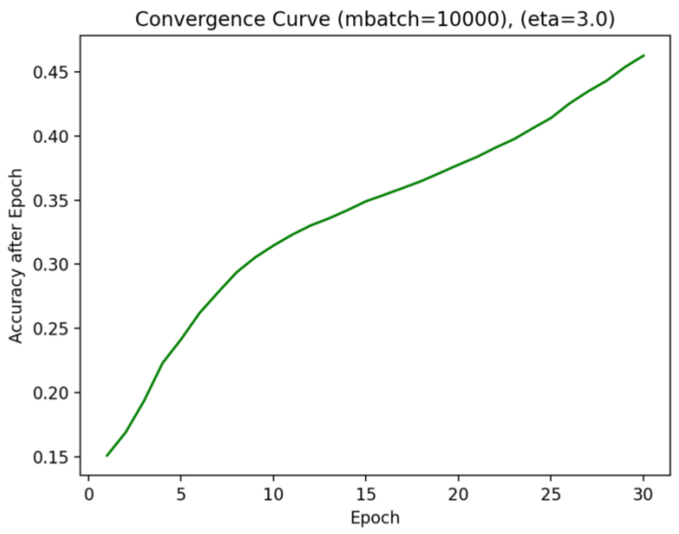
c. "Adjust the mini-batch size from 1, 10, 100, 1,000, to 10,000. Compare the time spent and accuracy at epoch = 30. Plot a bar chart or use a table to present the difference. Also attach the convergence curve for each size. From the performance differences you observed, what can you learn in terms of an appropriate size for the mini-batches?"

Varying mini-batch sizes:

Mini-batch Size	**Acc. at Epoch 30**	**Run Time**	**Convergence Curve**

1	0.88420	216.132	
10	0.9516	167.277	

100	0.9378	170.234	
1,000	0.7889	157.019	

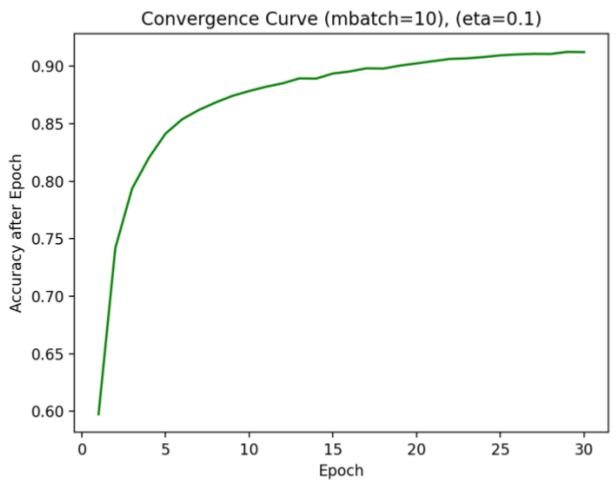
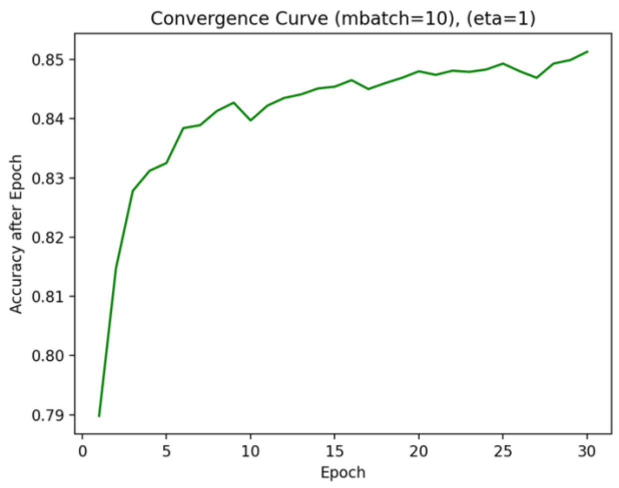
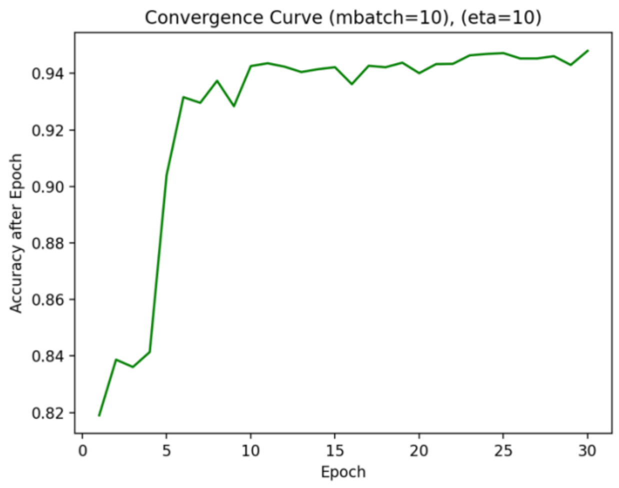
10,000	0.4626	147.799	

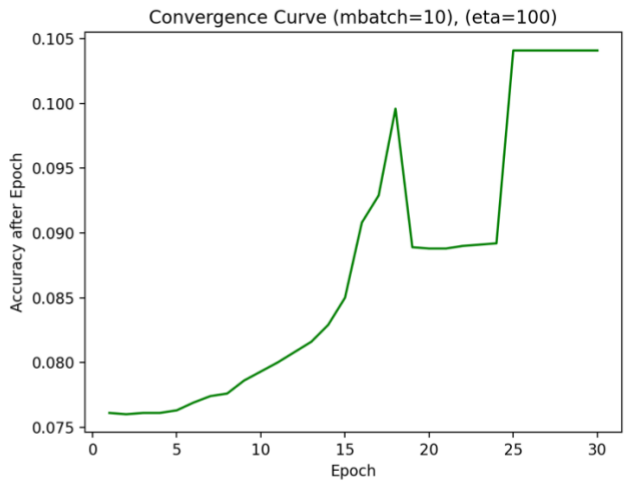
From observing these outcomes, it seems as if the optimal batch size should be relatively small, but not too small for your dataset. We can see that a mini-batch size of 10 performed best on this dataset, and we should expect that our network will have a better accuracy level at with a lower mini-batch size. We expect this because SGD is being performed once for each mini-batch; therefore, a large mini-batch would mean that SGD parameter optimization is only performed a few times. We want SGD to be performed often, but not so often that we slow down our network training significantly. In addition, we can observe more noise in the convergence curve of the models trained with lower mini-batch sizes, so we want to raise our mini-batch size to avoid this noise. Therefore, we use a moderate mini-batch size (say, 10) to not only reduce the runtime but also to increase the frequency that SGD is performed.

2. "On learning rate. Fix the mini-batch size at 10, adjust the learning rate to be 0.1, 1, 10, and 100. Compare the time spent and accuracy at epoch = 30. Plot a bar chart or use a table to present the difference. Also attach the convergence curve for each rate. From the performance differences you observed, what can you learn in terms of an appropriate learning rate? From the convergence curves, estimate a range that optimal learning rate might be. Provide reasoning."

Varying learning rate values:

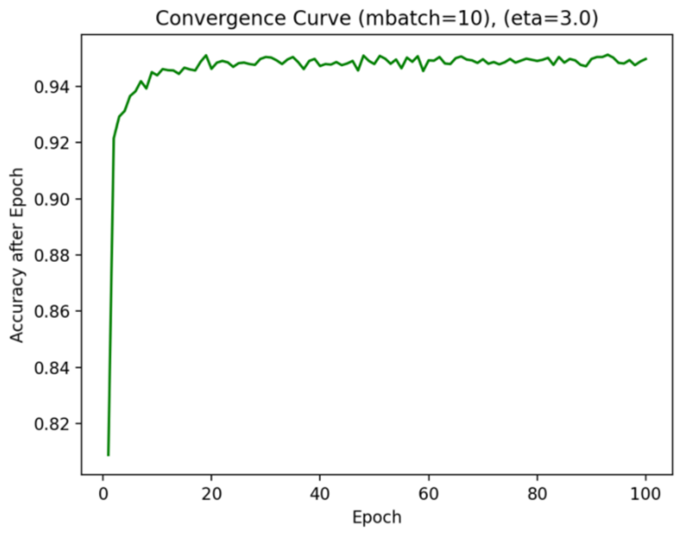
Learning rate	**Acc. at Epoch 30**	**Run Time**	**Convergence Curve**

0.1	0.9123	151.935	
1	0.8513	191.151	
10	0.9480	163.004	

100	0.10410	154.250	

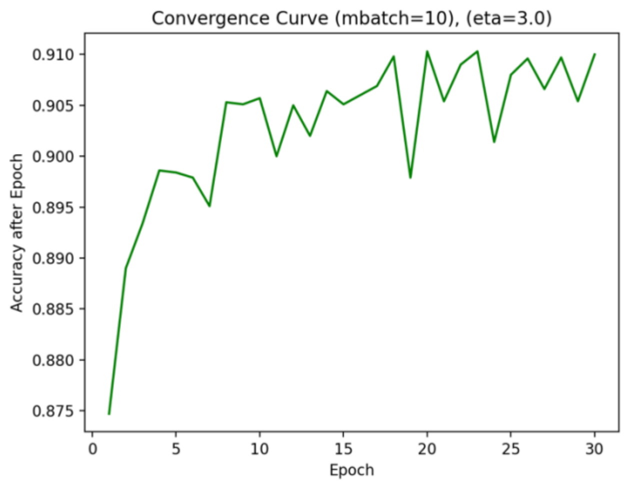
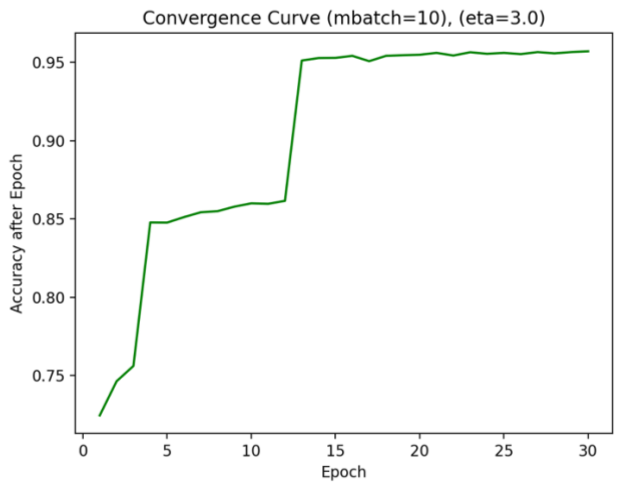
I believe the optimal learning rate should be in between 1 and 10, similar to what Nielsen used in his code. These values give the highest accuracy because they allow for the SGD to aggressively update the weights, compared to other examples we have seen that use low learning rates. This aggressive approach can be shown in the convergence curves; model runs with higher learning rates tend to have more jagged curves. These jagged curves indicate the ability of the weights to fluctuate, effectively overcoming local maxima to reach a lower local (or global) minimum. Therefore, we want a model that strikes a balance between this aggressive and conservative approach, so I propose choosing a value in between 1 and 10.

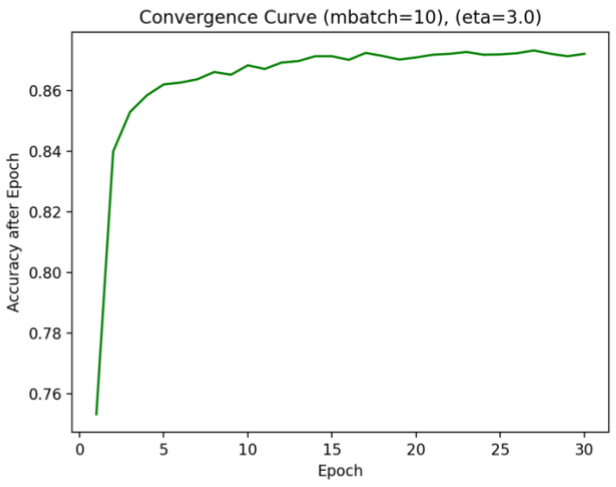
3. "On the number of epochs. Fix the mini-batch size at 10, and learning rate at 3.0, extend the number of epochs to 100. Show the convergence curve."



This plot shows that the model does not substantially gain accuracy after around 25 epochs. Therefore, it is entirely reasonable to stop the model at 30 epochs or less.

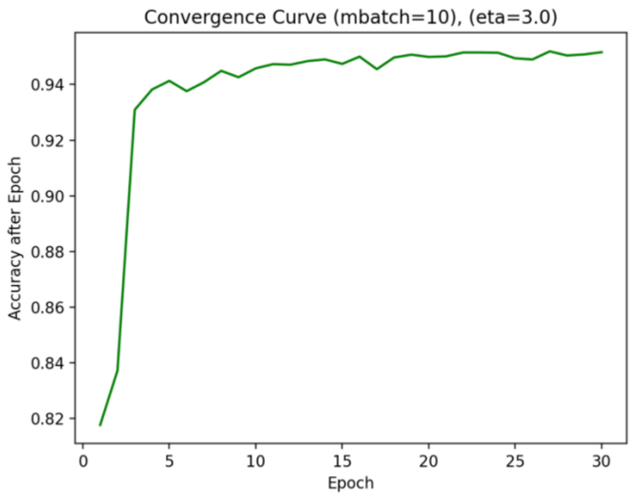
4. “On the network structure. Fix the mini-batch size at 10, the learning rate at 3.0, the number of epochs at 30.”
    a. “Change the number of hidden nodes in the second layer from 10, 50, to 100. Compare the time spent and accuracy at epoch = 30. Plot a bar chart or use a table to present the difference. Also attach the convergence curve for each size. From the performance differences you observed, what is your hypothesis on the appropriate number of hidden nodes?”

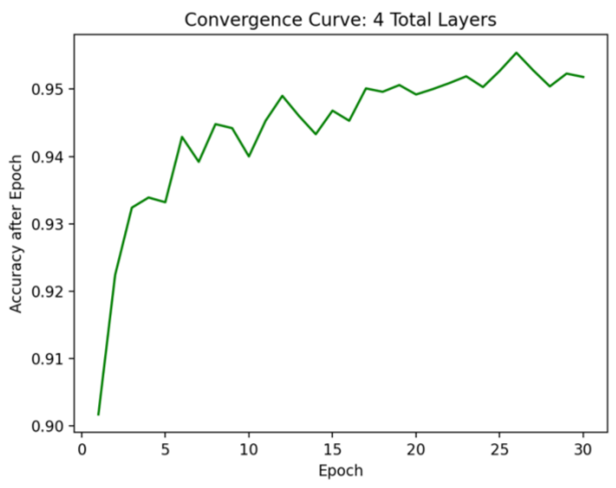
Num. Nodes in 2nd Layer	**Acc. at Epoch 30**	**Run Time**	**Convergence Curve**
10	0.9100	135.898	
50	0.9570	199.892	

100	0.87210	278.371	

From this observation, it seems as if the number of hidden-layer nodes should be held around 50 or the same order of magnitude. Higher numbers of layers will tend to overfit the training data, leading to worse results when introducing new testing data. Therefore, my hypothesis is that we should fix our number of hidden layer nodes to between 10 and 50 in order to achieve generalizability to the introduction of new data.

b. "Change the network structure to four layers, with [784, 30, 20, 10]. Compare the time spent and accuracy at epoch = 30 with [784, 30, 10]. Plot a bar chart or use a table to present the difference. Also attach the convergence curve for each size. From the performance differences you observed, is deeper always better? What is your hypothesis?"

Model	**Acc. at Epoch 30**	**Run Time**	**Convergence Curve**
3 layers	0.9516	167.277	

4 layers	0.95180	213.616	

From my observation, it does not seem as if a "deeper" network necessarily achieves better results. Similar to models with higher number of hidden-layer nodes, adding more layers to make the model deeper might cause overfitting. In this example, the 4-layer model did not overfit the data very much, but it did not achieve better results on the testing data. Therefore, in order to prevent overfitting, we should strive to restrict our network size to as small as still performs well on the validation or testing data.

<u>Bonus:</u>

1. "Replace the sigmoid function with ReLU. Self-study ReLU. Compare the performance with that of Sigmoid. Set the parameters to default."

   I attempted this problem, but I was not successful in my effort. I implemented ReLU and attempted to replace the sigmoid with it in the network, but the model kept on predicting every sample to belong to one class. After some extensive error-checking, I discovered that this algorithm was causing my weights and biases to become very small, eventually becoming NaN (not a number) values. I was not able to find exactly why this was occurring, but I believe that there is some discrepancy in the output of ReLU versus the output of sigmoid in Nielsen's implementation. ReLU outputs 0 for every negative value, and this causes some undesirable results if a significant number of your neuron outputs are negative. This may have been happening in the network, and more sophisticated techniques would be needed to avoid the issue.

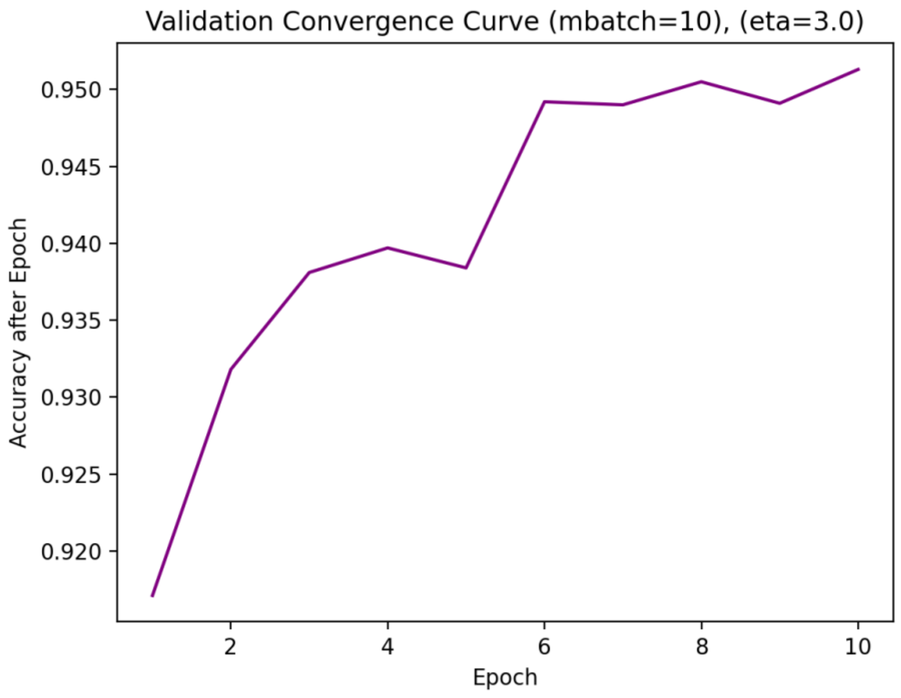
   One sophisticated technique that I attempted was leaky ReLU. This function is a modification of ReLU, but instead of outputting 0 for negative values, it outputs that negative value scaled to some small, constant factor alpha. Therefore, the negative values don't lose their cardinality when transformed by the function. Leaky ReLU is meant to fix the vanishing gradient problem with ReLU that occurs when too many neuron outputs are negative. Therefore, I thought that this would solve my problem, but leaky ReLU also proved to be unsuccessful in this network. I was again getting the network predicting all samples to be one label, which was obviously problematic and produced poor accuracy results.

You can see the notebook for more information on how to find my implementations of both ReLU and leaky ReLU. I am hoping that by including my journey of trying to implement these functions that I can at least get some partial credit for this problem. I simply did not have enough time to comprehensively study the internals of the network and get the functions working.

2. "Nielsen used a fixed number of epochs in his initial implementation of BPNN and he didn't utilize the validation set. Modify the code such that it uses the validation_data to obtain validation accuracy. If the average change in the last 5 epochs is less than, say, 0.01, then stop training. Use the weight and bias obtained then to evaluate the test set."

   I introduced a new function that breaks the iterations of the model when the model's accuracy on a validation set begins to plateau. This algorithm also implements an accuracy threshold, where the user can specify an accuracy goal for the model. This allows the user to prevent the model from converging to a low accuracy, and it forces the model to continue to iterate even past a plateau if the accuracy is too low.

   I ran this model with the standard layers of nodes ([784, 30, 10]) and achieved the following result:



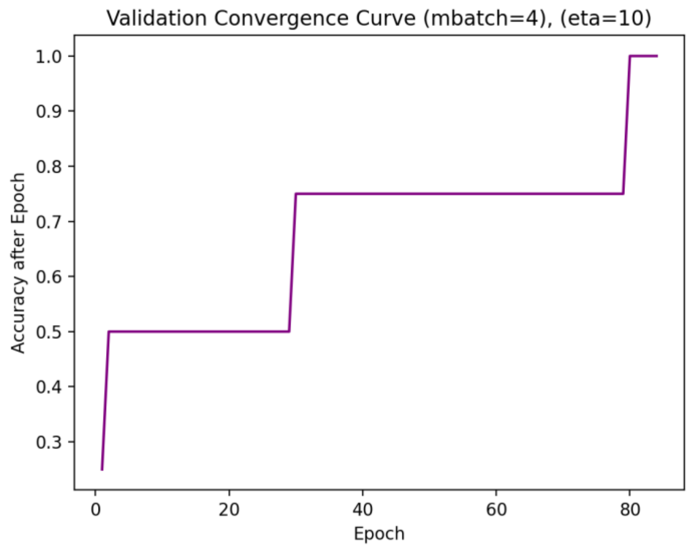
In these trials, the model converges to a high accuracy (0.953) in only 10 epochs, compared to the other trials where we ran the model for a fixed 30 epochs. In practice, I would use an algorithm with this kind of validation-check that limits the number of epochs; this proves to be much more efficient, especially when we do not know how many epochs will be required to reach convergence.

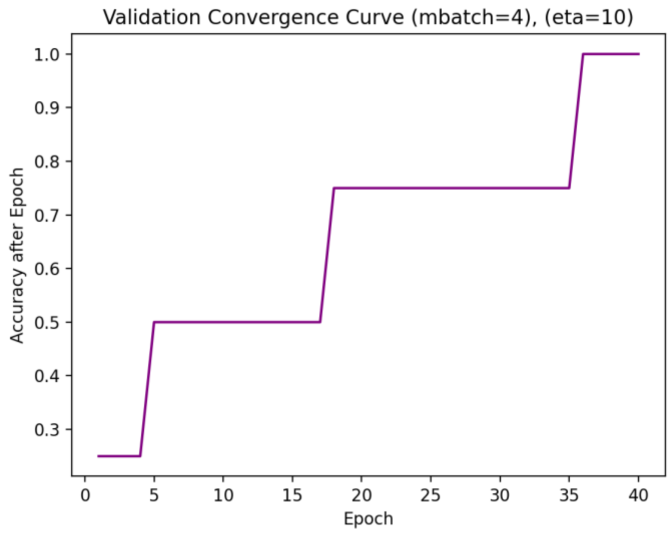
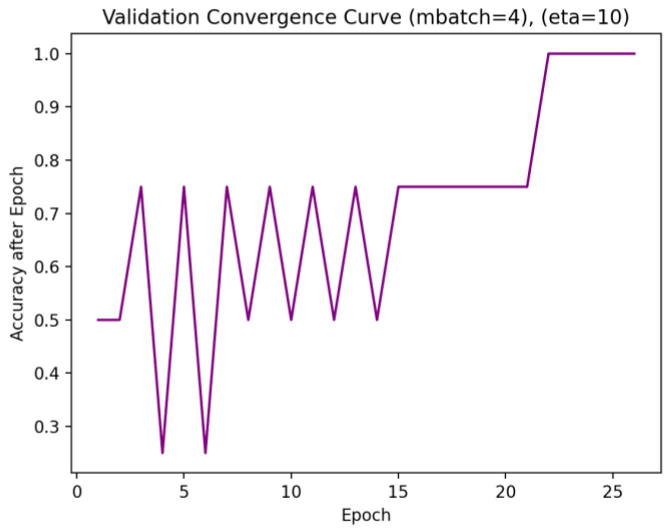
Details on the locations of the implementation can be found in the notebook.

Task 4

"Use the BPNN code used in Task 3 to classify the XOR dataset. Compare various design options and report your finding.**"**

1. "Using only one hidden layer, plot an accuracy vs. iteration figure with 3 different numbers of hidden nodes. That is, the figure should have three convergence curves."

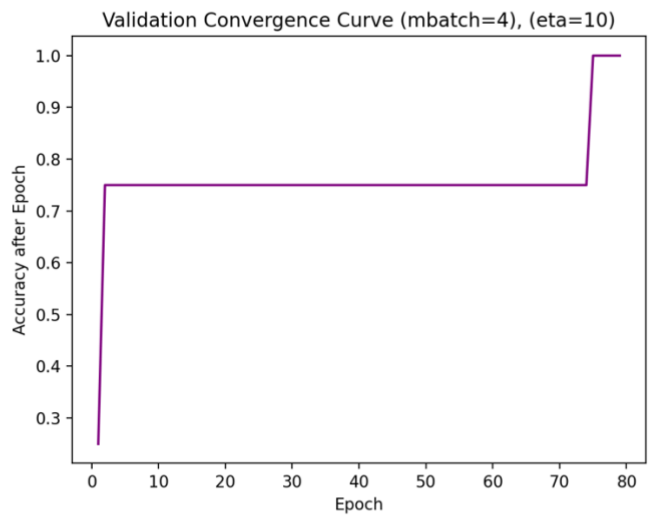
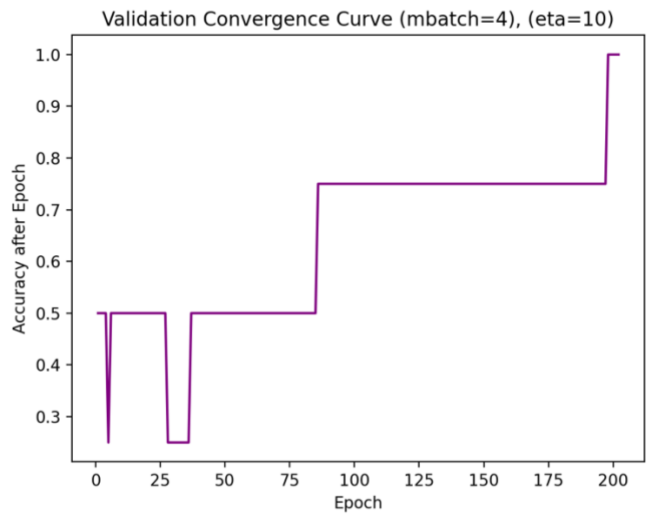
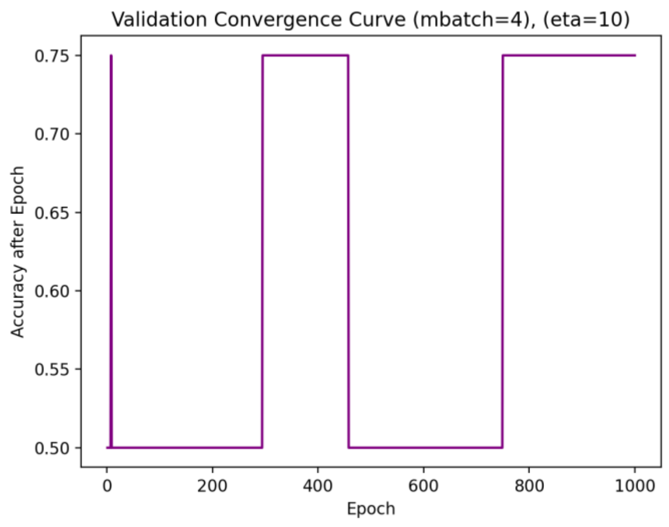
Layers in Model	**Epochs to Convergence**	**Convergence Curve**
[2, 2, 2]	84	

[2, 4, 2]	40	
[2, 8, 2]	26	

Note: From looking at multiple runs of these models, the results above tend to be better than the average results for each network structure. However, the hierarchy remains the same.

2. "Using only two nodes in each layer, plot an accuracy vs. iteration figure with 3 different numbers of layers."

Number of Hidden Layers	**Final Accuracy**	**Epochs**	**Convergence Curve**

1	1.0000	79	
2	1.0000	202	
3	0.75000	1000 (did not converge)	

3. Discuss your findings.

From the tables above, we can observe several important points about neural networks trained on simple data. First, a greater number of nodes in hidden layers may increase convergence time. Since this data is not linearly separable, we require several nodes to make our neural network fit this pattern. In addition, a higher number of nodes might cause the network to be more sensitive to improvements from the gradient descent optimization of weights and biases. These improvements may not be helpful on data that doesn't have much separation, but there is clear separation between the data points of the XOR dataset, so a higher number of nodes in the hidden layer is preferable.

Second, increasing the number of hidden layers does not improve the number of epochs needed for convergence of the network. This is because adding layers just adds more complexity that is not needed. The network, when adding these layers, starts to suffer from the vanishing gradient problem – the phenomenon when the computed gradient becomes very small as the backpropagation algorithm works towards the input layer. Therefore, in this network, we should stick to only one hidden layer.

Final Discussion

This project has been a fantastic way to explore neural networks and just a few of their capabilities. Deep learning is a fantastic tool, but it requires knowledge of many facets of the techniques used in the algorithms, including hyperparameter selection. This is a big issue with the usability of deep learning – there are few reliable and systematic methods for choosing hyperparameters. Therefore, we end up testing models ad hoc or by using techniques such as k-Fold cross validation. These problems emphasize how important the field of automated machine learning will become in the coming years. This field focuses on automating the entire pipeline of building models, but a major focus of this is the selection of hyperparameters for deep learning or regular machine learning models. Through the use of sophisticated optimization techniques, we can select hyperparameters that better suit the problem and related data that we are considering. This has widespread implications for increasing usability of machine learning models, and as data scientists, we must know how to leverage these techniques to improve our models.

One prevailing theme that arises in all of our projects is that we should never introduce unnecessary complexity in our models. This is evidenced by the XOR dataset, where the more complicated models with more layers in the network severely decreased the effectiveness of the model. This simple data required a simple solution, and we should be able to realize how complexity affects the quality of our models.