Project 1:
Neural Networks and Classification

Andy Varner
12153597
Spring 2023

Introduction / Project Description

In this work, we investigate three types of neural networks for classification tasks: a multilayer perceptron (MLP), a convolutional neural network (CNN), and a radial basis function network (RBFN). Our primary objective is to gain insight into how each system operates by exploring its architecture and learning algorithms. Additionally, we aim to identify the optimal configuration for each network to achieve the best classification performance. To achieve this, we compare our ideal network settings and evaluate which type of neural network is most suitable for the classification tasks at hand. We validate our approach by classifying the MNIST dataset and fine-tune the networks to enhance their ability to classify this dataset. Finally, we assess the performance of each network by testing them on the more challenging FashionMNIST dataset.

Background / Theory

In this section we provide an overview of the datasets used to benchmark our networks followed by a detailed discussion of the three types of networks explored in this work: MLPs, CNNs, and RBFNs.

The Datasets

The Modified Institute of Standards and Technology (MNIST) database is a set of 28x28 images of handwritten digits which is used as the "hello world" of machine learning classification tasks. The database contains 60,000 labeled training images and 10,000 labeled testing images. We use this "toy" dataset as a proof of concept for the work presented here. Fashion MNIST is a slightly more challenging dataset which shares the same image size, data format and number of training and testing images as MNIST. This dataset provides another benchmark for the neural network models presented in this work. Both datasets are available from the python package, Torchvision [1].

The datasets are conveniently balanced, so preprocessing the datasets requires only transforming them to Tensors for the PyTorch library and separating the samples into test data and train data and organizing samples and labels for training. To help with generalization, we shuffle the train data and use a batch size of 16. Finally, we one-hot-encode the labels for all experiments.

The Networks

Here we describe the theory, network, and architecture of the MLP, CNN and RBFN.

The Multilayer Perceptron (MLP)

This section provides a discussion of the architecture of the MLP and its learning algorithm. We go into some detail about various ways this can be implemented.

MLP Architecture

The multilayer perceptron is a fully connected, feed-forward network shown in Figure 1a. It consists of an input layer, where data flows into the network, one or more hidden layers, each of which contains a set of neurons with optimizable parameters, shown in Figure 1b. The output layer represents predictions. The neuron model of 1b resembles the McCulloch-Pitts model of a neuron, where inputs, $x_1, x_2, \dots, x_n$, are passed to a combination function, v, which aggregates the inputs. Unlike the McCulloch-Pitts model, this neuron has the added feature of the nonlinearity, σ, at the output end, as opposed to the hard limiter. This ensures the output, y, is monotonic and continuous and therefore differentiable. Each neuron of each layer is connected to the previous layer by weights, w, which are learned by the network through the backpropagation algorithm. In addition to the weights, the network may also have built in a bias, which gives more computational and geometrical flexibility to the network.

For the engineer, the biggest design consideration regarding network architecture is number of neurons (width) and number of layers (depth). The universal function approximation theorem says that a neural network with a single hidden layer and some number of neurons will approximate any function. A network with *too many* hidden layers becomes vulnerable to the vanishing gradient problem, where the gradients of the weights become very small, impeding learning. The number of neurons is generally proportional to ability to learn complex datasets, but also training time. Finding this balance is one of the goals of this work.

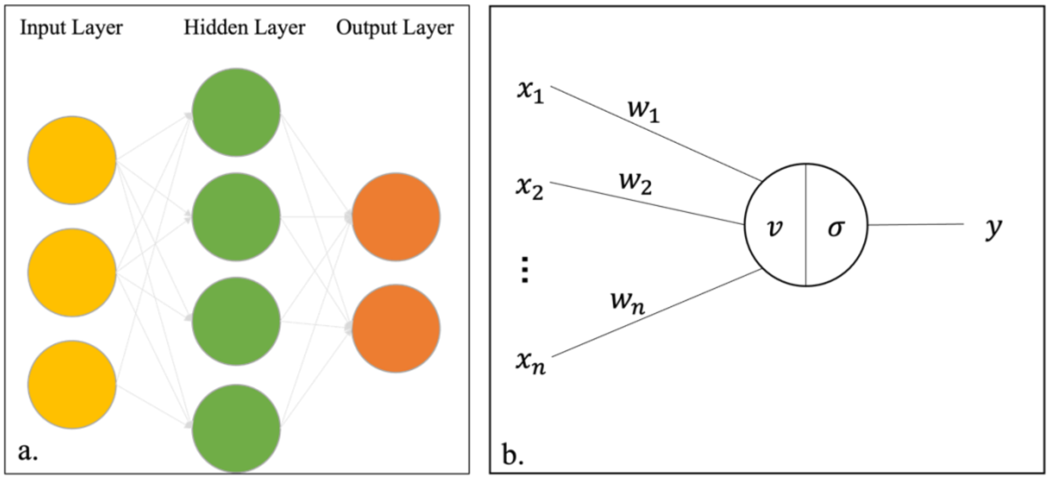


Figure 1. Visualization of the MLP. (a) The fully connected network with input layer, hidden layer, and output layer. (b) A single neuron from a hidden layer of the MLP

A multilayer perceptron can be used for regression tasks or classification tasks. The scope of this project is limited to classification; as such, the network's predictions correspond to classification labels, which are shown for our datasets in Table 1. To reduce bias in the system, we use one hot labels, such that each category is represented as a one-hot vector. In other words, rather than mapping labels to integers, they are mapped to a column of a vector which contains binary values. This ensures that the distance between labels is the same for each category.

Table 1. Classification labels for MNIST and Fashion MNIST

MNIST	Fashion MNIST	Label	One Hot Label
0	T-shirt/top	0	[1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
1	Trouser	1	[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
2	Pullover	2	[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
3	Dress	3	[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
4	Coat	4	[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
5	Sandal	5	[0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
6	Shirt	6	[0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
7	Sneaker	7	[0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
8	Bag	8	[0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
9	Ankle Boot	9	[0, 0, 0, 0, 0, 0, 0, 0, 0, 1]

MLP Learning Algorithm

The learning algorithm for MLP is called backpropagation, which adjusts the weights and biases of the connections between neurons with the goal of minimizing the difference between predicted outputs and actual outputs.

Pseudocode for the backpropagation algorithm is provided in Figure 2. First, input data is fed forward into the network as in Figure 1.a. The inputs are aggregated in the combination function and passed through the nonlinearity. A prediction is produced at the output layer, and error is calculated. This error is then propagated back through the network via backpropagation. Weights and biases are adjusted to minimize error, somewhat like a feedback control system. This process continues for a determined number of iterations, or epochs, and the calculation used to determine error is used to calculate network loss, which is a metric for evaluating the performance of the network. Network evaluation will be discussed in more detail in future sections.

```
for epoch in epochs:
   Forward pass:
          Aggregate inputs
          Pass through activation function / nonlinearity
   Compute error
   Backpropagation:
          Compute weight
          Compute weight update
```

Figure 2. Pseudocode for backpropagation algorithm

Within this framework, there a few options for carrying out various tasks throughout the network, particularly the activation function, objective function, and optimizer, which assits with weight updates.

The Activation Function a.k.a Nonlinearity

Once inputs are aggregated, the activation function is applied to the output of each neuron in the network, introducing a nonlinearity, which enables learning and making predictions. Tanh **Error! Reference source not found.**, sigmoid **Error! Reference source not found.**, and relu **Error! Reference source not found.** are well-studied activation functions for the multilayer perceptron. Their graphs are shown in **Error! Reference source not found.**. We know that sigmoid and tanh suffer from the vanishing gradient problem, but tanh has the advantage of not "losing" all the negative values, unlike sigmoid. Relu is known to overcome the vanishing gradient problem, and the community generally regards it as the activation function with the highest overall performance. We use Relu for all of our experiments, holding this parameter constant since it is well-understood, and doing so gives us the ability to explore other parameters in greater detail.

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$S(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1}$$

$$f(x) = \begin{cases} 0, if\ x < 0 \\ x, if\ x \leq 0 \end{cases}$$

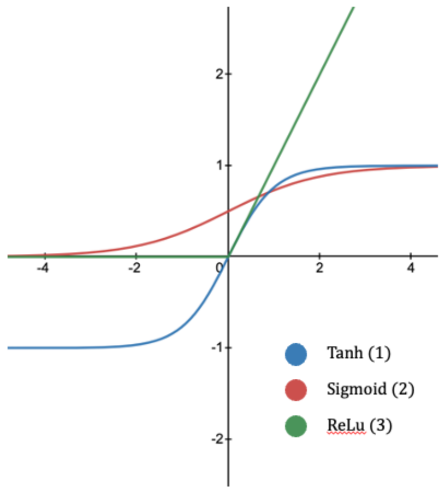


Figure 3. Common activation functions

The Objective Function

The objective function is used to calculate error. Although many sophisticated objective functions are available in the PyTorch library, but mean squared error (MSE) is often used in the current literature. It would be interesting to do an entire study of objective functions for classification tasks, but to keep the scope of this paper limited, we stick with MSE which works well with one-hot-encoding. MSE is defined as

$$MSE = \frac{\sum_{i=1}^{N} |y_i - d_i|}{N}$$

where y_i represents a single model prediction, d_i represents its corresponding truth label, and the calculation is performed for all samples, N.

The Optimizer

The optimizer assists with weight update algorithm, which is discussed in detail in the next section. Common algorithms are stochastic gradient descent (SGD) and Adam. Stochastic gradient descent is the simplest optimizer available in the PyTorch library. It updates the model's weights using a fixed learning rate. It replaces the actual gradient, calculated from the entire dataset, by an estimate, calculated from a randomly selected subset of the data, with the aim to speed up learning. The Adam optimizer adapts the learning rate dynamically based on the current gradient. The community generally regards Adam as superior to SGD, so we keep the Adam optimizer for each of our experiments.

Weight update rule

The backpropagation algorithm's weight update rule is defined as follows:

$$w_j^h = w_j^h - \alpha \nabla w_j^h$$

where h represents a given layer of the network and j represents a single neuron at the hth layer. The learning rate, α, is typically a small value, 0.0001. The gradient of a given weight is given by

$$\nabla w_j^{(h+1)} = \delta_j^h \frac{\partial y_j^h}{\partial w_j^h}$$

which is the local gradient of the jth neuron of the hth layer multiplied by the partial derivative of its output, y with respect to its weight, w. The local gradient, δ_j^h is given by

$$\delta_j^h = \sum_{i=0}^{z^{h+1}} \left(w_{ij}^{h+1}\delta_{ij}^{h+1}\right)\frac{\partial y_j^h}{\partial v_j^h}$$

which is sum of the products of the weights and local gradients of the next hidden layer, times the partial derivative of its output, *y* with respect to the activation function, *v*. This is the backpropagation algorithm which updates MLP parameters for each epoch of training.

Convolutional Neural Network (CNN)

Like MLPs, CNNs are feed-forward neural networks that make use of the backpropagation algorithm and optimization algorithms. Unlike MLPs, CNNs accept 2D or 3D inputs. This is related to the architecture of the network, which we outline here.

CNN Architecture

The goal of a CNN is to locate features from combinations of pixels. The architecture of a CNN consists of stacks of hidden layers in sequence. These layers are followed by activations layers and pooling layers, as in Figure 4. The input to the network is an NxN 2-dimensional image, which gets passed to a convolution layer followed by a pooling layer. For simplicity, the figure shows a single hidden layer, but in practice many layers may be used. The fully connected layer maps the output of the final pooling layer to some distribution of values in the output layer, depending on the problem. Since we are classifying MNIST, the output layer contains a distribution of 10 values, one for each class, which sum to 1.

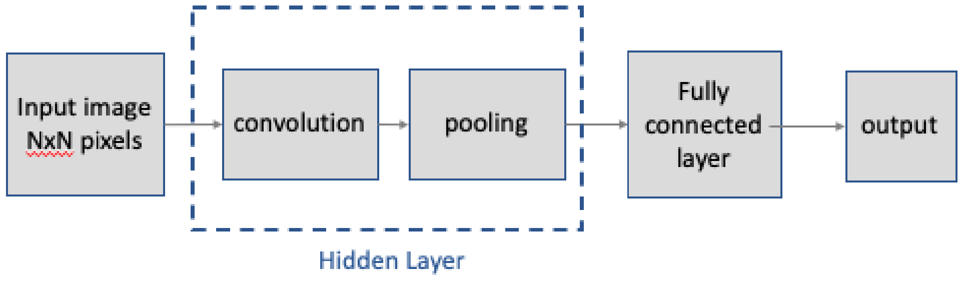


Figure 4. Simplified CNN architecture

The Convolution Layer

The convolution layer is built from many small square templates, known as convolutional kernels. The kernels slide across the image and collect patterns. Consider the image of a cat in Figure 5. This example shows a 3x3 kernel sliding across the image with a stride of 1. Let the features observed in the green 3x3 kernel map to the green square on the right, and so forth for the red, blue and yellow kernels. This operation condenses the image into a feature map. The pooling layer does some aggregation on these condensed feature maps.

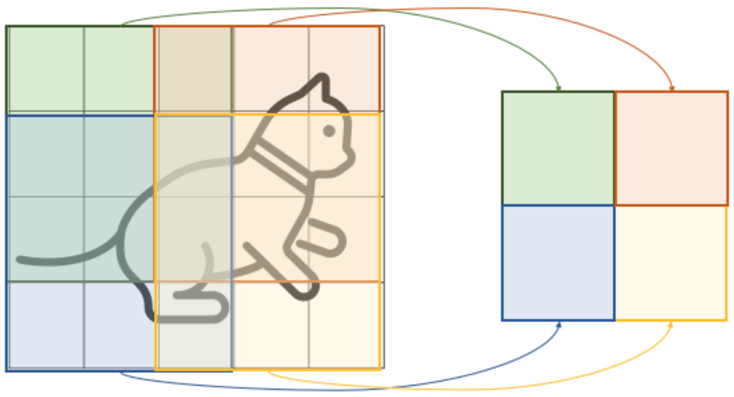


Figure 5.   Illustration of a 3x3 convolutional kernel

The Pooling Layer

The convolution operation gives us a feature map, which is reduced to a set of scalar values. The pooling layer aggregates those values in some way; we us MaxPool, which takes the max of those values. Figure 6 shows this operation: The 3x3 set of values corresponding to the first position of the kernel has a max value of 5, when it shifts by 1, the next 3x3 set has a max value of 4, and so on as shown in the figure. These scalar values can be thought of as features.

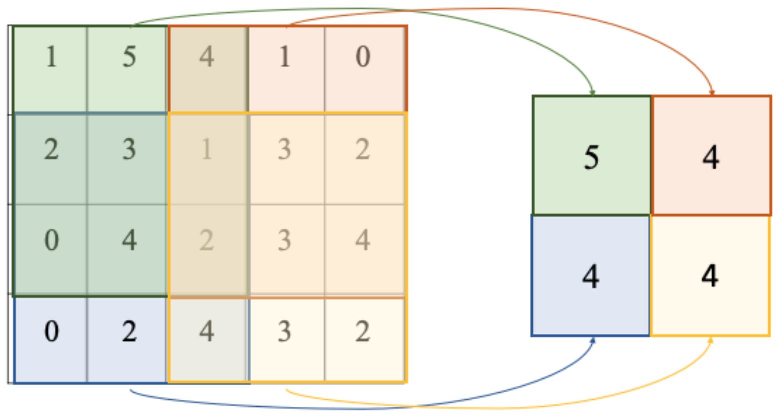


Figure 6.   Illustration of a pooling layer

The choice of kernel size and shift size are implementation choices for the designer. This work will focus on an exploration of kernel size for CNN performance.

Fully Connected Layer and Output

A fully connected layer, typically an MLP, is at the output of a CNN model to map the output of the final layer to the prediction. For this work, this is a 10-class output corresponding to the 10 classes of MNIST and Fashion MNIST.

The goal of the CNN is to locate features. Beginning with a high dimensional NxN image, we might imagine the first layer finds the simplest features, e.g. edges, curves, corners. The second layer might find slightly more complex feature, e.g. circles and squares. Each layer increases in complexity, so the third layer may find combinations of circles and squares and so on. We know in some sense that the network reduces the image to its basic building blocks and finds patterns within those building blocks, but the inner workings of this remains a black box.

Radial Basis Function Network (RBFN)

While MLPs and CNNs typically use sigmoid, tanh, or preferably ReLU as activation functions, a RBFN uses a radial basis function. We can think of the typical activation functions as sort of squishing functions, but basis functions are typically Gaussian functions, centered at different points in the input space with different widths. The neurons of Figure 7 contain $f_n(\mu, \sigma)$, the Gaussian consisting of a mean and standard deviation. The RBFN uses radial basis functions as a neuron to compare input data to training data. The input vector is passed through multiple neurons with varying weights, and the sum of the neurons produce a value of similiarity. High similiary values are associated with high degrees of matching between input data and training data. This type of similarity comparison allows for nonlinear classification.

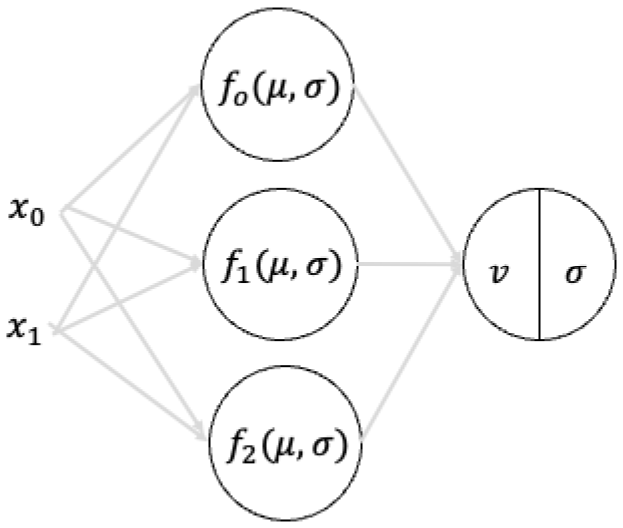


Figure 7. Radial basis function network

RBF networks also have fewer hidden layers than MLPs and CNNs, and unlike CNNs, they are typically fully connected. They can be trained using clustering, least squares, or other algorithms, which makes them useful for datasets that are sparse or noisy. RBF parameters are learned via clustering; then, standard neuron parameters are learned via pseudo-inverse least squares.

An interesting way to study RBF networks is to look at initialization methods. We will compare an RBF network initialized with k-means clustering to an RBF network initialized randomly, and evaluate their performance.

Experiment Design and Evaluation of Results

For evaluation, we look at loss over the training period and other metrics discussed in this section.

Evaluation Methods

We keep track of loss and develop confusion matrices to guide evaluation of the networks. From the confusion matrices, we determine accuracy, recall, precision, and F1 score for each experiment.

A confusion matrix (1) is a graphical representation of the overall performance of the network. It shows, for each class, the number of true positive (TP), true negative (TN), false positive (FP), and false negative (FN) classifications made by the network. True positives are the number of samples labeled correctly as belonging to the class. True negatives are the number of samples labeled correctly as not belonging to the class. False positives are the number of samples that are incorrectly labeled as belonging to the class, and false negatives are the number of samples that are incorrectly labeled as not belonging to the class. These results are used to calculate the following metrics: Accuracy (2) is how close a given set of measurements are to their true value. Recall (2) is the ability of the classifier to find all the positive samples. Precision (3) is the ability of the classifier not to label as positive a sample that is negative. The F1 score (4), or harmonic mean of precision and recall, is another measure of accuracy.

$$(1)\quad Confusion\ matrix: \begin{bmatrix} TP & FP \\ FN & TN \end{bmatrix}$$

$$(2)\quad Accuracy = \frac{TP + TN}{TP+TN+FP+FN}$$

$$(3)\quad Recall = \frac{TP}{TP+FN}$$

$$(4)\quad Precision = \frac{TP}{TP+FP}$$

$$(5)\quad F1\,Score = \frac{2TP}{2Tp+FP+FN}$$

These metrics allow us to assess the performance of the network from different perspectives, showing which classes it performs well on, and which classes it is struggling with. We will periodically use these metrics to assess the impact of hyperparameters on network performance. Next, we discuss the design and results of our experiments.

Multilayer Perceptron Design and Results

In our study of the MLP, we would like to know how width and depth affect network performance. Holding constant ReLU as our activation function, Adam as our optimizer, and MSE as our objective function, we conduct a width vs. depth experiment.

Width vs Depth

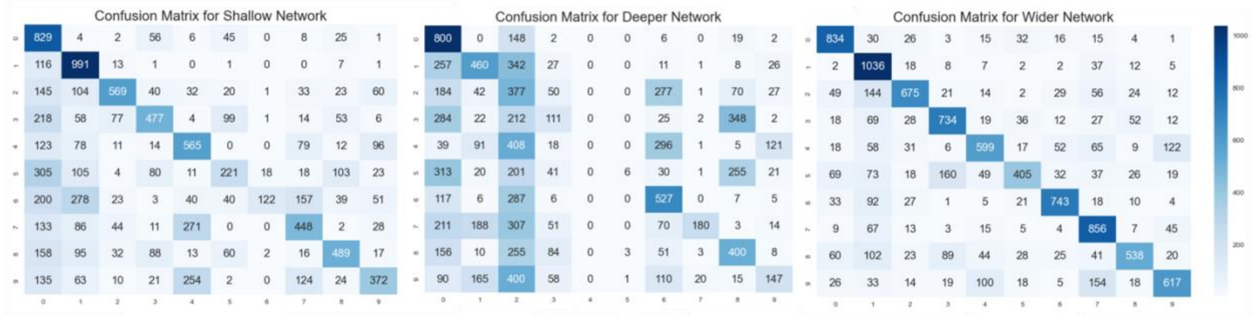
Network width refers to the number of neurons in a single layer, and depth refers to the number of layers in the network. Generally, the number of neurons in subsequent layers is reduced by powers of two, which is expected to improve performance and robustness of the network. This is a technique that remains constant for this experiment.

Three networks are built to explore width vs. depth; their parameters are outlined in Table 2. Experiment 1 is the control experiment, with three layers and 64 neurons in the first layer. Experiment two increases the number of layers and keeps the width the same, and experiment 2 increases the number of neurons and keeps depth the same. All experiments are run for 50 epochs.

Table 2. Parameters for width vs. depth experiment

	Experiment 1 (control)	Experiment 2 (depth)	Experiment 3 (width)
# layers	3	4	3
# neurons in layer 1	64	64	512

The confusion matrices and F1 scores for these experiments are shown in Figure 8. The Shallow, narrow network of experiment 1 performs moderately. It is by no means an excellent score, but it is clear from the somewhat darker region along the diagonal of the confusion matrix and 0.933 F1 score that the model is beginning to learn. Interestingly, adding a layer in experiment 2 actually worsens performance. The confusion matrix has lost the darker diagonal and F1 score went down. Adding neurons (experiment 3) significantly improves the model. Experiment three has both the highest F1 score and the most distinct diagonal.



```
---------  -----
F1 Score   0.933
---------  -----
```

```
---------  -----
F1 Score   0.862
---------  -----
```

```
--  -----
F1  0.981
--  -----
```

Figure 8. Confusion matrices and F1 scores for the three MLP models

The loss analysis of Figure 9 confirms this story and adds some additional texture. The deeper network of experiment 3 has the lowest loss, which is consistent with the story that this is the "best" model of the three. The wider network of experiment 2 has the lowest F1 score and worst looking confusion matrix, and it also has the worst loss. The control experiment, which performed the second best also has the second-best loss. Additional metrics, accuracy, precision, and recall, tell a similar story. These metrics are shown in Appendix I.

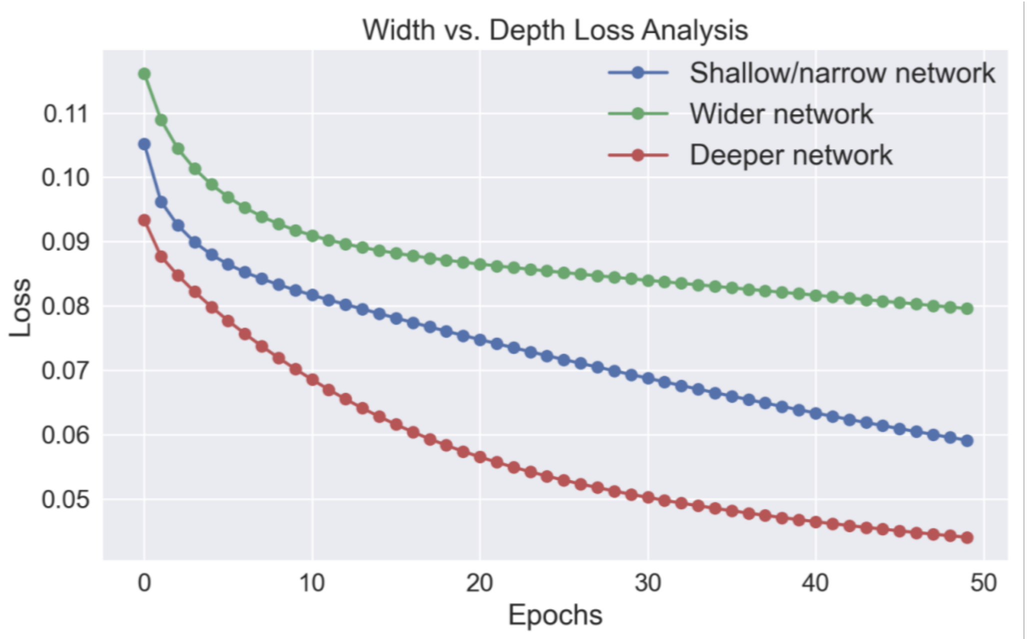
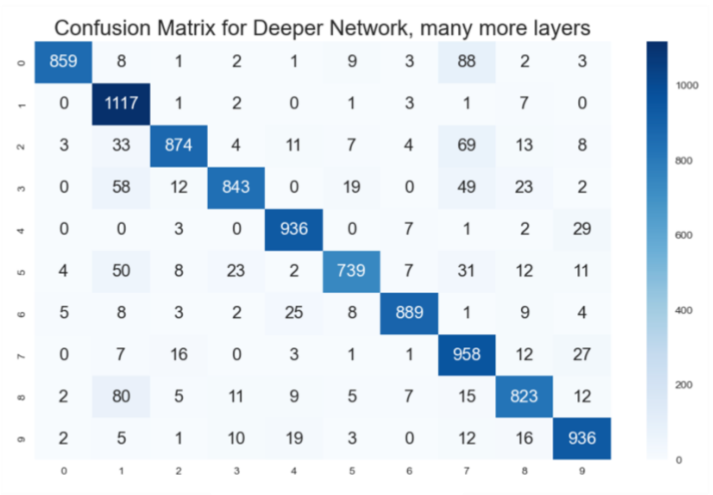


Figure 9. Analysis of loss for Width vs. Depth

The outcome of this experiment is unexpected. We would expect adding more layers to increase performance. It is interesting that increasing the number of layers by 1 actually worsens performance. On reflection, the increase in neurons from 64 to 512 is a much larger leap than the jump in layers from 3 to 4. The difference in number of parameters added to the network is very large, so this was not a great choice. It would have been smarter to keep the number of parameters added the same, which requires adding 15 layers instead of just 4 layers to the second experiment to really get a feel for how depth impacts network performance. The results of this re-do are shown in Figure 10. This is interesting because, while the F1 score still does not beat the model with increased width, it does improve on the control model. Additionally, the confusion matrix rivals that of the width experiment. However, the training time for this experiment was much, much longer, taking a full hour running PyTorch on the M1 GPU. The width network was at least twice as fast.



```
--------  -----
F1 Score  0.995
--------  -----
```

Figure 10. Confusion matrix and F1 score for a re-do of experiment 2, network depth

Convolutional Neural Network Design and Results

The MNIST dataset is not particularly challenging for the CNN, which is state of the art for image classification. In fact, even the vanilla CNN classifies MNIST with over 96 % accuracy, with an F1 score, precision, and recall also all above 96 %. It would be difficult to build on this meaningfully for a performance evaluation alone, so we designed an experiment to allow us to look at the network closely with respect to kernel size. We visually observe the weights learned by networks with different kernel sizes to try to determine how the network learns weights depending on kernel size.

The CNN is distinct from the MLP primarily due to its use of shared weights and its ability to perform feature extraction through the use of kernels, which is not possible in the MLP. Although the black box process cannot be fully explained, the weights of the CNN may provide some insight into the internal workings of the network. To explore this, we visualize the weights of the trained CNN for classification of MNIST to gain a better understanding of how the network is processing information. We examine the weights of several networks with varying kernel sizes with high performance, and we observe how these weights change as the network becomes more complex.

For the sake of experimental simplicity and computation time, we reduced the dataset to 1s and 8s. We used two hidden layers with MaxPool and ReLU, keeping stride and padding both set to 1 consistently. The output layer is an MLP with ReLU. The model uses the Adam optimizer and mean squared error with one hot encoding, consistent with the MLP. The only parameter we changed for these experiments is kernel size; we used kernel sizes 1, 3, 5 and 10.

Regardless of the kernel size, the CNN demonstrates rapid learning, with a reduction in loss to almost zero by the tenth epoch, as illustrated in Figure 11.

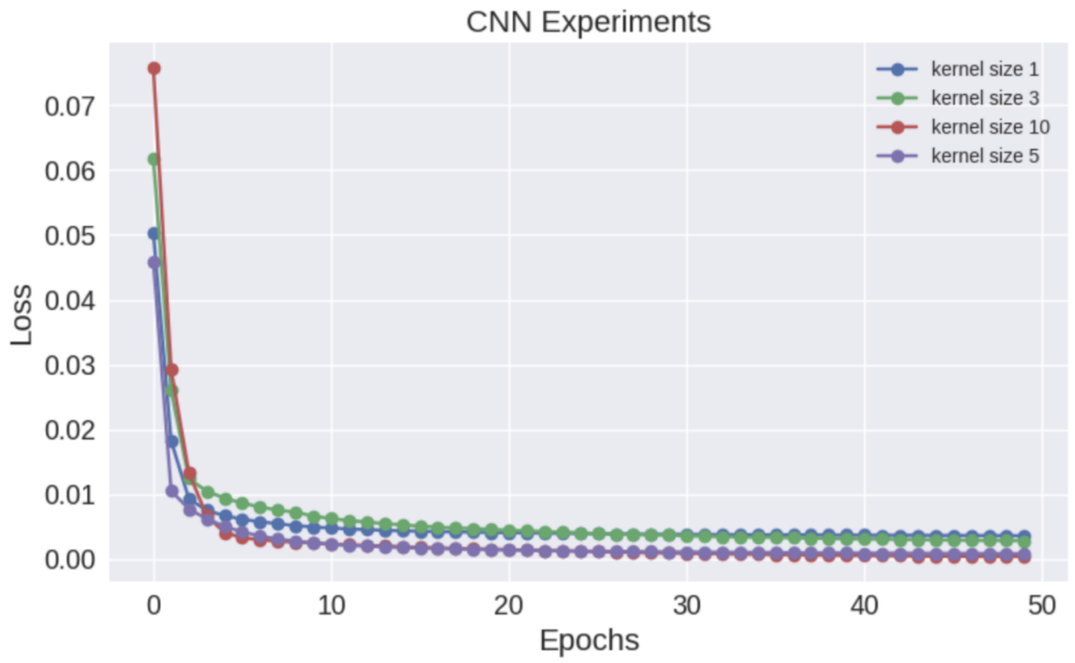
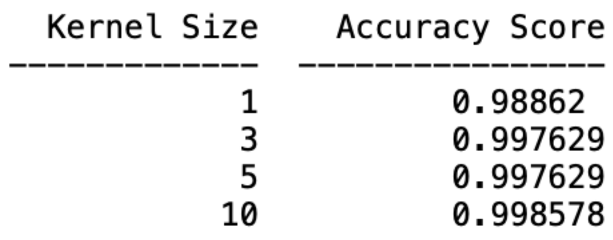


Figure 11. Loss for CNN experiments with increasing kernel sizes over 50 epochs for classifying 1s and 8s

Regardless of kernel size, the CNN has extremely high accuracy for classifying 1s and 8s, shown in Table 3.

Table 3. CNN Accuracy scores by kernel size



```
  Kernel Size    Accuracy Score
-------------  ----------------
            1          0.98862
            3          0.997629
            5          0.997629
           10          0.998578
```

These metrics are unsurprising; we already confirmed that the CNN performs above 96 percent on the entire dataset (see Appendix for more data visualization). The interesting piece comes from visualizing the kernels. The visualized weights from these experiments are shown in Figure 12. The 1x1 kernel is the most mysterious; since we only get to look at one pixel as a time, we don’t get any visual intuition from this at all. As kernel size increases, we get more complexity. We thought perhaps only visualizing 1s and 8s would give us an opportunity to see the weights take shape, but this is not obvious from the visualization. Still, we can conclude from this that greater complexity, i.e. larger kernel size results in higher accuracy and lower loss, which feels trivial in this case, but may make a bigger difference with a more complex dataset.

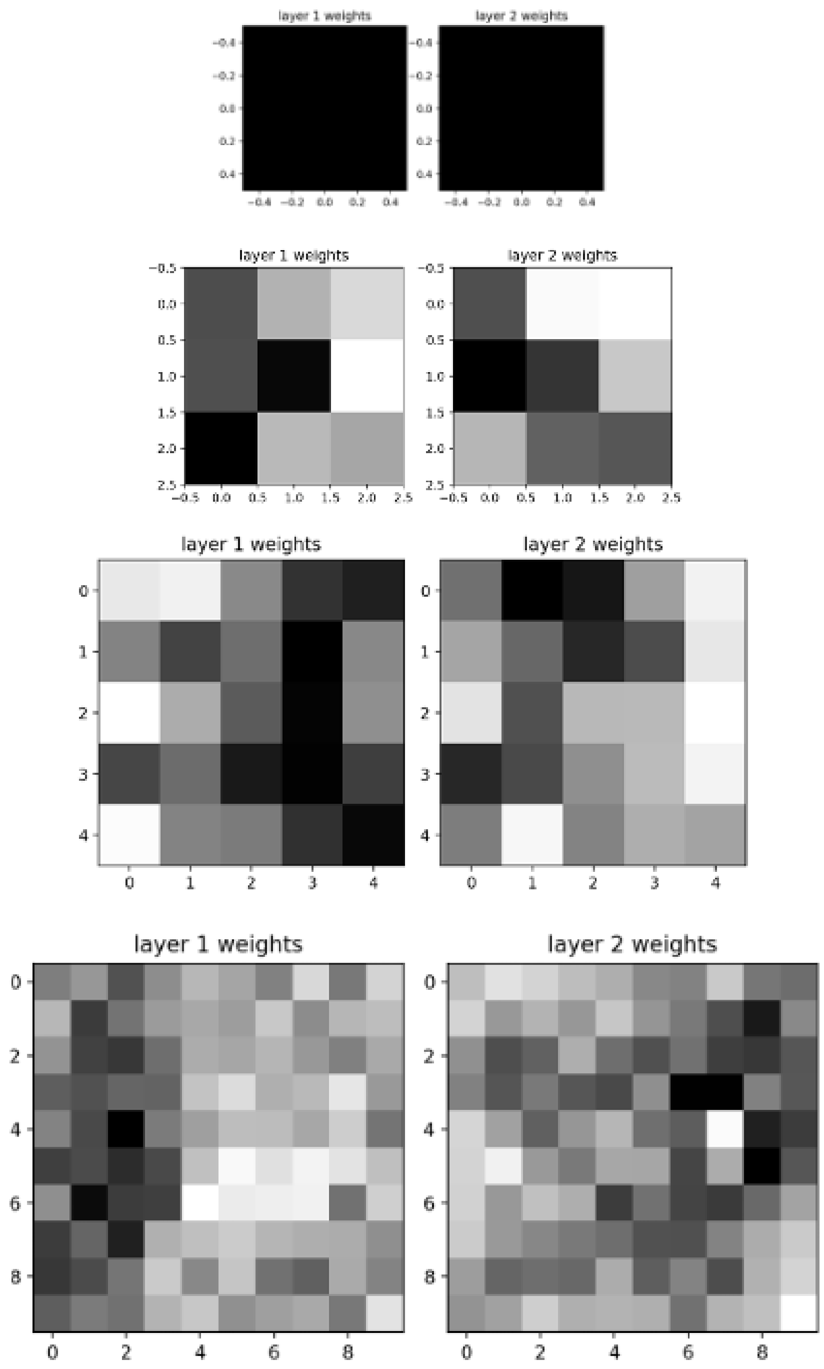


Figure 12. Visualization of weights for kernel size 1, 3, 5, and 10

Radial Basis Function Network Design and Results

To train a RBF Network, we initialize the centers of the radial basis functions, estimate the width of each function, and optimize the weights. In this study, we aimed to explore different initialization strategies, including random initialization and initialization using k-means. However, despite our efforst, the RBF network failed to classify MNIST and did not learn.

When using k-means initialization, we trained the network for ten epochs, with low accuracy values throughout. When using k-means initialization, we trained the network for ten epochs and achieved an extremely poor accuracy of 11% for all ten epochs. When we changed to random initialization, the network displayed similar behavior, with accuracy stagnating at 9%. We tried several strategies to improve the model, such as adjusting the learning rate from 0.001 to 0.01 and 0.1, and increasing the number of centers from 10 to 500, but none of these changes had any impact on the network's performance.

Upon further examination, whether we use MSE or Cross Entropy Loss, the model produces NaN values for loss. This likely means we have a data preprocessing error that warrants further investigation.

MLP vs. CNN for FashionMNIST

In our MLP study, we found that significantly increasing the neurons in the network increases network performance. We found that increasing the number of layers by a very small amount actually impedes network performance, and while increasing the number of layers dramatically gives a similar bump in performance to the equal-parameter increase in neurons, it significantly hampers computation time. Based on this, we kept the same increase in neurons as demonstrated in the MLP experiments, but increased the number of layers by 3 instead of just one or 20. This gave us extremely low loss from the very beginning, and very near zero after 50 epochs, high scores across all metrics for the training period, and a very nice looking confusion matrix. The MLP performs very well for this classification task.

We found that the CNN performs incrementally better for MNIST with increasing kernel size, but it was computationally time-intensive, so for the full Fashion MNIST dataset, we used a kernel size of 5. The two networks performed nearly the same for this task with the described configurations. Surprisingly, the MLP performed slightly better than the CNN, with slightly lower loss, shown in Figure 13.

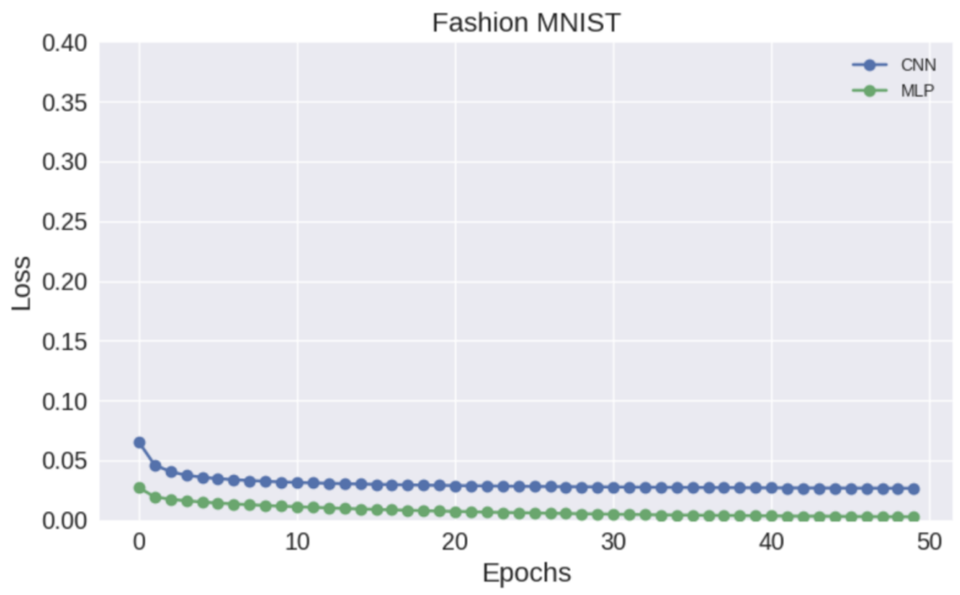
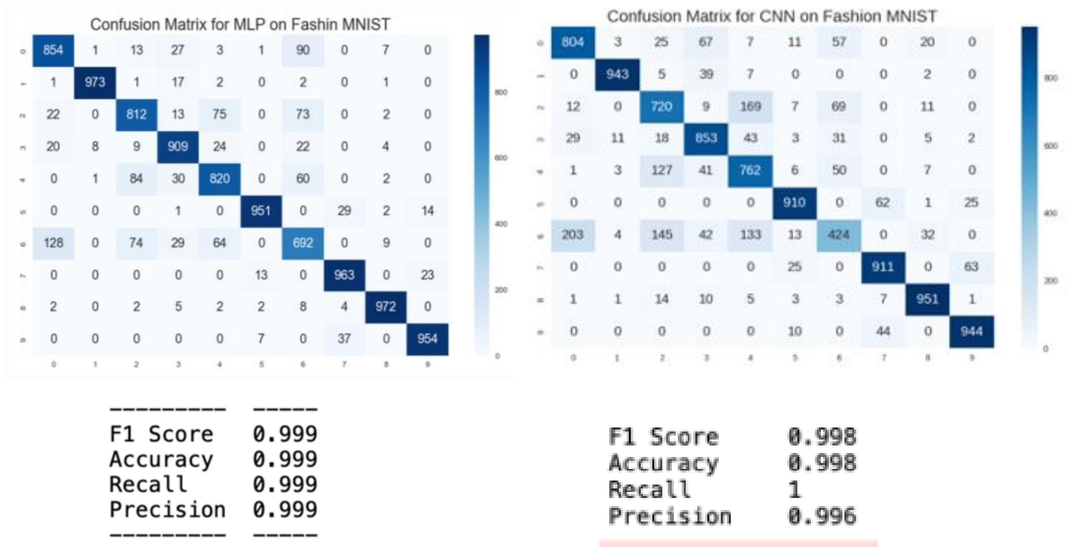


Figure 13. MLP vs CNN for Fashion MNIST

The confusion matrices and training metrics for the MLP and CNN for classifying Fashion MNIST are shown in Figure 14.



	0	1	2	3	4	5	6	7	8	9
0	854	1	13	27	3	1	90	0	7	0
1	1	973	1	17	2	0	2	0	1	0
2	22	0	812	13	75	0	73	0	2	0
3	20	8	9	909	24	0	22	0	4	0
4	0	1	84	30	820	0	60	0	2	0
5	0	0	0	1	0	951	0	29	2	14
6	128	0	74	29	64	0	692	0	9	0
7	0	0	0	0	0	13	0	963	0	23
8	2	0	2	5	2	2	8	4	972	0
9	0	0	0	0	0	7	0	37	0	954

	0	1	2	3	4	5	6	7	8	9
0	804	3	25	67	7	11	57	0	20	0
1	0	943	5	39	7	0	0	0	2	0
2	12	0	720	9	169	7	69	0	11	0
3	29	11	18	853	43	3	31	0	5	2
4	1	3	127	41	762	6	50	0	7	0
5	0	0	0	0	0	910	0	62	1	25
6	203	4	145	42	133	13	424	0	32	0
7	0	0	0	0	0	25	0	911	0	63
8	1	1	14	10	5	3	3	7	951	1
9	0	0	0	0	0	10	0	44	0	944

```
---------  -----
F1 Score   0.999
Accuracy   0.999
Recall     0.999
Precision  0.999
---------  -----
```

```
F1 Score      0.998
Accuracy      0.998
Recall        1
Precision     0.996
```

Figure 14. Metrics for MLP (left) and CNN (right) on Fashion MNIST

Conclusions and Future Work

This work investigated three different neural network architectures for the purpose of classification. We found success classifying both MNIST and Fashion MNIST with the MLP and the CNN. The CNN should have outperformed the MLP, but results were very close to the same. Perhaps an even more difficult dataset than Fashion MNIST, like CIFAR, would give the CNN a chance to stand out a little bit more.

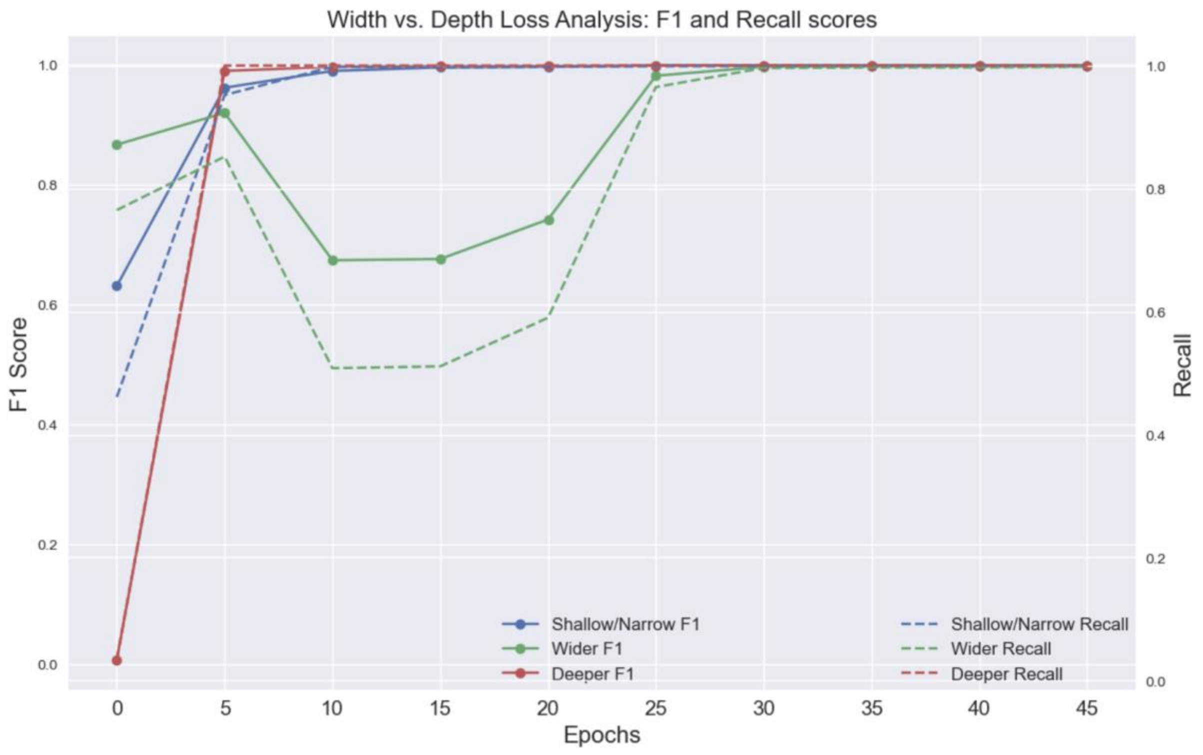
The RBF should have classified MNIST, but we encountered an error that we could not debug. This error was likely due to preprocessing the data, so an investigation should be conducted to determine what the datasets look like, and we we are getting NaNs for loss. Beyond this, it will be interesting to do a full study of how initialization impacts network performance.

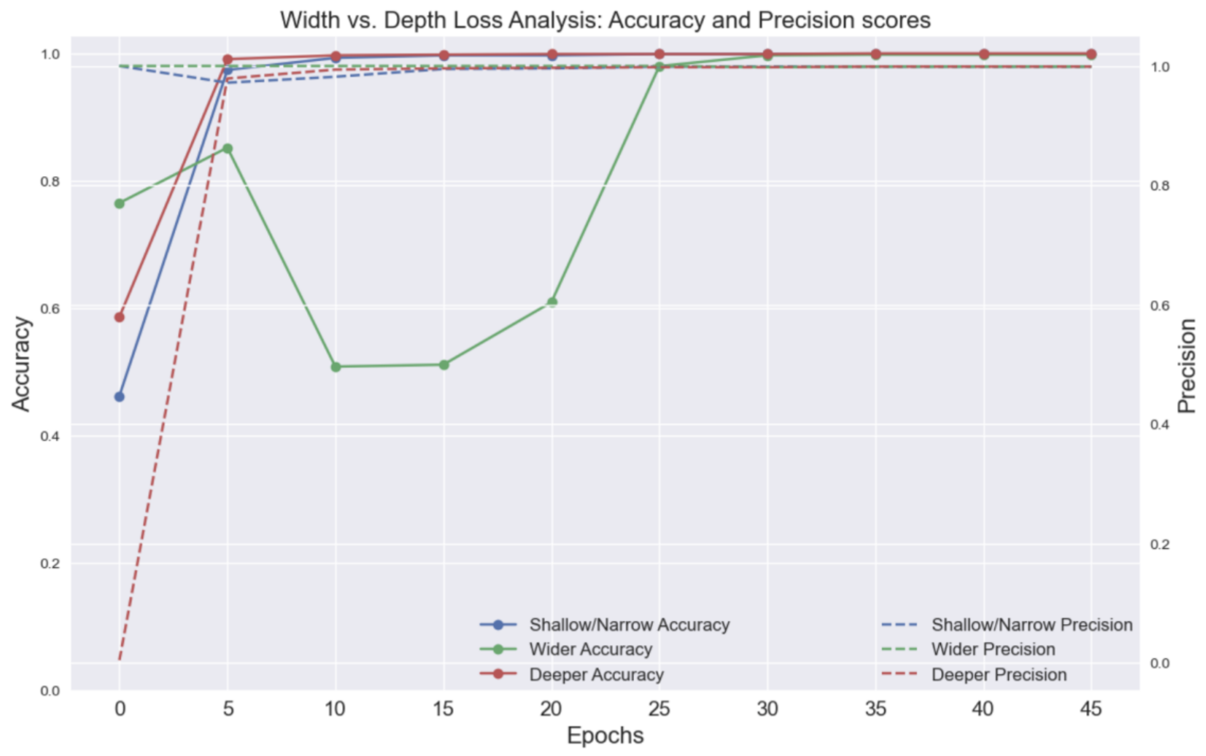
Appendix

I. Additional Data from MLP Experiments

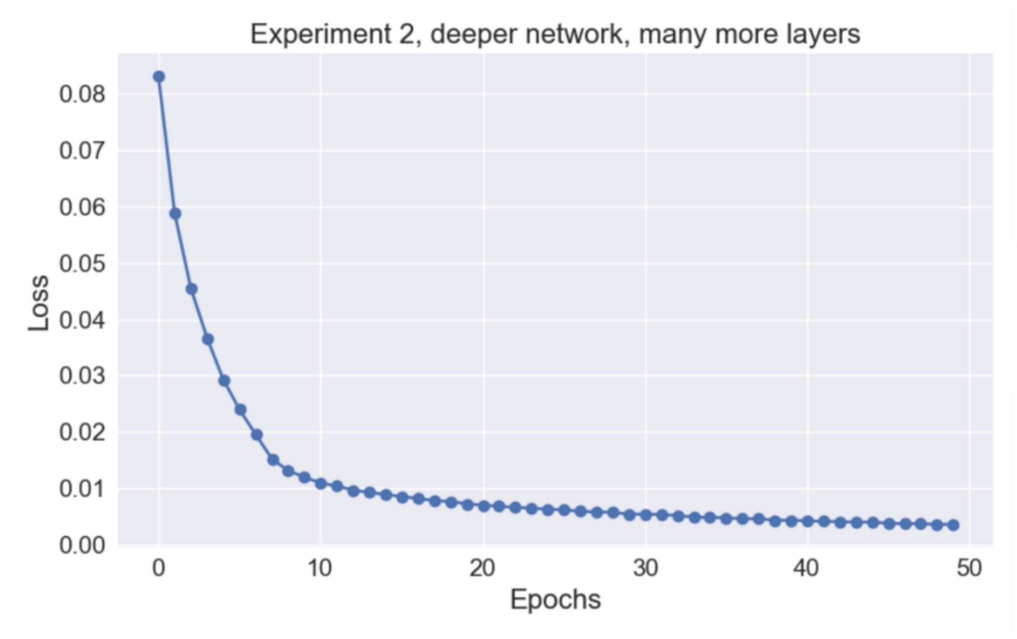
Additional metrics for first three experiments

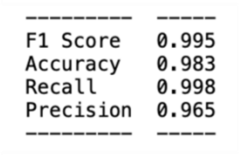
```
Shallow Net                 Deeper Net                  Wider Net
Metric         Score        Metric         Score        Metric         Score
---------     -------       ---------     -------       ---------     -------
F1 Score       0.933        F1 Score       0.862        F1             0.981
Accuracy       0.938        Accuracy       0.831        Accuracy       0.983
Recall         0.877        Recall         0.757        Recall         0.998
Precision      0.995        Precision      1.000        Precision      0.965
```





Additional metrics for re-run of second experiment

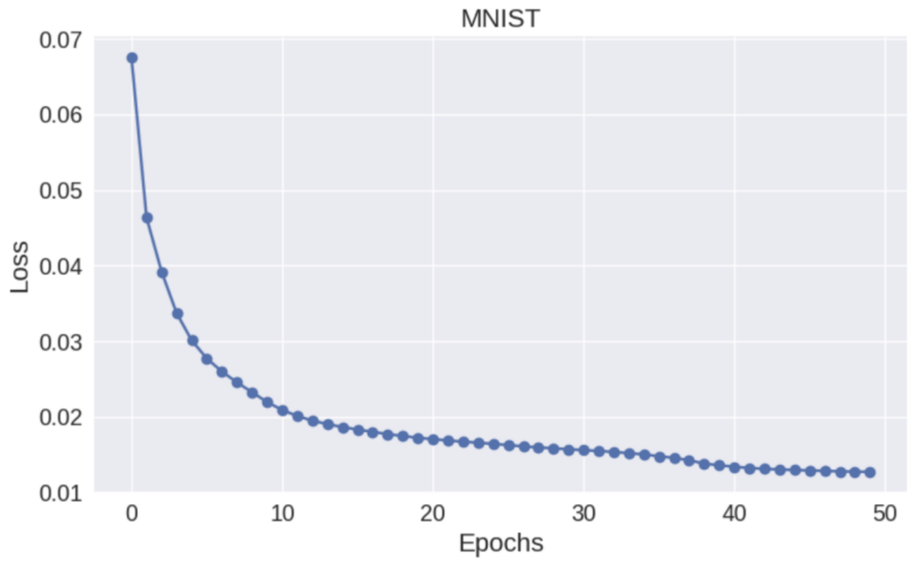




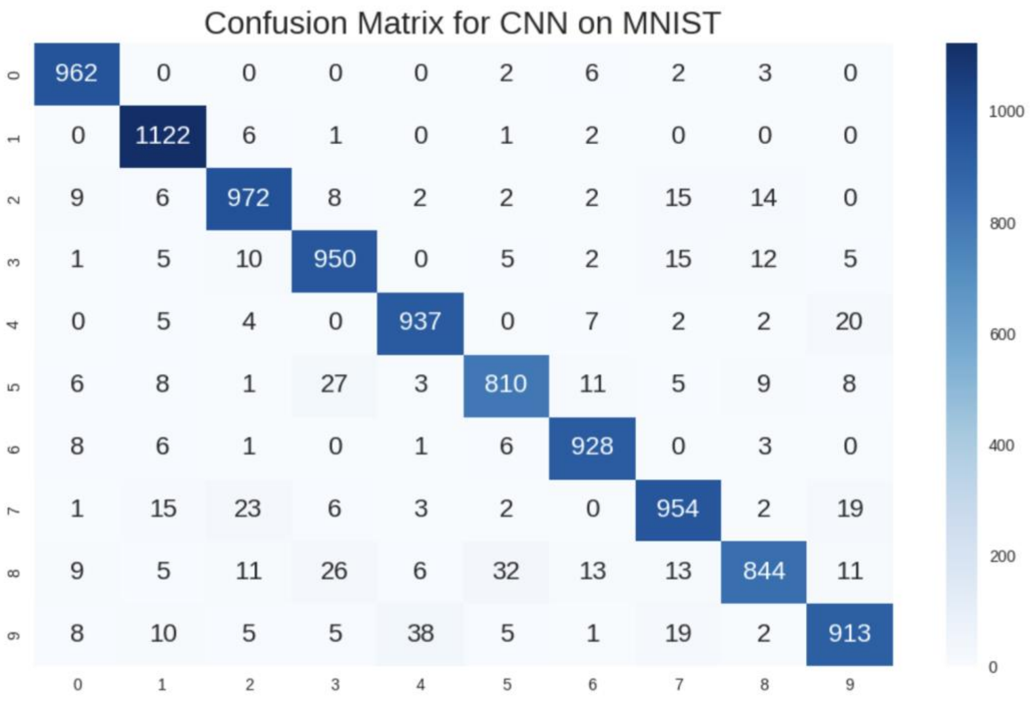
```
---------  -----
F1 Score   0.995
Accuracy   0.983
Recall     0.998
Precision  0.965
---------  -----
```

II. Additional Data from CNN experiments

The vanilla CNN, trained with two layers, ReLu activation function and Adam optimizer, classifies the entire MNIST dataset with excellent scores.



```
Metric        Score
---------   -------
F1 Score          1
Accuracy          1
Recall            1
Precision         1
```



	0	1	2	3	4	5	6	7	8	9
0	962	0	0	0	0	2	6	2	3	0
1	0	1122	6	1	0	1	2	0	0	0
2	9	6	972	8	2	2	2	15	14	0
3	1	5	10	950	0	5	2	15	12	5
4	0	5	4	0	937	0	7	2	2	20
5	6	8	1	27	3	810	11	5	9	8
6	8	6	1	0	1	6	928	0	3	0
7	1	15	23	6	3	2	0	954	2	19
8	9	5	11	26	6	32	13	13	844	11
9	8	10	5	5	38	5	1	19	2	913

References

[1] https://pytorch.org/vision/stable/datasets.html