Multilayer Perceptron and Convolutional Neural Networks on Fashion-MNIST Data

Cecilia Jiang, David Kronish and Matt Ludwig

1 Abstract

In this assignment, we constructed a multilayer perceptron model and used it to classify image data. In experimenting with different network architectures, hyperparameters and regularization we found that training an MLP model with a mini-batch size of 400, a learning rate of .2, with 2 hidden layers, 128 hidden units and with epochs gave the best test accuracy of 87.75%. As a general trend, we found that increasing the number of hidden layers and hidden units increased the accuracy of our model. We also implemented a check gradient function and verified that our calculated were accurate. In addition, we used grid search methods to find the best learning rate, epochs, batch size and test accuracy for MLP architectures with zero, one and two hidden layers. We found that they had test accuracies of 84.12%, 85.70% and 87.74% respectively. We also implemented a $L2$ regularization and found that our MLP model had the highest test accuracy while using a $\lambda = 3$. We also found that the model performed significantly better (with an accuracy difference of around 30%) on the normalized vs. unnormalized images. After this, we implemented a convolutional neural network (CNN) and after using keras' gridsearchCV to tune the hyperparameters we found that it had a best test accuracy of 90.79%. We also experimented with different dropout proportions and found that as a general trend of increasing the dropout proportion from 0 to 1 decreased the test accuracy.

2 Introduction

In this assignment we were tasked with acquiring and preprocessing the Fashion-MNIST dataset, implementing a multilayer perceptron model that could be trained with mini-batch gradient descent and using it to classify images from the Fashion-MNIST dataset. The Fashion-MNIST dataset consists of $70,000$ (28×28) grayscale images of articles of clothing ranging from shirts to sneakers to trousers. The Fashion-MNIST dataset was created in response to the overuse and incredible accuracy that machine learning models had achieved on the original MNIST dataset which is comprised of handwritten digits. As a result, the Fashion-MNIST dataset shares the same image size and structure of training and test splits as the original MNIST dataset.[XRV17] Since its creation in 2017, it has been widely used as a benchmark dataset by machine learning researchers and has been used in thousands of publications. The dataset has been of particular interest to researchers working with Generative adversarial networks (GANs) which are neural networks that can be used for image generation and image processing. For instance, in the paper "On the Tradeoff Between Mode Collapse and Sample Quality in Generative Adversarial Networks" the authors acknowledge how GANs often suffer from the issue of that the images generated by GANs often either lack diversity or are of poor quality. In effort to resolve this issue, the authors proposed different metrics and scores to quantify the quality and diversity of generated images and studied the performances of various GANs on the Fashion MNISt dataset. [AACT18] The Fashion-MNIST dataset has also been used in research regarding Capsule Networks. In the paper "Pushing the Limits of Capsule Networks," the authors use the Fashion-MNIST dataset to test their implementations of Capsule Networks which are neural networks that are similar to convolutional neural networks but that take into account the location of features in an image relative to one another.[NDK21] In this assignment, we implement a multilayer perceptron model and a CNN to the Fashion-MNIST dataset to classify each of the images in the Fashion-MNIST dataset into one the 10 categories of clothing.

3 Datasets

After loading the Fashion-MNIST dataset, we observed that the training and testing data were tensors with dimensions $(60000, 28, 28)$ and $(10000, 28, 28)$ where the first dimensions corresponded to the number of images and the last two to the (28×28) image size. We then flattened these tensors in two-dimenional numpy arrays with dimensions $(60000, 784)$ and $(10000, 784)$ so that we could input them into our multilayer perceptron model. We also standarized the data by subtracting the mean and dividing by the standard deviation. In addition, we also one-hot encoded the labels. We also created another copy of the original training and testing data and similarly changed the dimensions so that we could input the data into the CNN. In addition, we also plotted several of the images to verify that they actually contained articles of clothing.

4 Results

4.1 Results for basic MLP models

4.1.1 Check Gradient Implementation

After implementing our MLP model we implemented a check gradient function in order to verify the correctness of analytically derived gradients. The method we used for checking the gradient is small perturbation: for each linear layer, we add/subtract a small epsilon to the element with position $(0, 0)$ in the derived weight matrix, and calculate the "loss plus" and "loss minus" in these two scenarios. We get the calculated gradient by estimatedGrad = (lossplus − lossminus)/2 ∗ ϵ, and then compare it to the analytically derived gradient. We found that in the case that the gradient is zero, the error is not defined. In other cases, the error is small, usually under 0.01.

4.1.2 Results for MLP with no hidden layer

In order to find the best learning rate and batch size to use when training the MLP with no hidden layer, we implemented a basic grid search from scratch. We found that the model achieved the highest test accuracy of with a learning rate of .1 and a batch size of 400. We also generated plots of the model accuracy as a function of learning rate and batch size, these plots can be seen in Figures 6 and 7 the appendix. Next, using these choices of hyperparameters we compared the model's training and testing accuracies vs. the number of epochs to find the number of epochs to train the model on to ensure that our model didn't overfit the training data. We noticed that after 5 epochs the testing accuracy stabilized around 84.12%. A plot of this is also shown in Figure 1 below.

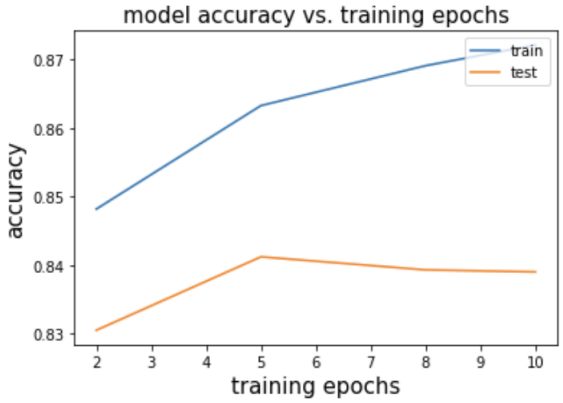

Figure 1: Training and Testing Accuracy vs. Epochs for MLP with no hidden layers

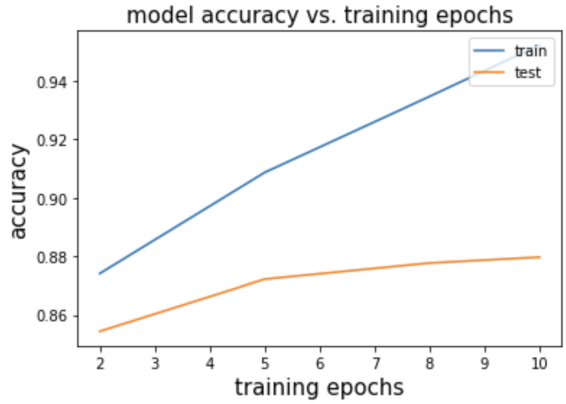

Figure 2: Training and Testing Accuracy vs. Epochs for MLP with 1 hidden layers

4.1.3 Results for MLP with single hidden layer

For our MLP with a single hidden layer with 128 hidden units and ReLu activation functions we also used a grid search and found that the model achieved the highest test accuracy of 85.70% with a learning rate of .2 and batch size of 400 for one epoch. Plots of the learning rate and batch size vs. model accuracy are shown in Figures 8 and 9 in the appendix. Next we used a plot of the training and testing accuracies vs. the number of epochs to find the number of epochs to train the model on to ensure that our model did not overfit the testing data and noticed that after 5 epochs the training accuracy stabilized around 85.70%. This plot is show in 2 above.

4.1.4 Results for MLP with two hidden layers

For our MLP with two hidden layers with 128 hidden units and ReLu activation function we also used a grid search and found that the model achieved the highest test accuracy of 84.65% with a learning rate of 0.2 and batch size of 400. Plots of these hyperparameters vs. model accuracy are shown in 10 and 11 in the appendix. Next we used a plot of the training and testing accuracies vs. the number of epochs to find the number of epochs to train the model on to ensure that our model did not overfit the training data and noticed that after 8 epochs the training accuracy stabilized around 87.74%. This plot is shown in 12 in the appendix.

4.2 Results for MLP with 2 hidden layers with Tanh and Leaky-ReLu activation functions

We also tried running a MLP with 2 hidden layers each having 128 hidden units with the Tanh and Leaky-ReLu activation functions and found that using the Tanh activation function gave 82.47% accuracy and the Leaky-ReLu gave 83.78% accuracy

in comparison to 87.74% accuracy with the ReLu activation functions. We would expect Leaky-ReLu to have better accuracy than ReLu because the derivative of Leaky-ReLu has 1 in the positive part and a small value in the negative part as opposed to ReLu which has 0 in the negative part. In backpropagation this zero in the negative part can result in "dead neurons" as a result of the entire gradient becoming zero. Since Leaky-ReLu avoids this problems, we would expect it to have a higher accuracy than ReLu. We would also expect ReLu to have a higher accuracy than Tanh as the Tanh activation function suffers from the saturating gradient problem which will slow down the weight updates and therefore reduce the model accuracy compared with the ReLu activation function. One reason why our model with Leaky-ReLu had lower accuracy than our model with ReLu could be due to the fact that we trained the model with Leaky-ReLu for less epochs than the model with ReLu.

4.3 Results for MLP with L2-Regularization

We also implemented $L2$ regularization to our MLP model with 2 hidden layers each having 128 units with ReLu activation functions. We adjusted our original MLP model by creating an $L2$ fit method in the MLP class which calculated the cross entropy loss with the regularization term and called an $L2$ backward function in the LinearLayer class which updated the weights using the gradient of the new loss function. After experimenting with lambda, we found that when $lamda = 3$ we achieved 84.88% accuracy. We also observed that as we increased the value of lambda (which is the weight penalty in the cost function) that the accuracy of our model generally improved. We would have liked to experiment with a larger range of lambda values although given the amount of time it took to compile the model, we were unable to. Nonetheless, we found that the model with regularization and a lambda value of $\lambda = 3$ had a similar accuracy to the same MLP architecture without regularization. One reason why the MLP with regularization did not perform drastically better than the MLP without regularization could be because we were unable to train the MLP model for a large number of epochs. We also plotted the accuracy of the regularized MLP model with different values of lambda which is shown in the figure below.

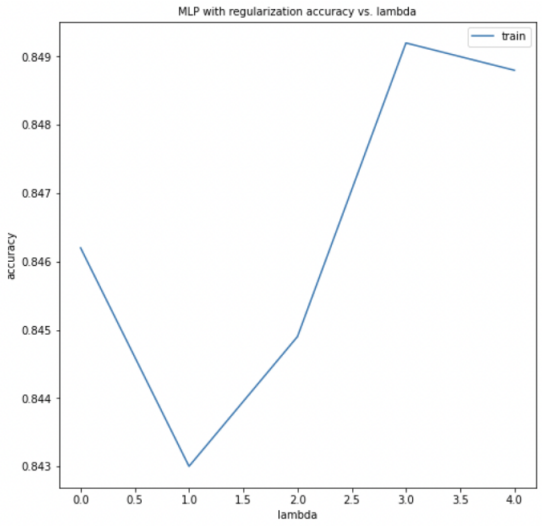

Figure 3: Plot of testing accuracy for MLP with Regularization Accuracy vs. lambda

Figure 4:

4.4 Results for MLP trained on unnormalized images

In addition to implementing L2 regularization for our MLP implementation, we also tried training our implementation using unnormalized data. We found that accuracy of our MLP on the unnormalized images increased as we decreased the learning rate. In particular, we got an accuracy of 10% for a learning rate of .1, an accuracy of 67.43% for a learning rate of .001 and an accuracy of 32.9% for a learning rate .0001 while training the model with 5 epochs. We found that these accuracies are significantly lower than the accuracies that we got for the normalized images using the same MLP architecture.

4.5 Convolution Neural Network Implementation

Using existing libraries from Keras and Tensorflow we implemented a Convolutional Neural Network (ConvNet) with 2 convolutional layers, 2 fully connected layers, 8 filters of size (3×3), a pooling size of 2, a stride size of 1 and no padding. We fit the model using categorical cross entropy as a loss function and adaptive moment estimation (Adam). Using the keras package's gridsearchCV, we found that found that a learning of 0.01 and a batch of 50 which gave a testing accuracy of 87.7%

while the model on a single epoch. To justify our learning rate and batch size, we included the output of the gridsearchCV which shows the accuracies for the different batch sizes and learning rates we tested with. This can be seen in 13 in the appendix. The CNN model with these parameters had a higher accuracy than our MLP implementation. One reason for this is because the filters in the CNN model takes in a tensor as opposed to a numpy array and also can take into account the spatial representation of the image and therefore more accurately predict the class of the image. We then also ran our model with the best learning rate and batch size and plotted the accuracy and loss vs. epochs in order to determine if the model was over fitting. We then found that with 8 epochs the model achieved 90.79% accuracy without over fitting on the training data. This can be seen in figure 5 below.

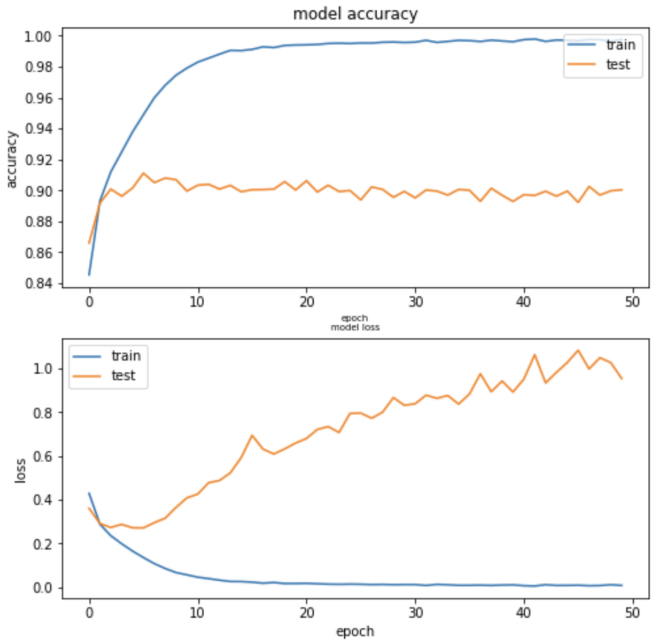

Figure 5: Training and Testing Accuracy vs. Epochs for our CNN model

4.5.1 Varying Dropout rate for CNN

We also tried varying the node dropout proportions in the ConvNet model above and found the accuracy in each case. As a general trend we found that as the node dropout proportion increased, the accuracy of the model decreased. This trend is shown in table 14 in the appendix.

4.6 Results for the best MLP implementation

We also compared several MLP models with a learning rate of .2, batch size of 400, 8 epochs of training and varied the number of hidden units and the network depth. Using a naive grid search, we found as a general trend that as we increased the number of hidden units and the network depth that the accuracy of our MLP model increased. The tables of the MLP accuracies which can be found in the codebase under section 6.). Overall, we found that a MLP with 192 hidden units per layer and a network depth of 3 along with the other hyperparameters we used above gave a the best accuracy of 87.73% and a training accuracy of 92.5%

5 Discussion and Conclusion

Overall, we found that the tensorflow implementation of CNN achieved the highest test accuracy of 90.79% on the Fashion-MNIST dataset. After experimenting with different network architectures, hyperparameters and regularization we found that training an MLP model with a mini-batch size of 400, a learning rate of .2, with 2 hidden layers, 128 hidden units and with epochs gave the best test accuracy of X% on the same dataset. This difference in accuracy illustrates the power of the tensorflow and keras libraries in performing image classification. After implementing a grid search from scratch and using the gridsearchCV from keras' library we also learned how well the keras package works for tuning hyperparamters. The CNN of the tensorflow implementation was also better than the test accuracies of 84.12%, 85.70% and 87.74% that we found for training an MLP model with the best hyperparamters that we got from our gridsearch. In addition we noticed that the model performed significantly better (with an accuracy difference of around 30%) on the normalized vs. unnormalized images. Finally, we also experimented with varying the dropout proportions of the CNN and found that as a general trend

of increasing the dropout proportion from 0 to 1 decreased the test accuracy. Further investigations could implement various data augmentation techniques on the Fashion-MNIST in order to increase the size of the training data. Another thing further investigation could do is implement cross validation as well as experiment with different optimizers and their parameters like momentum.

6 Statement of Contributions

Matt: Task 1, assisted with implementation and debugging of task 2. Task 3.3, 3.5, 3.2, wrote whole report, 3.4, regularization

Cecilia: Check gradient, added paragraph 4.1.1 in the report on check gradient, task 3.4

David: Task 2 implementation of MLP, gridsearch for CNN on 3.5, gridsearch implementation and plot generation for Task 3, experimented with finding the best MLP architecture.

References

[AACT18] Sudarshan Adiga, Mohamed Adel Attia, Wei-Ting Chang, and Ravi Tandon. On the tradeoff between mode collapse and sample quality in generative adversarial networks. In *2018 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 1184–1188, 2018.

[NDK21] Prem Nair, Rohan Doshi, and Stefan Keselj. Pushing the limits of capsule networks, 2021.

[XRV17] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

7 Appendix - (plots of hyperparameters)

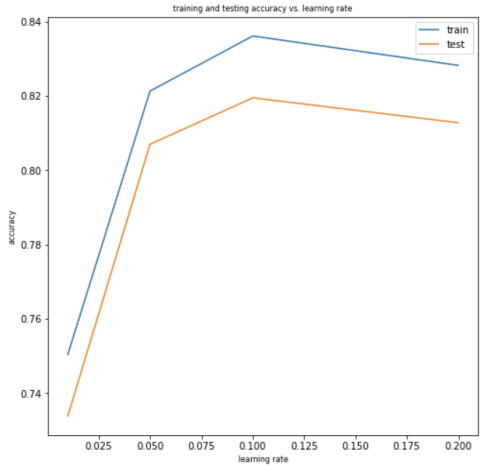

Figure 6: Learning rate vs. Accuracy plot for MLP with no hidden layer

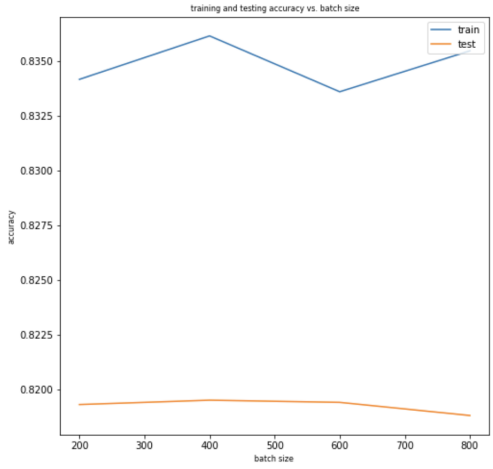

Figure 7: Batch-size vs. Accuracy plot for MLP with no hidden layers

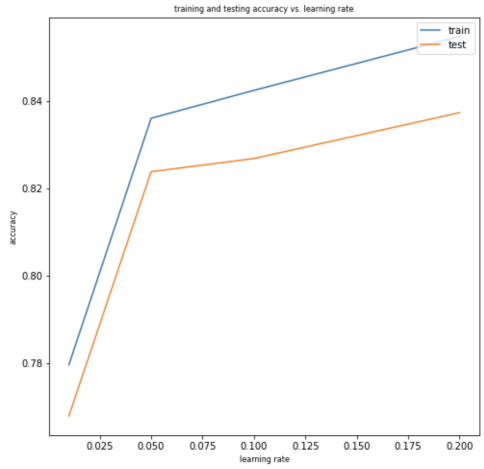

Figure 8: Learning rate vs. Accuracy plot for MLP with 1 hidden layer

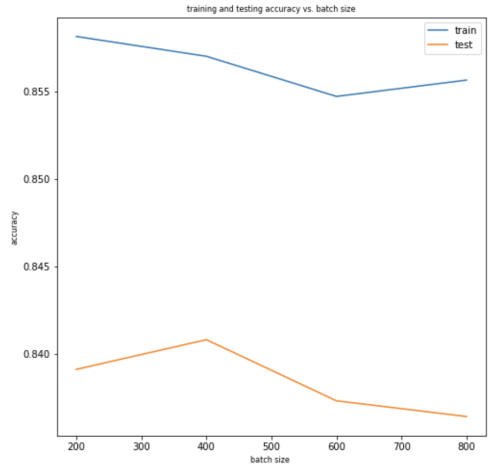

Figure 9: Batch-size vs. Accuracy plot for MLP with 1 hidden layers

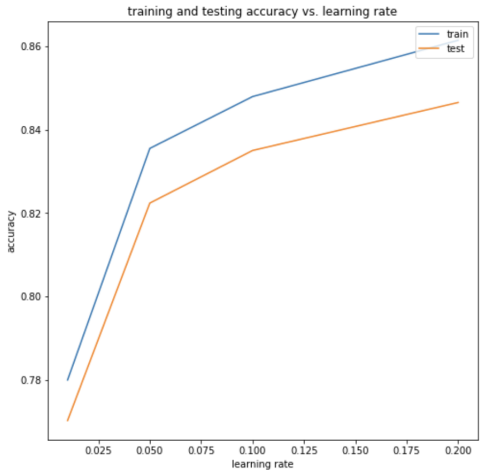

Figure 10: Learning rate vs. Accuracy plot for MLP with 2 hidden layers

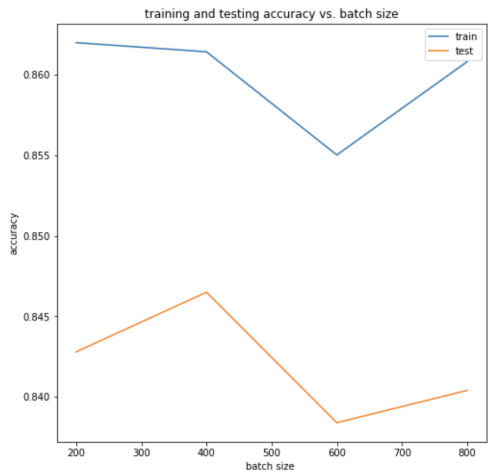

Figure 11: Batch-size vs. Accuracy plot for MLP with 2 hidden layers

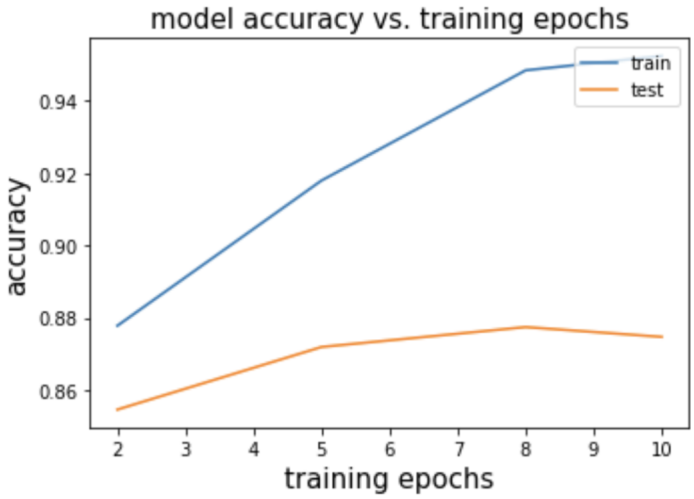

Figure 12: Training and Testing Accuracy vs. Epochs for MLP with 2 hidden layers

```
Best: 0.877033 using {'batch_size': 50, 'model__learning_rate': 0.001}
0.100133 (0.001165) with: {'batch_size': 50, 'model__learning_rate': 0.1}
0.101250 (0.000885) with: {'batch_size': 50, 'model__learning_rate': 0.05}
0.828150 (0.018715) with: {'batch_size': 50, 'model__learning_rate': 0.02}
0.862967 (0.008557) with: {'batch_size': 50, 'model__learning_rate': 0.01}
0.877033 (0.006447) with: {'batch_size': 50, 'model__learning_rate': 0.001}
0.099867 (0.000710) with: {'batch_size': 100, 'model__learning_rate': 0.1}
0.099733 (0.001805) with: {'batch_size': 100, 'model__learning_rate': 0.05}
0.693717 (0.297828) with: {'batch_size': 100, 'model__learning_rate': 0.02}
0.873000 (0.007631) with: {'batch_size': 100, 'model__learning_rate': 0.01}
0.869333 (0.006442) with: {'batch_size': 100, 'model__learning_rate': 0.001}
0.100467 (0.001268) with: {'batch_size': 200, 'model__learning_rate': 0.1}
0.098133 (0.002070) with: {'batch_size': 200, 'model__learning_rate': 0.05}
0.856250 (0.008870) with: {'batch_size': 200, 'model__learning_rate': 0.02}
0.859200 (0.008459) with: {'batch_size': 200, 'model__learning_rate': 0.01}
0.861600 (0.007117) with: {'batch_size': 200, 'model__learning_rate': 0.001}
0.096350 (0.001715) with: {'batch_size': 300, 'model__learning_rate': 0.1}
0.098617 (0.001193) with: {'batch_size': 300, 'model__learning_rate': 0.05}
0.712200 (0.303661) with: {'batch_size': 300, 'model__learning_rate': 0.02}
0.870067 (0.007394) with: {'batch_size': 300, 'model__learning_rate': 0.01}
0.858200 (0.009390) with: {'batch_size': 300, 'model__learning_rate': 0.001}
0.099700 (0.002618) with: {'batch_size': 600, 'model__learning_rate': 0.1}
0.100483 (0.002497) with: {'batch_size': 600, 'model__learning_rate': 0.05}
0.844750 (0.008580) with: {'batch_size': 600, 'model__learning_rate': 0.02}
0.865867 (0.003351) with: {'batch_size': 600, 'model__learning_rate': 0.01}
0.829633 (0.011830) with: {'batch_size': 600, 'model__learning_rate': 0.001}
```

Figure 13: Table of CNN accuracy for different learning rates and batch sizes

```
938/938 [==============================] - ETA: 0s - loss: 0.4864 - accuracy: 0.8274
Epoch 1: val_loss improved from inf to 0.37036, saving model to mnist_2_layer_ConvNN.hdf5
938/938 [==============================] - 46s 47ms/step - loss: 0.4864 - accuracy: 0.8274 - val_loss: 0.3704 - val_accuracy: 0.8688
937/938 [==============================>.] - ETA: 0s - loss: 0.4920 - accuracy: 0.8260
Epoch 1: val_loss improved from inf to 0.36464, saving model to mnist_2_layer_ConvNN.hdf5
938/938 [==============================] - 49s 52ms/step - loss: 0.4918 - accuracy: 0.8261 - val_loss: 0.3646 - val_accuracy: 0.8693
938/938 [==============================] - ETA: 0s - loss: 0.4957 - accuracy: 0.8232
Epoch 1: val_loss improved from inf to 0.37919, saving model to mnist_2_layer_ConvNN.hdf5
938/938 [==============================] - 49s 52ms/step - loss: 0.4957 - accuracy: 0.8232 - val_loss: 0.3792 - val_accuracy: 0.8665
937/938 [==============================>.] - ETA: 0s - loss: 0.5173 - accuracy: 0.8143
Epoch 1: val_loss improved from inf to 0.43195, saving model to mnist_2_layer_ConvNN.hdf5
938/938 [==============================] - 55s 58ms/step - loss: 0.5171 - accuracy: 0.8144 - val_loss: 0.4319 - val_accuracy: 0.8349
937/938 [==============================>.] - ETA: 0s - loss: 0.5404 - accuracy: 0.8021
Epoch 1: val_loss improved from inf to 0.40010, saving model to mnist_2_layer_ConvNN.hdf5
938/938 [==============================] - 50s 53ms/step - loss: 0.5405 - accuracy: 0.8021 - val_loss: 0.4001 - val_accuracy: 0.8558
937/938 [==============================>.] - ETA: 0s - loss: 0.5482 - accuracy: 0.8004
Epoch 1: val_loss improved from inf to 0.39607, saving model to mnist_2_layer_ConvNN.hdf5
938/938 [==============================] - 50s 52ms/step - loss: 0.5481 - accuracy: 0.8004 - val_loss: 0.3961 - val_accuracy: 0.8590
937/938 [==============================>.] - ETA: 0s - loss: 0.5854 - accuracy: 0.7867
Epoch 1: val_loss improved from inf to 0.41535, saving model to mnist_2_layer_ConvNN.hdf5
938/938 [==============================] - 49s 52ms/step - loss: 0.5853 - accuracy: 0.7867 - val_loss: 0.4154 - val_accuracy: 0.8504
938/938 [==============================] - ETA: 0s - loss: 0.6582 - accuracy: 0.7534
Epoch 1: val_loss improved from inf to 0.45000, saving model to mnist_2_layer_ConvNN.hdf5
938/938 [==============================] - 53s 56ms/step - loss: 0.6582 - accuracy: 0.7534 - val_loss: 0.4500 - val_accuracy: 0.8334
937/938 [==============================>.] - ETA: 0s - loss: 0.7886 - accuracy: 0.7032
Epoch 1: val_loss improved from inf to 0.53113, saving model to mnist_2_layer_ConvNN.hdf5
938/938 [==============================] - 91s 96ms/step - loss: 0.7885 - accuracy: 0.7033 - val_loss: 0.5311 - val_accuracy: 0.7966
937/938 [==============================>.] - ETA: 0s - loss: 1.7231 - accuracy: 0.3547
Epoch 1: val_loss improved from inf to 1.09252, saving model to mnist_2_layer_ConvNN.hdf5
938/938 [==============================] - 46s 49ms/step - loss: 1.7228 - accuracy: 0.3548 - val_loss: 1.0925 - val_accuracy: 0.6329
```

Figure 14: Table of CNN accuracy vs. dropout proportion