Part A:

I brought in the dataset using scipy.io.mio.loadmat() and named is data. from there the data was split into two categories, 'images' and 'labels'. From there the data was split again into testing and training data and testing and training labels using the train\_split\_test function that made 20 percent of the dataset the testing data and the other 80 percent the training data.

Part B:

I took the mean subtraction and normalization of the data. The mean subtraction was done by finding the mean of the testing and training data and subtracting that from training and testing data. To normalize the data I took the standard deviation of the two sets of images and divided that by two sets of images after the main subtraction was done.

|  |
| --- |
| *#PART A* **from** \_\_future\_\_ **import** absolute\_import, division, print\_function, unicode\_literals **from** os.path **import** dirname, join **as** pjoin **import** scipy.io **as** sio **import** tensorflow **as** tf **from** tensorflow **import** keras **import** numpy **as** np **import** matplotlib.pyplot **as** plt **from** sklearn.model\_selection **import** train\_test\_split **from** tensorflow.keras **import** regularizers  data = sio.loadmat(**"notMNIST\_small.mat"**) *#find the name of the columns to make an array for the images and labels* print(data.keys()) print(**"number of images: "**, len(data))  x = data[**'images'**] x = x.transpose() print(x.shape) y = data[**'labels'**]  x\_train, x\_test, y\_train, y\_test = train\_test\_split(x, y, test\_size=0.2, random\_state=42)  *#PART B #mean subtraction* x\_train = x\_train - x\_train.mean() x\_test = x\_test - x\_test.mean() *#normalization* x\_train = x\_train/x\_train.std() x\_test = x\_test/x\_test.std()  print(**"Training data stats:\n"**, **"min: "**,x\_train.min(), **"max: "**, x\_train.max(), **"mean: "**, x\_train.mean(), **"std: "**, x\_train.std()) print(**"Testing data stats:\n"**, **"min: "**,x\_test.min(), **"max: "**, x\_test.max(), **"mean: "**, x\_test.mean(), **"std: "**, x\_test.std())   model = tf.keras.Sequential() model.add(tf.keras.layers.Flatten(input\_shape=(28, 28))) *#PART C, D, & Initialization, number of layers, and activation function* model.add(tf.keras.layers.Dense(128, activation=**'relu'**, kernel\_initializer=**"glorot\_normal"**)) *#PART G regularization techniques* model.add(tf.keras.layers.Dense(10, activation=**"softmax"**, activity\_regularizer=regularizers.l1(l=.01)))  *#PART E GRADIENT OPTIMIZATION TECHNIQUE* model.compile(optimizer=**'adam'**, loss=**'sparse\_categorical\_crossentropy'**, metric=[**'accuracy'**])  model.fit(x\_train, y\_train, epochs = 10)  model.evaluate(x\_test, y\_test, verbose=2) model.summary() |
|  |
|  |

Part C:

part. i: The model was built using the keras framework, specifically the Sequential() model. i first build the model by creating a variable called model = tf.keras.model.Sequential(), and from there I add layers by using model.add. to initialize the glorot initializer, in the first Dense() layer i used the kernel\_initializer and the "glorot\_normal" initializer. Below are the loss results and model summary of the model using the glorot initializer

|  |
| --- |
|  |
|  |

part ii: like the glorot initializer, the He initializer was accepted in to the keras API, so I only had to use its function name “he\_normal” to use it in the model.

|  |
| --- |
|  |

Part D:

Exp.1: The first configuration of the network is with one hidden layer. That layer uses relu activation and He initialization. My choice to use the He initialization is that it seems to be an improvement on the Xavier initializer, as it has similar results for a normal neural network, but the deeper the network becomes it still maintains its performance. I will be using it for the remainder of the experiments. I choose this as it is the most common and simply configuration that I have seen being used.

|  |
| --- |
| model = tf.keras.Sequential() model.add(tf.keras.layers.Flatten(input\_shape=(28, 28))) *#PART C, D, & Initialization, number of layers, and activation function* model.add(tf.keras.layers.Dense(128, activation=**'relu'**, kernel\_initializer=**"he\_normal"**)) *#PART G regularization techniques* model.add(tf.keras.layers.Dense(10, activation=**"softmax"**, activity\_regularizer=regularizers.l1(l=.01)))  *#PART E GRADIENT OPTIMIZATION TECHNIQUE* model.compile(optimizer=**'adam'**, loss=**'sparse\_categorical\_crossentropy'**, metric=[**'accuracy'**]) |
|  |

Exp. 2: I n the second experiment I increased the amount of layers to 10. This was an arbitrary number in that the average number of layers I saw being used for teaching purposes has been just 2-4, but now industry level neural networks number above 100, 10 seemed appropriate for the power that my computer has and seems sufficiently larger for comparison with the original neural network that was created. I also increased the number of nodes to 200 and unlike what I expected, the loss was higher than the original neural network with a loss of 1.30002, compared to the original neural network with a loss of .2270

|  |
| --- |
| Final Epoch: |
| Model summary: |

Part E:

Exp.1: For this part of the experiment I went back to the original format of the neural network, which is a single hidden layer and the He initializer. Originally I used the ADAM optimization in the beginning since it is the most commonly used optimization function

|  |
| --- |
| model = tf.keras.Sequential() model.add(tf.keras.layers.Flatten(input\_shape=(28, 28))) *#PART C, D, & Initialization, number of layers, and activation function* model.add(tf.keras.layers.Dense(128, activation=**'relu'**, kernel\_initializer=**"he\_normal"**)) *#PART G regularization techniques* model.add(tf.keras.layers.Dense(10, activation=**"softmax"**, activity\_regularizer=regularizers.l1(l=.01)))  *#PART E GRADIENT OPTIMIZATION TECHNIQUE* model.compile(optimizer=**'adam'**, loss=**'sparse\_categorical\_crossentropy'**, metric=[**'accuracy'**]) |
|  |

Exp. 2: For the second experiement I choose the AdaDelta optimizer. The AdaDelta optimizer has a continual decay of the learning rate throughout the training which, in theory, helps in keeping the model from overfitting as the error reduces. The results show that. In training the loss was 1.2068, but in the testing the loss more than doubles to 2.4873. additionally I did find out that making sure to set the learning rate, and rho myself made the model perform better, as I redid the experiment with standard notation (model.compile(optimizer=**"Adadelta"**, loss=**'sparse\_categorical\_crossentropy'**, metric=[**'accuracy'**])) and the loss for testing was even high at 3.8178 I assume these high loss rates are all due to the lack of time to converge as there are only 10 epochs in my experiment neural network

|  |
| --- |
| model = tf.keras.Sequential() model.add(tf.keras.layers.Flatten(input\_shape=(28, 28))) *#PART C, D, & Initialization, number of layers, and activation function* model.add(tf.keras.layers.Dense(128, activation=**'relu'**, kernel\_initializer=**"he\_normal"**)) *#PART G regularization techniques* model.add(tf.keras.layers.Dense(10, activation=**"softmax"**, activity\_regularizer=regularizers.l1(l=.01)))  *#PART E GRADIENT OPTIMIZATION TECHNIQUE* model.compile(tf.keras.optimizers.Adadelta(learning\_rate=0.001, rho=0.95, epsilon=1e-07, name=**"Adadelta"**),  loss=**'sparse\_categorical\_crossentropy'**, metric=[**'accuracy'**]) |
|  |

Exp. 3: For the 3rd experiment I used the RMSProp optimizer because it also seems to have a good algorithm with appropriate moving parts. The square of gradients has a moving average and the gradient is divided by the root of this average. In theory, it seems that the gradient will adapt to the error situation at each run of the neural network. The loss in the testing phase was 0.3542

|  |
| --- |
| model = tf.keras.Sequential() model.add(tf.keras.layers.Flatten(input\_shape=(28, 28))) *#PART C, D, & Initialization, number of layers, and activation function* model.add(tf.keras.layers.Dense(128, activation=**'relu'**, kernel\_initializer=**"he\_normal"**)) *#PART G regularization techniques* model.add(tf.keras.layers.Dense(10, activation=**"softmax"**, activity\_regularizer=regularizers.l1(l=.01)))  *#PART E GRADIENT OPTIMIZATION TECHNIQUE* model.compile(tf.keras.optimizers.RMSprop(learning\_rate=0.002, rho=0.87, epsilon=1e-07, centered=**False**, name=**"RMSprop"**),   loss=**'sparse\_categorical\_crossentropy'**, metric=[**'accuracy'**]) |
|  |

Part F:

exp.1:for the first activation function I used the LeakyReLu activation function

|  |
| --- |
| model = tf.keras.Sequential() model.add(tf.keras.layers.Flatten(input\_shape=(28, 28))) *#PART C, D, & Initialization, number of layers, and activation function* model.add(tf.keras.layers.Dense(128,tf.keras.layers.LeakyReLU(alpha=0.3), kernel\_initializer=**"glorot\_normal"**)) *#PART G regularization techniques* model.add(tf.keras.layers.Dense(10, activation=**"softmax"**, activity\_regularizer=regularizers.l1(l=.01)))  *#PART E GRADIENT OPTIMIZATION TECHNIQUE* model.compile(optimizer=**'adam'**, loss=**'sparse\_categorical\_crossentropy'**, metric=[**'accuracy'**]) |
|  |

Exp. 2: PReLu

|  |
| --- |
| model = tf.keras.Sequential() model.add(tf.keras.layers.Flatten(input\_shape=(28, 28))) *#PART C, D, & Initialization, number of layers, and activation function* model.add(tf.keras.layers.Dense(128,tf.keras.layers.PReLU(  alpha\_initializer=**"zeros"**, alpha\_regularizer=**None**, alpha\_constraint=**None**, shared\_axes=**None**), kernel\_initializer=**"glorot\_normal"**)) *#PART G regularization techniques* model.add(tf.keras.layers.Dense(10, activation=**"softmax"**, activity\_regularizer=regularizers.l1(l=.01)))  *#PART E GRADIENT OPTIMIZATION TECHNIQUE* model.compile(optimizer=**'adam'**, loss=**'sparse\_categorical\_crossentropy'**, metric=[**'accuracy'**]) |
|  |

Part G:

Exp.1 I used the l1+l2 regularization in the activity regularizer for the first experiment

|  |
| --- |
| model = tf.keras.Sequential() model.add(tf.keras.layers.Flatten(input\_shape=(28, 28))) *#PART C, D, & Initialization, number of layers, and activation function* model.add(tf.keras.layers.Dense(128, activation=**'relu'**, kernel\_initializer=**"glorot\_normal"**)) *#PART G regularization techniques* model.add(tf.keras.layers.Dense(10, activation=**"softmax"**, activity\_regularizer=regularizers.l2(1e-5))) |
|  |

Exp.2 I used the l2 regularization for the activity regularizer in the second experiment

|  |
| --- |
| model = tf.keras.Sequential() model.add(tf.keras.layers.Flatten(input\_shape=(28, 28))) *#PART C, D, & Initialization, number of layers, and activation function* model.add(tf.keras.layers.Dense(128, activation=**'relu'**, kernel\_initializer=**"glorot\_normal"**)) *#PART G regularization techniques* model.add(tf.keras.layers.Dense(10, activation=**"softmax"**, activity\_regularizer=regularizers.l1\_l2(l1=1e-5, l2=1e-4))) |
|  |