Deep Learning for Image Analysis

Mohammed Al-Jaff

Assignment 2:
Implementing a Minimal Neural Network Modul Equipped With Stochastic Gradient Decent Learning in python

Date: 2021-April-22

Table of Content

---

1. Exercise 1.1 - Deriving an expression for the gradient of the cost- and loss-functions.

In this section we will build up notation and expression for the outputs of a neural network in order to achive finding the the gradients of a cost functions J with respect to the network parameters for an M class classification task.

We begin with the expression for the 'product sum' output of the m th node in the k th layer of a multiple layer netowrk as the simple linear combination of the corresponding weights of the node, $\mathbf{w}_m^k$ and the output of the previous layer,$\mathbf{o}^{k-1}$, :

$$z^k{}_{n,m} = \left(\mathbf{w}_m^k\right)^\top \mathbf{o}^{k-1} + b_m = b_m^k + \sum_{l=1}^{L} w_{m,l}^k o_{n,l}^{k-1} \qquad (1)$$

Note 1: The n supcript indicates that this expression it specific for the outputs due to the n 'th data point $\mathbf{x}_n$
Note 2: When we are dealing with the first hidden layer of the network, $k = 1$, then the previous layer is simply the input data-vector $\mathbf{x}$

Next, we now focus on the partial derivatives for the above expression with respect to the weights. That is, we want to get at $\frac{\partial z_{n,m}}{\partial w_{m,l}}$ and $\frac{\partial z_m}{\partial b_m^k}$.

These are simply:

$$\frac{\partial z_{n,m}}{\partial w_{m,l}} = \frac{\partial z_m^K}{\partial w_{m,l}} = o_{n,l}^{K-1} \qquad (2)$$

$$\frac{\partial z_m}{\partial b_m^k} = \frac{\partial z_m^K}{\partial b_m^k} = 1 \qquad (3)$$

which we recall deriving and using from the previous assignment (Hand-in assignment 1: Implementing a linear regression model)

In a M-class classification task, the relevant loss function we use here is the cross-entropy loss:

$$L = \ln\left(\sum_{m=1}^{M} e^{z_m}\right) - \sum_{m=1}^{M} y_m z_m$$

Where the z_m are the elements of the last-layer/output of the network for $\mathbf{x}$. The y_m 's are the elements of the one-hot encoded label vector for the data point $\mathbf{x}$, A member of class m will have a one-hot encoded label-vector with all zeros expect at the m-th index where there is a one.

We are interested in the partial derivatives of the above loss function with respect to the network outputs, z_m, that is, we want to arrive at an expression for \frac{\partial L}{\partial z_m}:

$$\frac{\partial L}{\partial z_m} = \frac{\partial}{\partial z_m}\left(\ln\left(\sum_{m=1}^{M} e^{z_m}\right)\right) - \frac{\partial}{\partial z_m}\left(\sum_{m=1}^{M} y_m z_m\right)$$

$$\frac{\partial L}{\partial z_m} = \frac{e^{z_m}}{\sum_{m=1}^{M} e^{z_m}} - 1_{y=m} \qquad (4)$$

Note 1: We have used implicit differention here for the first term, ie $\frac{1}{dx} ln(y(x)) = \frac{y'(x)}{y(x)}$.
Note 2: The second term is the indicator function which is 1 when the true label is the same as the math element and 0 otherwise.

The cost function (more properly called the empirical risk function) the average loss given our network parameters that generated the class predictions for our data. The cost is the average loss over the N data-points:

$$J = \frac{1}{N}\sum_{n=1}^{N} L_n \qquad (5)$$

In our optimzation/learning/training phase, we update our network weights with respect to the gradient of the cost function:

$$w_{new} \leftarrow w_{old} - \alpha \nabla_w J$$

As such, we are interested in the partial derivatives of the cost function w.r.t all the network weights $w_{m,j}$. In other words, we went to get at expressions for $\frac{\partial J}{\partial w_{m_j}}$

$$\frac{\partial J}{\partial w_{m_j}} = \frac{1}{N}\sum_{n=1}^{N}\frac{\partial L_n}{\partial w_{m_j}} = \frac{1}{N}\sum_{n=1}^{N}\frac{\partial L_n}{\partial z_{n,m}}\frac{\partial z_{n,m}}{\partial w_{m,j}} \qquad (6)$$

.... and $\frac{\partial J}{\partial b_m}$

$$\frac{\partial J}{\partial b_m} = \frac{1}{N}\sum_{n=1}^{N}\frac{\partial L_n}{\partial b_m} = \frac{1}{N}\sum_{n=1}^{N}\frac{\partial L_n}{\partial z_{n,m}}\frac{\partial z_{n,m}}{\partial b_m} \qquad (7)$$

Note 1: Implicit differentiation at play here **Note 2:** The expression inside the summand for both (6) and (7) are exactly the ones we found in (2), (3) and (4)

---

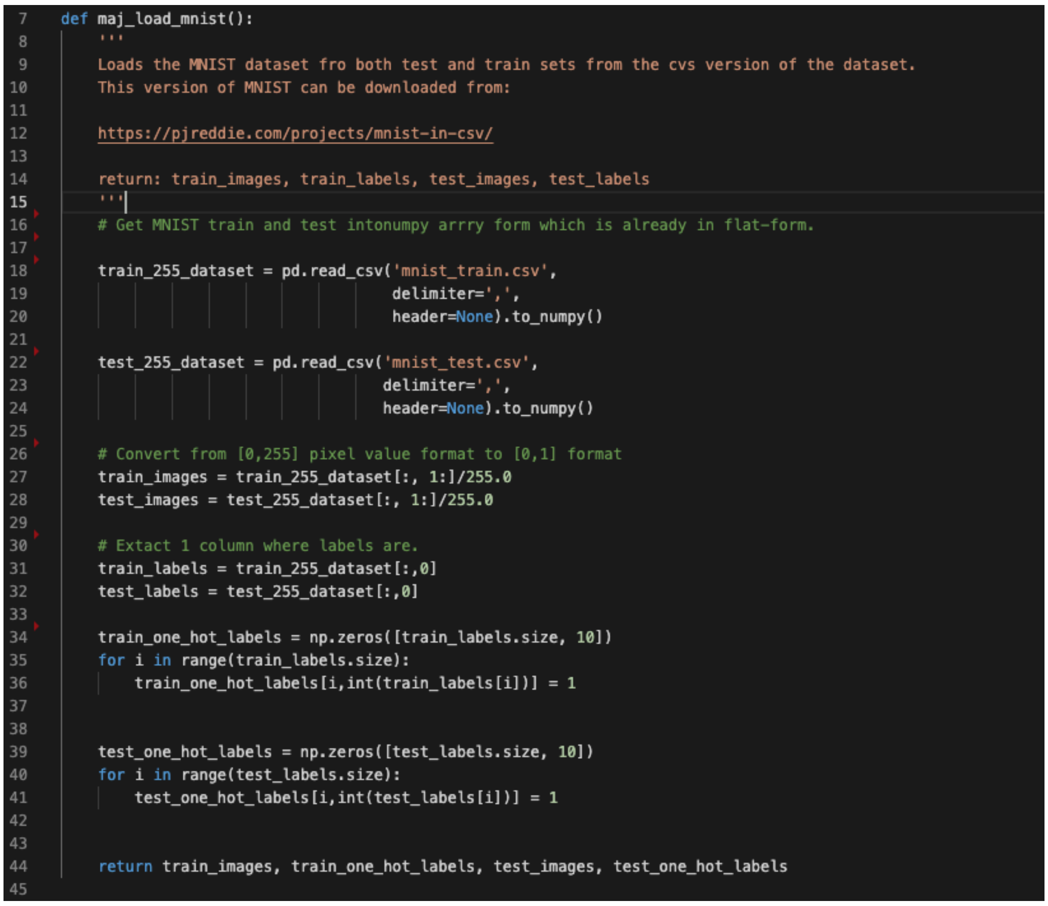
2. Exercise 1.2 - Implementation Highlights

In this section I will go through some of my implementation code and describe some of the non-trivial parts and the parts I opted to solve some of the subtasks my own specific way. The more straightforward elements and function of the implementation are thus not included in this report but are of course in the attached and provided python files which hopefully are well commented enough.

2.1. Alternative MNIST source.

To start with, there was a slight issue downloading the original course provided MNIST dataset the first few days, so to get started early I downloaded and used an alternativ source of the mist dataset that is already in csv format from https://pjreddie.com/projects/mnist-in-csv/

Content-wise, this version is identical to original dataset except that the images are already represented as rows in a csv file. Due to this, I hade to write my own 'load_data' function which reads the csv file, converts the images into [0,1] values and one-hot encodes the labels for each image and returns back test and train data and label matrices:

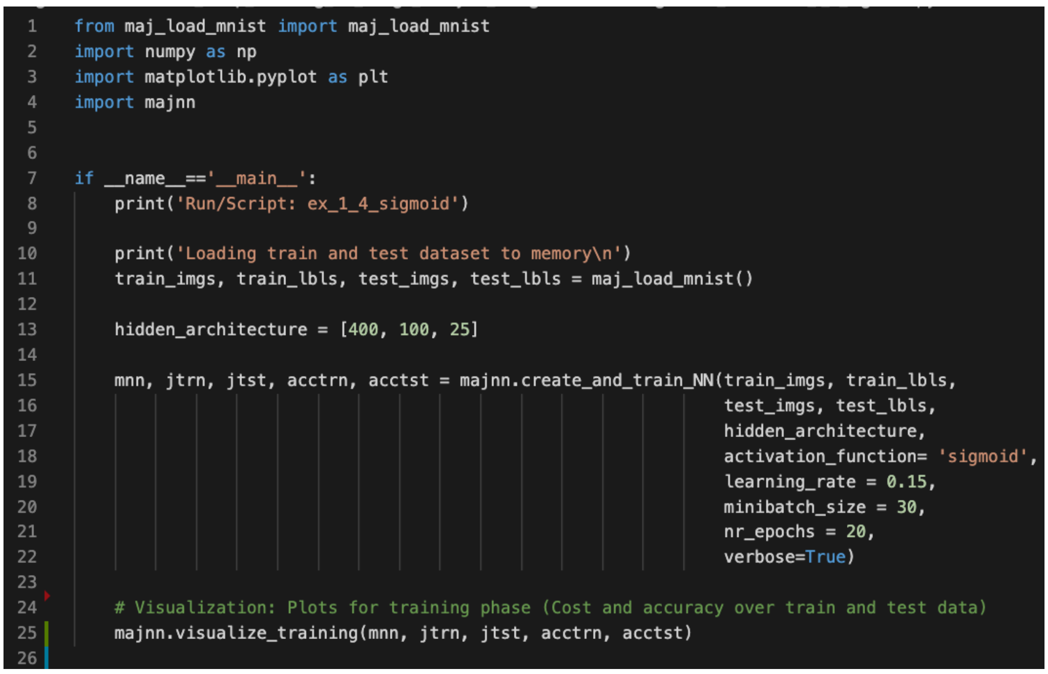


```python
def maj_load_mnist():
    '''
    Loads the MNIST dataset fro both test and train sets from the cvs version of the dataset.
    This version of MNIST can be downloaded from:

    https://pjreddie.com/projects/mnist-in-csv/

    return: train_images, train_labels, test_images, test_labels
    '''
    # Get MNIST train and test intonumpy arrry form which is already in flat-form.

    train_255_dataset = pd.read_csv('mnist_train.csv',
                                    delimiter=',',
                                    header=None).to_numpy()

    test_255_dataset = pd.read_csv('mnist_test.csv',
                                   delimiter=',',
                                   header=None).to_numpy()

    # Convert from [0,255] pixel value format to [0,1] format
    train_images = train_255_dataset[:, 1:]/255.0
    test_images = test_255_dataset[:, 1:]/255.0

    # Extact 1 column where labels are.
    train_labels = train_255_dataset[:,0]
    test_labels = test_255_dataset[:,0]

    train_one_hot_labels = np.zeros([train_labels.size, 10])
    for i in range(train_labels.size):
        train_one_hot_labels[i,int(train_labels[i])] = 1


    test_one_hot_labels = np.zeros([test_labels.size, 10])
    for i in range(test_labels.size):
        test_one_hot_labels[i,int(test_labels[i])] = 1


    return train_images, train_one_hot_labels, test_images, test_one_hot_labels
```

2.2. The practical use case.

I chose to approach this assignment as if writing and constructing a python library. The two major elements are i) a 'train_and_create' function that takes in a bunch of user-specified arguments which it useses to instantiate, train and return a neural network object back to the user. and ii) The user can then use this network object to predict other data, see and extract its weights and even further train on additional data. In addition to the neural network object, additional training phase data is returned back which can be visualised and plotted with the function 'visualise_training' in the same majnn.py module/file.

The below code snippet would be how one would typically use the module. Here we specify MNIST as our train and test data (line 11) and we specify the hidden layer architecture(line 13), ie the number of layers and nodes in each hidden layer we want our neural network to have.

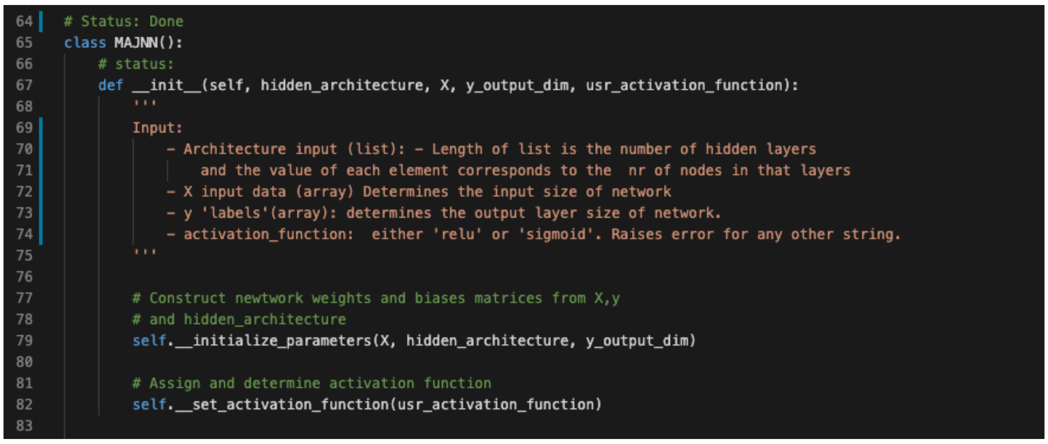


```python
from maj_load_mnist import maj_load_mnist
import numpy as np
import matplotlib.pyplot as plt
import majnn


if __name__=='__main__':
    print('Run/Script: ex_1_4_sigmoid')

    print('Loading train and test dataset to memory\n')
    train_imgs, train_lbls, test_imgs, test_lbls = maj_load_mnist()

    hidden_architecture = [400, 100, 25]

    mnn, jtrn, jtst, acctrn, acctst = majnn.create_and_train_NN(train_imgs, train_lbls,
                                                                test_imgs, test_lbls,
                                                                hidden_architecture,
                                                                activation_function= 'sigmoid',
                                                                learning_rate = 0.15,
                                                                minibatch_size = 30,
                                                                nr_epochs = 20,
                                                                verbose=True)

    # Visualization: Plots for training phase (Cost and accuracy over train and test data)
    majnn.visualize_training(mnn, jtrn, jtst, acctrn, acctst)
```

2.3. The Network model object and the main 'train_mode' function.

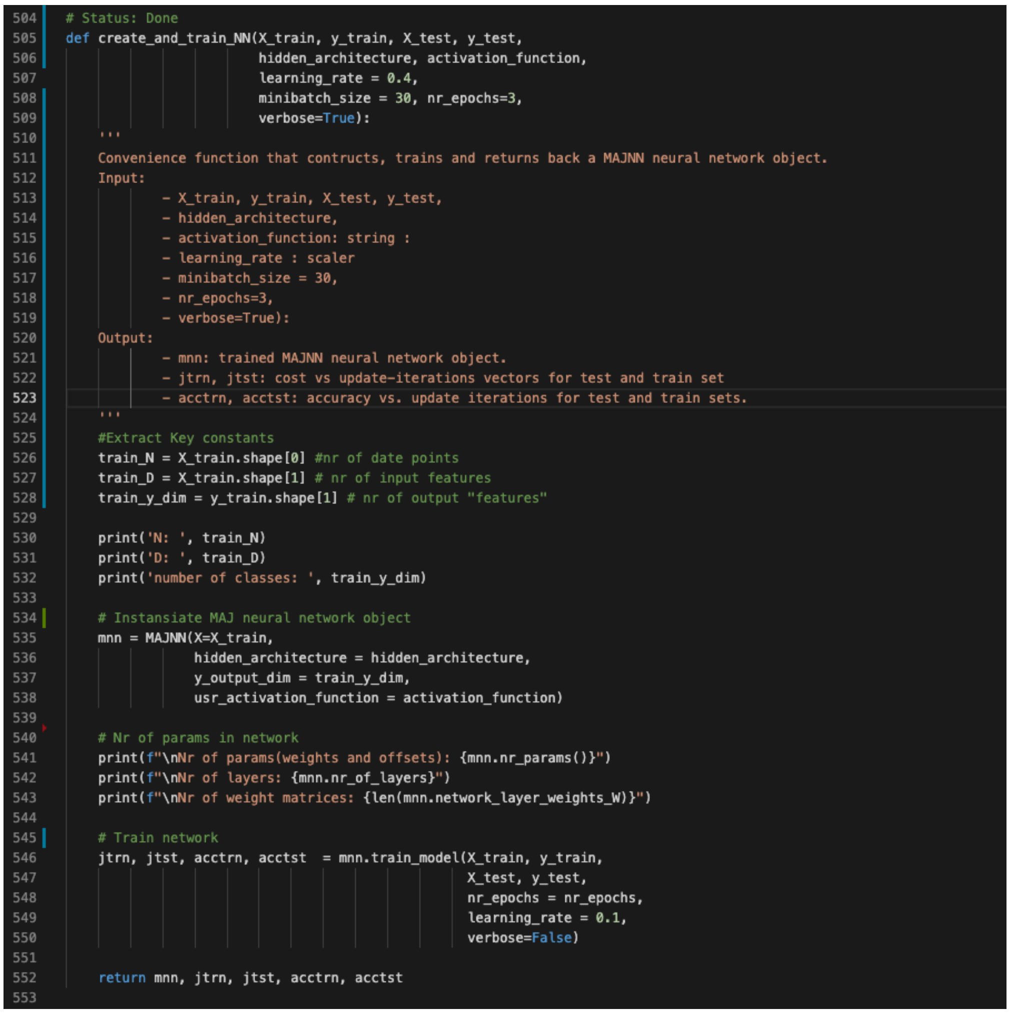
The MAJNN object. My implementation approach involves representing neural network in an object-oriented-way with internal states and internal methods/functions that transform these states. The states here are among others key object attributes variables like the i) architecture of the network , the number of hidden layers and the number of nodes in each of these, ii) the weight-matrices and bias vectors for each layer, iii) the type of hidden node activation function and so.

At the instantiation of a new MAJNN object, the inputs needed are i) (Some) input data matrix. ii) (some) output data matrix. iii) a user-specified architecture for the hidden layers represented as a python list where the length of the list corresponds to the number of desired hidden layers and the elements correspond to the number of nodes in each layer. and iv) The activation function type of the hidden layers, these being either 'relu' or 'sigmoid'. With these the 'constructor' basically sets up all the above mention model state and determines the size of each hidden layer matrix and randomly initialises these. The below code snippet shows the 'init' of the MAJNN class.



```python
# Status: Done
class MAJNN():
    # status:
    def __init__(self, hidden_architecture, X, y_output_dim, usr_activation_function):
        '''
        Input:
            - Architecture input (list): - Length of list is the number of hidden layers
              and the value of each element corresponds to the  nr of nodes in that layers
            - X input data (array) Determines the input size of network
            - y 'labels'(array): determines the output layer size of network.
            - activation_function:  either 'relu' or 'sigmoid'. Raises error for any other string.
        '''

        # Construct newtwork weights and biases matrices from X,y
        # and hidden_architecture
        self.__initialize_parameters(X, hidden_architecture, y_output_dim)

        # Assign and determine activation function
        self.__set_activation_function(usr_activation_function)

```

The 'create_and_train' function. In order to streamline both the construction of a network and its training, the user doesn't actually contsctruct anything but specifics the desired network to the 'create_and_train' function that does all the standar "routine work". The below bellow snippet shows the internals of function which boils down to basically its name:



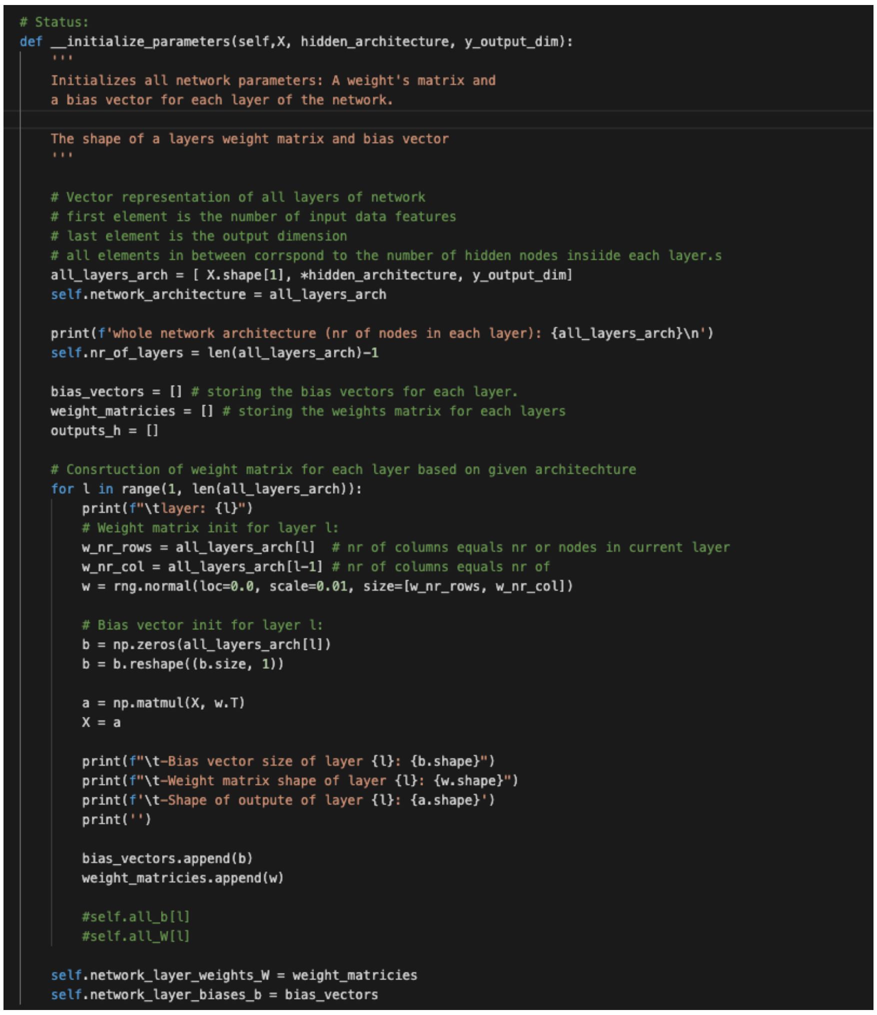
```python
# Status: Done
def create_and_train_NN(X_train, y_train, X_test, y_test,
                        hidden_architecture, activation_function,
                        learning_rate = 0.4,
                        minibatch_size = 30, nr_epochs=3,
                        verbose=True):
    '''
    Convenience function that contructs, trains and returns back a MAJNN neural network object.
    Input:
            - X_train, y_train, X_test, y_test,
            - hidden_architecture,
            - activation_function: string :
            - learning_rate : scaler
            - minibatch_size = 30,
            - nr_epochs=3,
            - verbose=True):
    Output:
            - mnn: trained MAJNN neural network object.
            - jtrn, jtst: cost vs update-iterations vectors for test and train set
            - acctrn, acctst: accuracy vs. update iterations for test and train sets.
    '''
    #Extract Key constants
    train_N = X_train.shape[0] #nr of date points
    train_D = X_train.shape[1] # nr of input features
    train_y_dim = y_train.shape[1] # nr of output "features"

    print('N: ', train_N)
    print('D: ', train_D)
    print('number of classes: ', train_y_dim)

    # Instansiate MAJ neural network object
    mnn = MAJNN(X=X_train,
                hidden_architecture = hidden_architecture,
                y_output_dim = train_y_dim,
                usr_activation_function = activation_function)

    # Nr of params in network
    print(f"\nNr of params(weights and offsets): {mnn.nr_params()}")
    print(f"\nNr of layers: {mnn.nr_of_layers}")
    print(f"\nNr of weight matrices: {len(mnn.network_layer_weights_W)}")

    # Train network
    jtrn, jtst, acctrn, acctst  = mnn.train_model(X_train, y_train,
                                                  X_test, y_test,
                                                  nr_epochs = nr_epochs,
                                                  learning_rate = 0.1,
                                                  verbose=False)

    return mnn, jtrn, jtst, acctrn, acctst

```

2.4. Network weight initialisation, architecture and model_state

The number and 'shape' of the parameters for layer is strictly determined by the number of nodes in the previous layer and the size of the current layer such that the matrix the K'th layer eight metric has the shape:

$$W^k : |N_k| \times |N_{k-1}|$$

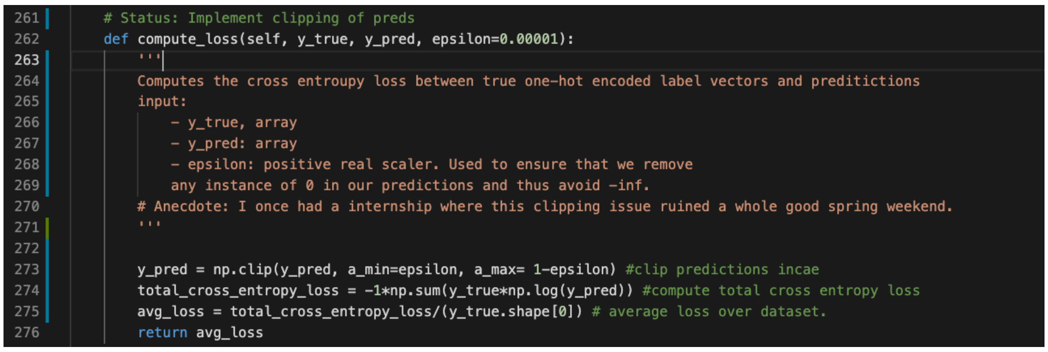
Where $|N_k|$ is the number of nodes in the set of nodes in the k'th layer. When k is 1, the number of input features is used instead for the rows. The below code snippets shows the responsible function for determining this. All the comments and print-outs make it more beefier than it actually should:



```
# Status:
def __initialize_parameters(self,X, hidden_architecture, y_output_dim):
    '''
    Initializes all network parameters: A weight's matrix and
    a bias vector for each layer of the network.

    The shape of a layers weight matrix and bias vector
    '''

    # Vector representation of all layers of network
    # first element is the number of input data features
    # last element is the output dimension
    # all elements in between corrspond to the number of hidden nodes insiide each layer.s
    all_layers_arch = [ X.shape[1], *hidden_architecture, y_output_dim]
    self.network_architecture = all_layers_arch

    print(f'whole network architecture (nr of nodes in each layer): {all_layers_arch}\n')
    self.nr_of_layers = len(all_layers_arch)-1

    bias_vectors = [] # storing the bias vectors for each layer.
    weight_matricies = [] # storing the weights matrix for each layers
    outputs_h = []

    # Consrtuction of weight matrix for each layer based on given architechture
    for l in range(1, len(all_layers_arch)):
        print(f"\tlayer: {l}")
        # Weight matrix init for layer l:
        w_nr_rows = all_layers_arch[l]  # nr of columns equals nr or nodes in current layer
        w_nr_col = all_layers_arch[l-1] # nr of columns equals nr of
        w = rng.normal(loc=0.0, scale=0.01, size=[w_nr_rows, w_nr_col])

        # Bias vector init for layer l:
        b = np.zeros(all_layers_arch[l])
        b = b.reshape((b.size, 1))

        a = np.matmul(X, w.T)
        X = a

        print(f"\t-Bias vector size of layer {l}: {b.shape}")
        print(f"\t-Weight matrix shape of layer {l}: {w.shape}")
        print(f'\t-Shape of outpute of layer {l}: {a.shape}')
        print('')

        bias_vectors.append(b)
        weight_matricies.append(w)

        #self.all_b[l]
        #self.all_W[l]

    self.network_layer_weights_W = weight_matricies
    self.network_layer_biases_b = bias_vectors
```

2.5. Cross entropy loss clipping

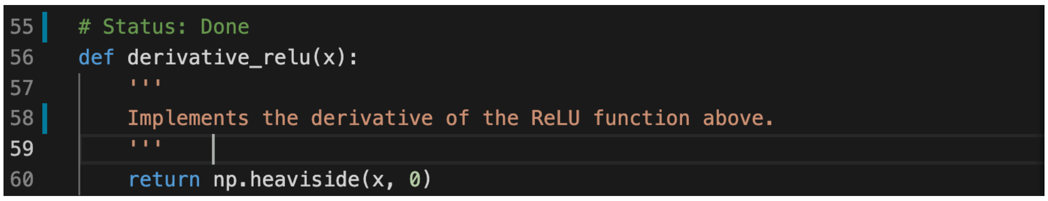
One thing that kind of gave me a small issue early on and that I later read is already dealt with in the major neural network frameworks/librraies is 'clipping the class predictions'. As we know, the cross-entropy loss involves taking the (natural) logarithm of the softmax-output of the layer in our network. An issue here that can arise is if the softmax value is very small, resulting in a very large negative value, potentially leading to numerical instability. There, as a counter measure, we 'clip' the softmax such that we replace any element below a threshold value ϵ with epsilon. We also replace any value of 1 with $1 - \epsilon$. The below code snippet shows this in the 'compute_loss' function.



```python
# Status: Implement clipping of preds
def compute_loss(self, y_true, y_pred, epsilon=0.00001):
    '''
    Computes the cross entroupy loss between true one-hot encoded label vectors and preditictions
    input:
        - y_true, array
        - y_pred: array
        - epsilon: positive real scaler. Used to ensure that we remove
        any instance of 0 in our predictions and thus avoid -inf.
    # Anecdote: I once had a internship where this clipping issue ruined a whole good spring weekend.
    '''

    y_pred = np.clip(y_pred, a_min=epsilon, a_max= 1-epsilon) #clip predictions incae
    total_cross_entropy_loss = -1*np.sum(y_true*np.log(y_pred)) #compute total cross entropy loss
    avg_loss = total_cross_entropy_loss/(y_true.shape[0]) # average loss over dataset.
    return avg_loss
```

2.6. Heaviside function approach to the derivative of the ReLU function.

This is a minor thing but in found this a pleasing connection: When computing the error with respect to each layers activation outputs, the expression involves the derivative of the activation function. In the case for a ReLU activation function, the derivative is 1 if the input is strictly positive and 0 otherwise. And this is done element-wise for the input vector. The practical issue I was facing was how to vectorize the above 'conditional output' and not use any for loops. Luckily, the above expression for the derivative of the ReLU function happens to be more or less a modification of the Heaviside function which already has a vectorised numpy implementation:



```python
# Status: Done
def derivative_relu(x):
    '''
    Implements the derivative of the ReLU function above.
    '''
    return np.heaviside(x, 0)
```

2.7. One-step gradient decent parameter update

I opted here to stray a bit from the assignment guidelines when it came to how to 'chop up' the logic in the back-propagoation parameter weights update. The alignment description suggested to write a bunch of dedicated helper functions that separate the backward_linear and backward_activation steps, but i found this frustrating to deal with when writing the algorithm and when debugging so instead I instead opted code up all the parts inline in a large the 'update_parameters' function. But because it should result in the same underlying operation, there is no harm. The "algorithmics" in pseudocode for the layer by layer update through back propagation was:

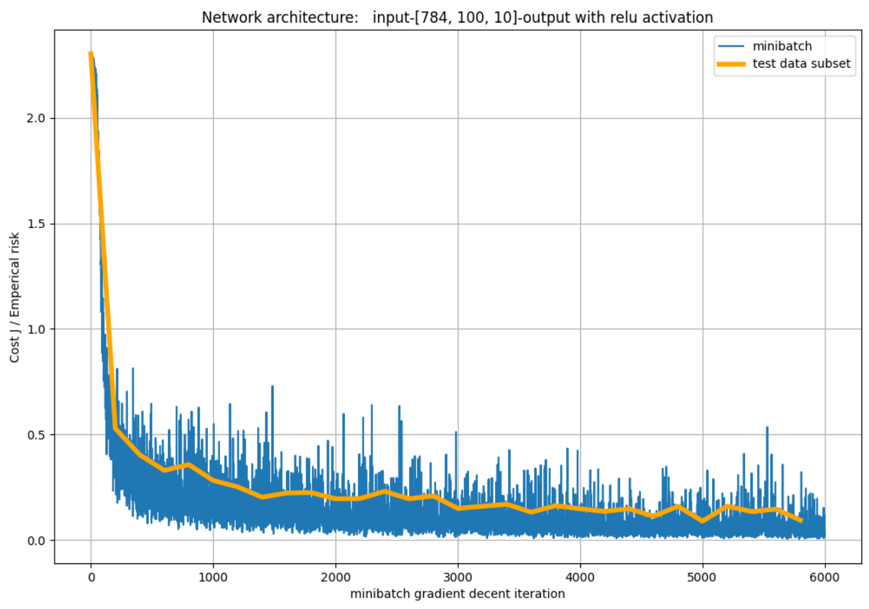
```
0. Compute error vector defined by dL/doutputlayer z
and assign to  'delta_current'


For each layer starting with the last hidden layer and working backwards
layer by layer:
    1. Compute the partial derivates w.r.t to the current layers weights by
matrix multiplication of the product-sum output of the current layer and
the output of the previous layer.
    2. compute the new weight of the current layer by subtract a quantity
learning_rate*gradient from old weight
    3. compute next error vector for the next layer and assign to
'delta_current'
    4. update old weights to new weights for current layer.
```

---

3. Exercise 1.3 - One hidden layer Network for MNIST classification

In this section, I'll briefly show my results for trying two single hidden layer networks (one using a sigmoid activation function and the other, a ReLu activation function) to demonstrate that my implementation works.

As described in the assignment, we were to perform a classification task on the MNIST dataset of hand-written digits. The training occurred via mini-batch gradient decent with a batch size of 30 or 32 in accordance with [***Masters, D., & Luschi, C. (2018). Revisiting small batch training for deep neural networks. arXiv preprint arXiv:1804.07612.***]. In figure 1 below we show the training phase cost and accuracy over the update-iterations for both the training and sampled validation (testset in our case) datasets for the case of the Relu activation functions. In figure 2, we have the same content but for a network using sigmoid activations instead. In both cases, the hidden layer contained 100 nodes.



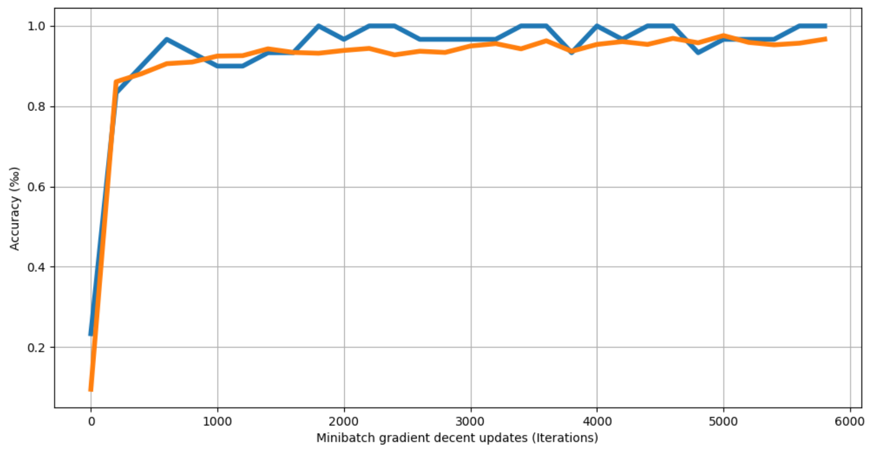
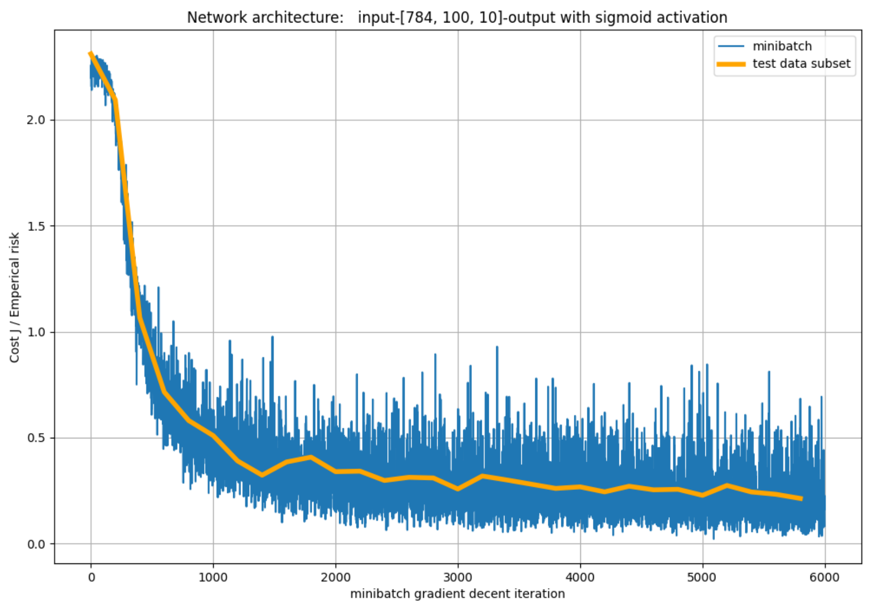


Figure 1: *Training phase for a neural network with a single hidden layer and the ReLu function for the hidden nodes activation function.* ***(Top)*** *Cost after each mini-batch stochastic gradient decent update. Blue line is cost over each mini batch and the orange/yellow line is the cost for the validation set [1000 random images sampled from the training set].* ***(Bottom)*** *Classification accuracy of the network during the training phase: Blue line accuracy over a mini batch every 200th mini-batch. Organge/Yellow line is the accuracy of the network after the k'th weights update over 1000 randomly sampled images from the MNIST test set*



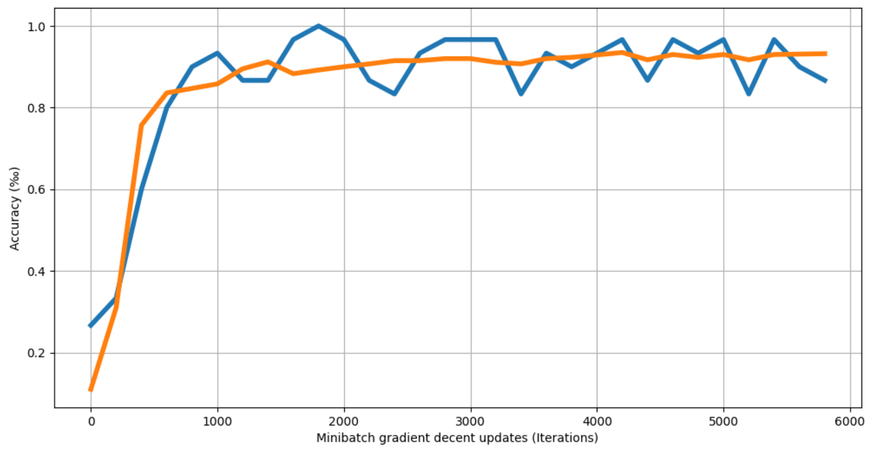
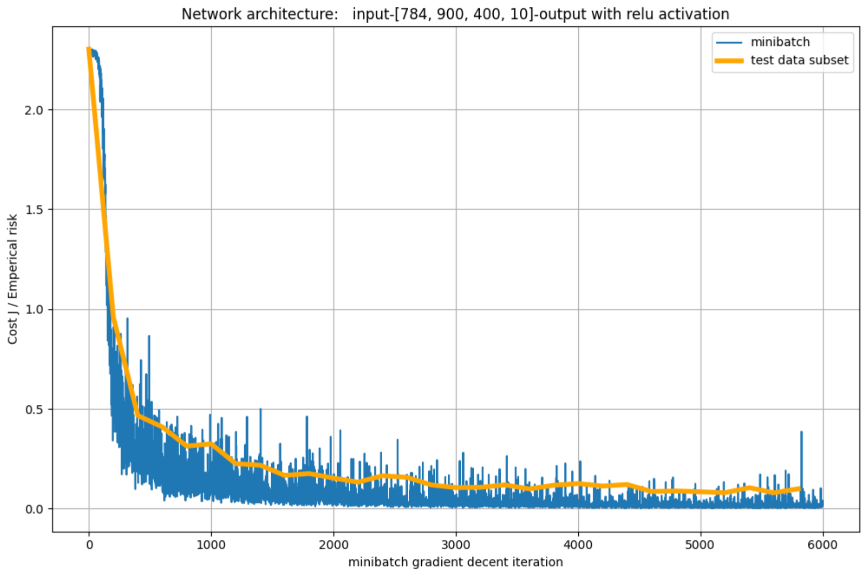


Figure 2: *Training phase for a neural network with a single hidden layer and the sigmoid function for the hidden nodes activation function.* ***(Top)*** *Cost after each mini-batch stochastic gradient decent update. Blue line is cost over each mini batch and the orange/yellow line is the cost for the validation set [1000 random images sampled from the training set].* ***(Bottom)*** *Classification accuracy of the network during the training phase: Blue line accuracy over a mini batch every 200th mini-batch. Organge/Yellow line is the accuracy of the network after the k'th weights update over 1000 randomly sampled images from the MNIST test set*

4. Exercise 1.4 - Multiple hidden layers Network for MNIST classification

In this section, I'll briefly show my results for trying two multi hidden layer networks (one using a sigmoid activation function and the other, a ReLu activation function) to demonstrate that my implementation works.

Like for exercise 1.3, in figures 3 and 4 below we show the training phase cost and accuracy over the update-iterations for both the training and sampled validation (testset in our case) datasets for the two cases of activation functions. The network used in figure 3 had a [900, 400] "hidden" architecture, and for figure 4, the network used was a 3 hidden layer [400, 100, 25] hidden architecture.



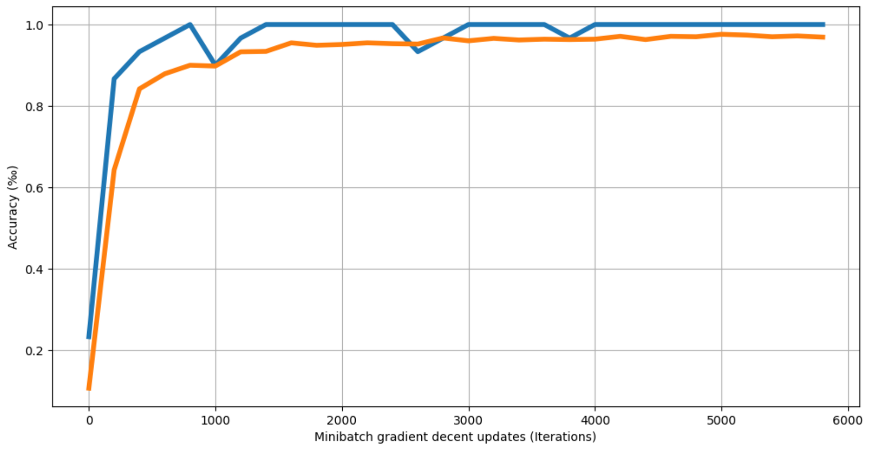
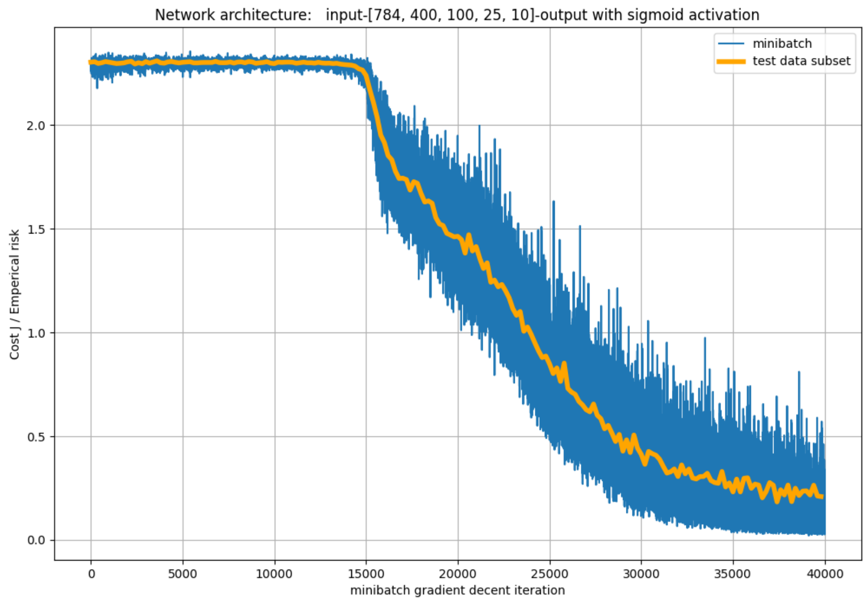


Figure 3: *Training phase for a neural network with 2 hidden layers and the ReLu function as the hidden nodes activation function.* ***(Top)*** *Cost after each mini-batch stochastic gradient decent update. Blue line is cost over each mini batch and the orange/yellow line is the cost for the validation set [1000 random images sampled from the training set].* ***(Bottom)*** *Classification accuracy of the network during the training phase: Blue line accuracy over a mini batch every 200th mini-batch. Organge/Yellow line is the accuracy of the network after the k'th weights update over 1000 randomly sampled images from the MNIST test set*



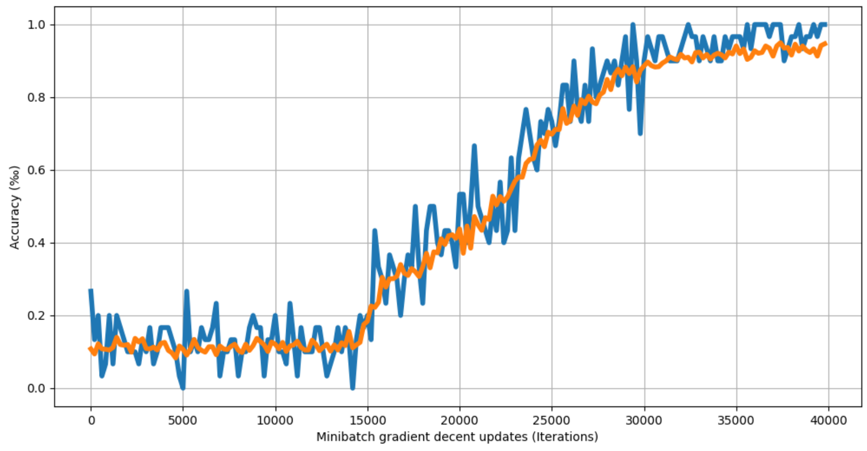


Figure 4: *Training phase for a neural network with 2 hidden layers and the Sigmoid function as the hidden nodes activation function.* ***(Top)*** *Cost after each mini-batch stochastic gradient decent update. Blue line is cost over each mini batch and the orange/yellow line is the cost for the validation set [1000 random images sampled from the training set].* ***(Bottom)*** *Classification accuracy of the network during the training phase: Blue line accuracy over a mini batch every 200th mini-batch. Organge/Yellow line is the accuracy of the network after the k'th weights update over 1000 randomly sampled images from the MNIST test set*

4.1. Last layer weights as Images

In this section we show the asked for 'weights images' for the last layer weight matrices for two networks. In figure 5 below we have the weights images for the final layer of a single hidden layer relu network. In figure 6 below we have the weights images for the final layer of a multiple hidden layer sigmoid network with hidden architecture [900, 784].

I think the major take home is that there doesn't seem to be a lot of spatial structure or human perceptible spatial meaning to these weight 'images' when rearranging the columns into squares. I think this is kind of reasonable because we are both flattening our digits images inputs AND we use fully connected layers which in effect breaks the spatial relations between pixels as far as the network 'understands' and learns. When the network then computes the inner product between the current weights and the previous layers output for a given input images, this resulting project value is detached from a 2d notion av pixel neighbourhood and spatial structure.

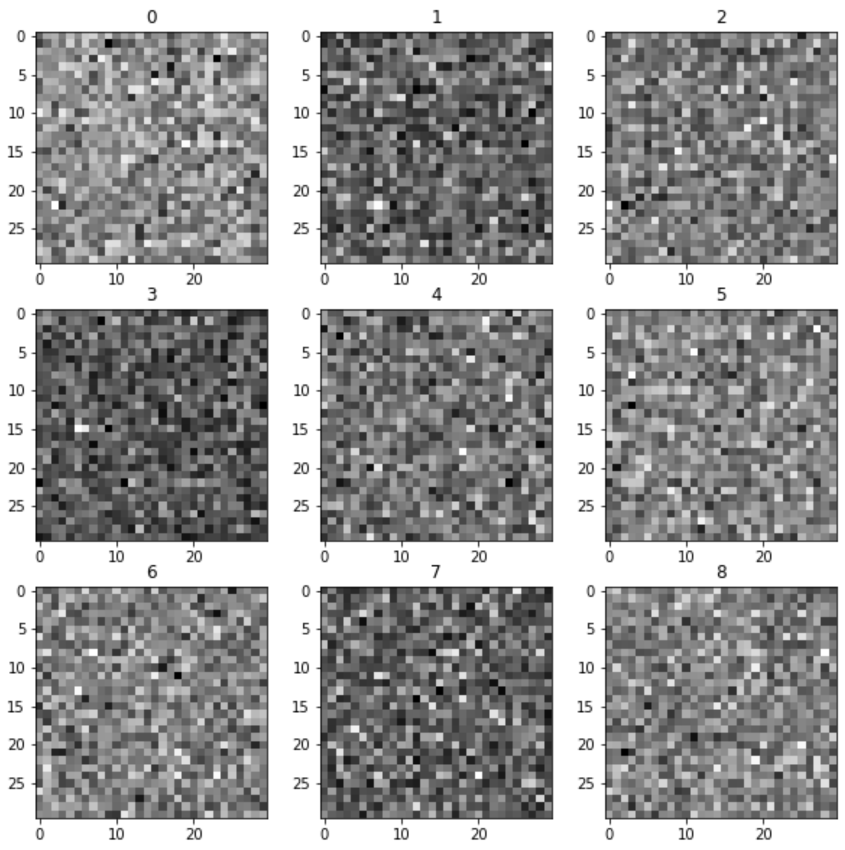


Figure 5: Final hidden layer weights rearranged as 30x30 images for a network with 1 single hidden relu layer with 900 hidden nodes. Each sub-image corresponds to the class in its title

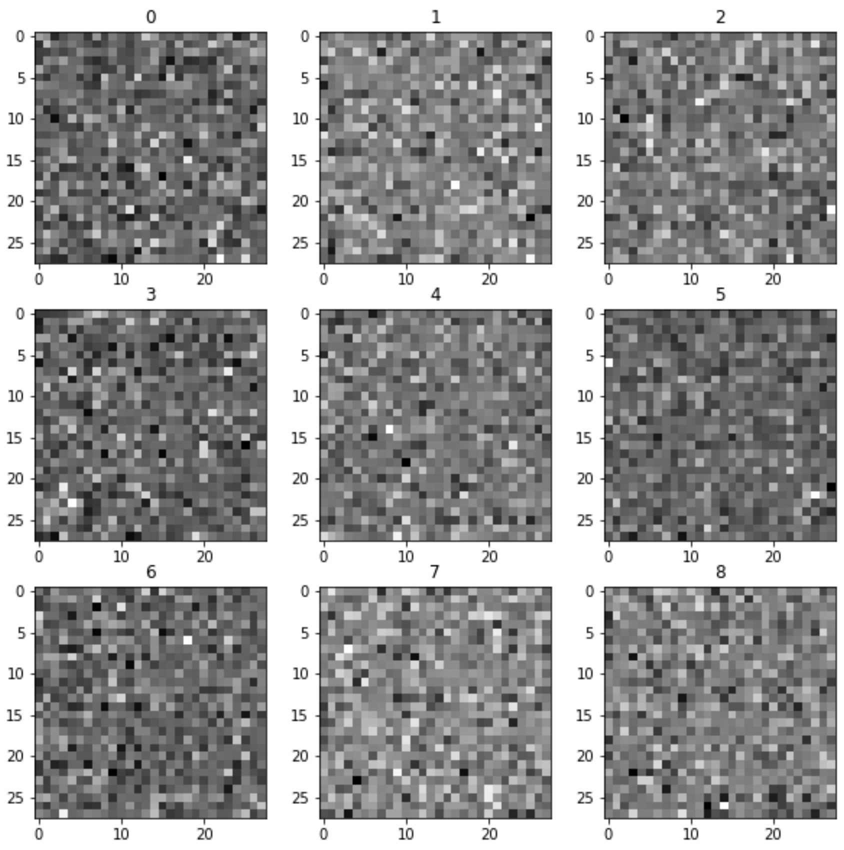


Figure 6: Final hidden layer weights rearranged as 28x28 images for a network with 2 hidden sigmoid layer with a [900, 784] architechture. Each sub-image corresponds to the class in its title