Building a neural network from scratch for digit recognition

1 Abstract

The objective of this project is to build a simple neural network with three layers (input, hidden, output) that takes handwritten character digits in a 28x28 pixel image format as input and outputs what it thinks the character is.

2 Data

The EMNIST dataset is a set of handwritten character digits derived from the NIST Special Database 19 and converted to a 28x28(=784) pixel image format. Specifically, we will work with the emnist-balanced-test and emnist-balanced-train datasets:

1. emnist-balanced-train has 112800 rows and 785 columns. Each row represents a different image. Column 1 represents the correct labelling of the image and columns 2 to 785 represent the pixel values(greyscale, all between 0 and 255). This will be our training data.

2. emnist-balanced-test has 18800 rows and 785 columns. Each row represents a different image. Column 1 represents the correct labelling of the image and columns 2 to 785 represent the pixel values(greyscale, all between 0 and 255). This will be our testing data.

3. emnist-balanced-mapping: There are 47 different characters in our training and testing datasets. Each character is thus encoded by an index between 0 and 46. emnist-balanced-mapping gives the mapping from the encoded label of the characters in the data to their Unicode representation. We can then use the built-in Python function char() to display the character. The relevant code and decoding table are displayed below in Figure 1 and Table 1 respectively.

In [3]:
```
#each label is encoded, so we also read in the file emnist-balanced-mapping.txt for decoding purposes
mapping = pd.read_table('emnist-balanced-mapping.txt', names = ['label', 'unicode'], header = None, delim_whitespace  = True)

#column 0 represents the labels in the data, and column 1 is the corresponding unicode integer.
#we can see what character each label represents using the chr() function
char_vec = []
for i in range(len(mapping)):
    a = chr(mapping.loc[i][1])
    char_vec.append(a)
mapping['character'] = char_vec
mapping
```

Figure 1: Decoding code

Encoded label	Unicode	Character
0	48	0
1	49	1
2	50	2
3	51	3
4	52	4
5	53	5
6	54	6
7	55	7
8	56	8
9	57	9
10	65	A
11	66	B
12	67	C
13	68	D
14	69	E
15	70	F
16	71	G
17	72	H
18	73	I
19	74	J
20	75	K
21	76	L
22	77	M
23	78	N
24	79	O
25	80	P
26	81	Q
27	82	R
28	83	S
29	84	T
30	85	U
31	86	V
32	87	W
33	88	X
34	89	Y
35	90	Z
36	97	a
37	98	b
38	100	d
39	101	e
40	102	f
41	103	g
42	104	h
43	110	n
44	113	q
45	114	r
46	116	t

Table 1: Decoding table

2.1 An example

Below, in Figure 1 we return the output of a snippet of our main code. We extract the pixels vector indexed 71 from our testing data and reshape it to show the image it represents. We also note that the returned label is 26. Using the above table we can see this corresponds to 'Q'.

```
In [77]: image_Example = testing_data[71:72, 1:785]
         label_image_Example = testing_data[71,0]

         label_image_Example

Out[77]: 26

In [78]: #we reshape the vector into a 28 by 28 pixel format for plotting purposes
         image_Example = image_Example.reshape((28,28))
         #we transpose this new matrix to ensure the image is the right way up
         image_Example = np.transpose(image_Example)

In [79]: #we plot the image
         plt.gray()
         plt.imshow(image_Example)
         plt.show()
```

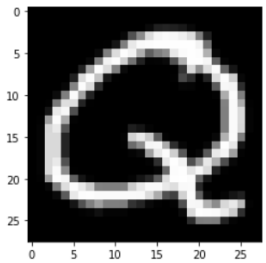

Figure 2: image example

3 Architecture

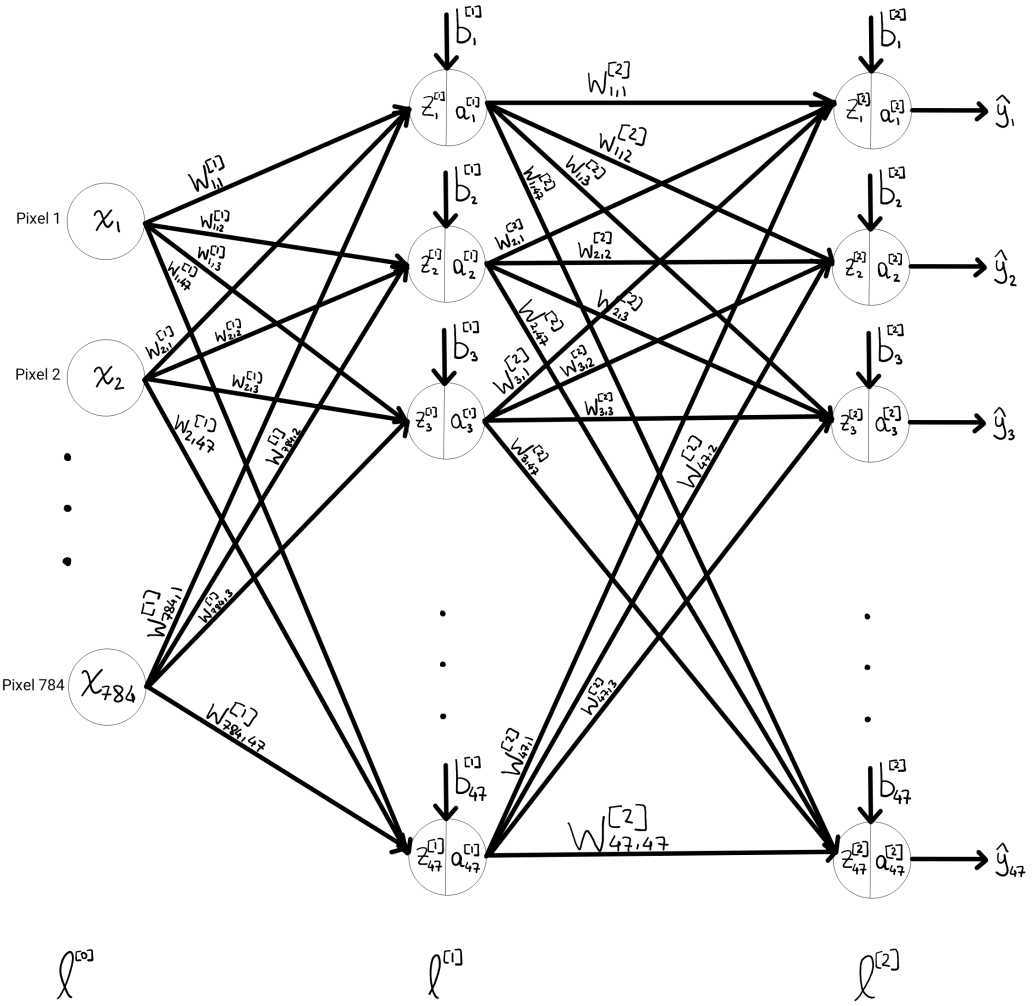

Figure 3: Neural Network

3.1 Forward Propagation

We begin at $\ell^{[0]}$ with our vectorised image, where each entry is a pixel value:

$$X = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{784} \end{pmatrix}$$

Next we apply our $\ell^{[1]}$ layer weights $W^{[1]}$ and biases $B^{[1]}$ to X and store the resulting vector in $Z^{[1]}$:

$$W^{[1]} = \begin{pmatrix} w_{1,1}^{[1]} & w_{2,1}^{[1]} & \dots & w_{784,1}^{[1]} \\ \vdots & \vdots & \dots & \vdots \\ w_{1,47}^{[1]} & w_{2,47}^{[1]} & \dots & w_{784,47}^{[1]} \end{pmatrix}$$

$$B^{[1]} = \begin{pmatrix} b_1^{[1]} \\ \vdots \\ b_{47}^{[1]} \end{pmatrix}$$

$$Z^{[1]} = W^{[1]}X + B^{[1]} = \begin{pmatrix} w_{1,1}^{[1]}x_1 + w_{2,1}^{[1]}x_2 + \ldots + w_{784,1}^{[1]}x_{784} + b_1^{[1]} \\ \vdots \\ w_{1,47}^{[1]}x_1 + w_{2,47}^{[1]}x_2 + \ldots + w_{784,47}^{[1]}x_{784} + b_{47}^{[1]} \end{pmatrix} = \begin{pmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ \vdots \\ z_{47}^{[1]} \end{pmatrix}$$

Now we apply our $\ell^{[1]}$ layer activation function to $Z^{[1]}$ and store the result in $A^{[1]}$:

$$A^{[1]} = \begin{pmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ \vdots \\ a_{47}^{[1]} \end{pmatrix} = \begin{pmatrix} ReLU(z_1^{[1]}) \\ ReLU(z_2^{[1]}) \\ ReLU(z_3^{[1]}) \\ \vdots \\ ReLU(z_{47}^{[1]}) \end{pmatrix}$$

Where

$$ReLU(z) = \begin{cases} z & \text{if } z > 0 \\ 0 & otherwise \end{cases}$$

$A^{[1]}$ is the output of $\ell^{[1]}$. Now we move on to $\ell^{[2]}$.
We apply our $\ell^{[2]}$ layer weights $W^{[2]}$ and biases $B^{[2]}$ to $A^{[1]}$ and store the resulting vector in $Z^{[2]}$:

$$W^{[2]} = \begin{pmatrix} w_{1,1}^{[2]} & w_{2,1}^{[2]} & \dots & w_{47,1}^{[2]} \\ \vdots & \vdots & \dots & \vdots \\ w_{1,47}^{[2]} & w_{2,47}^{[2]} & \dots & w_{47,47}^{[2]} \end{pmatrix}$$

$$B^{[2]} = \begin{pmatrix} b_1^{[2]} \\ \vdots \\ b_{47}^{[2]} \end{pmatrix}$$

$$Z^{[2]} = W^{[2]}A^{[1]} + B^{[2]} = \begin{pmatrix} w_{1,1}^{[2]}a_1^{[1]} + w_{2,1}^{[2]}a_2^{[1]} + \ldots + w_{47,1}^{[2]}a_{47}^{[1]} + b_1^{[2]} \\ \vdots \\ w_{1,47}^{[2]}a_1^{[1]} + w_{2,47}^{[2]}a_2^{[1]} + \ldots + w_{47,47}^{[2]}a_{47}^{[1]} + b_{47}^{[2]} \end{pmatrix} = \begin{pmatrix} z_1^{[2]} \\ z_2^{[2]} \\ z_3^{[2]} \\ \vdots \\ z_{47}^{[2]} \end{pmatrix}$$

Lastly we apply our $\ell^{[2]}$ layer activation function to $Z^{[2]}$ and store the result in $A^{[2]}$ and output it:

$$A^{[2]} = \begin{pmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \hat{y}_3 \\ \vdots \\ \hat{y}_{47} \end{pmatrix} = softmax(Z^{[2]}) = \begin{pmatrix} \frac{exp(z_1^{[2]})}{\sum_{m=1}^{47} exp(z_m^{[2]})} \\ \frac{exp(z_2^{[2]})}{\sum_{m=1}^{47} exp(z_m^{[2]})} \\ \frac{exp(z_3^{[2]})}{\sum_{m=1}^{47} exp(z_m^{[2]})} \\ \vdots \\ \frac{exp(z_{47}^{[2]})}{\sum_{m=1}^{47} exp(z_m^{[2]})} \end{pmatrix}$$

output: $\hat{y} = A^{[2]}$.
Where the objective of the softmax function is to take a vector of numbers and convert it into a vector of probabilities. This concludes the forward propagation phase.

4 Training

For the neural network to learn, we must first be able to quantify how much the network's prediction $\hat{y}$ deviates from the target prediction (which we will denote by y^{target}). We are given the true label for each image, thus given an image we define:

$$y^{target} = \begin{pmatrix} y_1^{target} \\ y_2^{target} \\ y_3^{target} \\ \vdots \\ y_{47}^{target} \end{pmatrix}$$

Where

$$y_i^{target} = \begin{cases} 1 & \text{if } i = label \\ 0 & otherwise \end{cases}$$

This is known as one hot encoding and has a nice interpretation as we would like for our network's prediction $\hat{y}$ to converge to the above as that would mean our network's prediction is right and the network has a confidence of 1 (100%) in its prediction.
We can now define a loss function to quantify how far off our predictions are:
$Loss = \frac{1}{2}\|\hat{y} - y^{target}\|^2 = \frac{1}{2}\sum_{i=1}^{47}(\hat{y}_i - y_i^{target})^2$

4.1 Backward Propagation

We now compute the derivative of the Loss function with respect to each of our hyper parameters. We will make extensive use of the chain rule.

Consider $\frac{\partial Loss}{\partial b_i^{[2]}} = \frac{\partial Loss}{\partial a_i^{[2]}} \times \frac{\partial a_i^{[2]}}{\partial z_i^{[2]}} \times \frac{\partial z_i^{[2]}}{\partial b_i^{[2]}} = (\hat{y}_i - y_i^{target})(\hat{y}_i(1-\hat{y}_i))(1)$

where we used $\frac{\partial a_i^{[2]}}{\partial z_i^{[2]}} = \frac{\partial}{\partial z_i^{[2]}}(\frac{exp(z_i^{[2]})}{\sum_{m=1}^{47} exp(z_m^{[2]})}) = \frac{exp(z_i^{[2]})}{\sum_{m=1}^{47} exp(z_m^{[2]})} - (\frac{exp(z_i^{[2]})}{\sum_{m=1}^{47} exp(z_m^{[2]})})^2 = a_i^{[2]}(1-a_i^{[2]}) = \hat{y}_i(1-\hat{y}_i)$.

Using the above we can conclude :
$\frac{\partial Loss}{\partial B^{[2]}} = (\hat{y} - y^{target}) * \hat{y} * (1-\hat{y})$ where $*$ represents element-wise multiplication (Hadamard product).

Similarly $\frac{\partial Loss}{\partial w_{i,j}^{[2]}} = \frac{\partial Loss}{\partial a_j^{[2]}} \times \frac{\partial a_j^{[2]}}{\partial z_j^{[2]}} \times \frac{\partial z_j^{[2]}}{\partial w_{i,j}^{[2]}} = (\hat{y}_j - y_j^{target})(\hat{y}_j(1-\hat{y}_j))(a_i^{[1]}))$ implies $\frac{\partial Loss}{\partial W^{[2]}} = (\hat{y} - y^{target}) * \hat{y} * (1-\hat{y}) A^{[1]^T}$.

By the same process:

$$\frac{\partial Loss}{\partial B^{[1]}} = (W^{[2]}((\hat{y} - y^{target}) * \hat{y} * (1 - \hat{y}))) * \begin{pmatrix} ReLU'(z_1^{[1]}) \\ ReLU'(z_2^{[1]}) \\ ReLU'(z_3^{[1]}) \\ \vdots \\ ReLU'(z_{47}^{[1]}) \end{pmatrix}$$

and

$$\frac{\partial Loss}{\partial W^{[1]}} = ((W^{[2]}((\hat{y} - y^{target}) * \hat{y} * (1 - \hat{y}))) * \begin{pmatrix} ReLU'(z_1^{[1]}) \\ ReLU'(z_2^{[1]}) \\ ReLU'(z_3^{[1]}) \\ \vdots \\ ReLU'(z_{47}^{[1]}) \end{pmatrix})X^T$$

where

$$ReLU'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & otherwise \end{cases}$$

Now we can use gradient descent (with learning rate α) to update our hyper-parameters at each training step by the following:

$$W^{[1]} \longleftarrow W^{[1]} - \alpha \frac{\partial Loss}{\partial W^{[1]}}$$

$$B^{[1]} \longleftarrow B^{[1]} - \alpha \frac{\partial Loss}{\partial B^{[1]}}$$

$$W^{[2]} \longleftarrow W^{[2]} - \alpha \frac{\partial Loss}{\partial W^{[2]}}$$

$$B^{[2]} \longleftarrow B^{[2]} - \alpha \frac{\partial Loss}{\partial B^{[2]}}$$

This is done in neuralnet_training.ipynb , where we repeated the training process 100 times via a for loop. Each of the 100 iterations consisted of training the network using the whole training dataset. The training section takes around 45 minutes to complete.

5 Results

Given an image, we will forward pass it through our neural network and predict its digit based on the index at which the highest probability occurs. This is also the reason why we had 47 nodes at $\ell^{[1]}$ and $\ell^{[2]}$ as we are working with 47 classes (or types) of digits and we need a probability corresponding to each possible class.

5.1 Testing accuracy

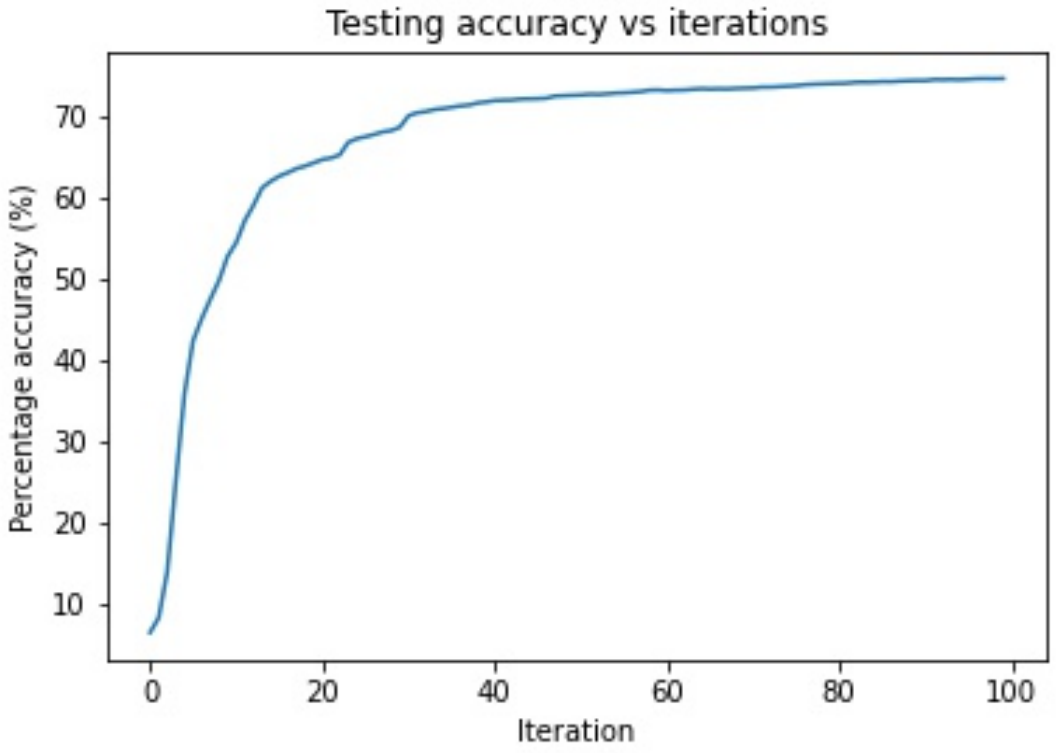

Figure 4: Testing accuracy vs training iterations

The final testing accuracy(percentage of correct predictions over testing dataset) after 100 iterations is 74.7%.

5.2 Training accuracy

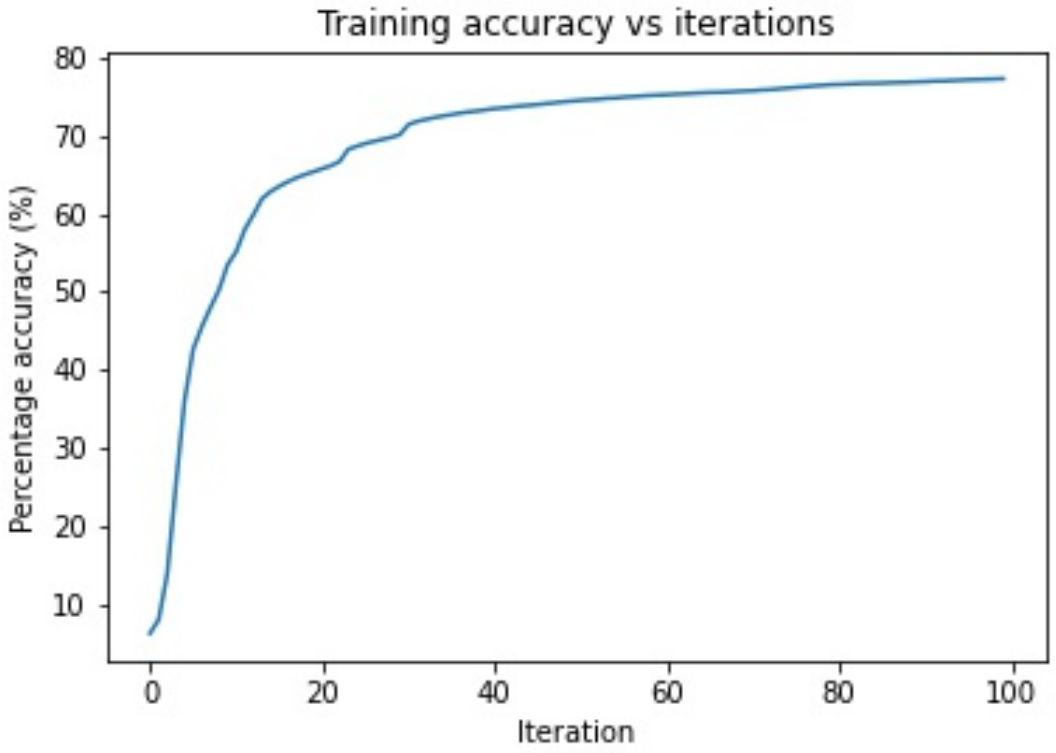

Figure 5: Training accuracy vs training iterations

The final training accuracy(percentage of correct predictions over training dataset) after 100 iterations is 77.3%.

6 GUI

Lastly, I decided to code up a very simple GUI that allows the user to draw a digit using the mouse and uses the network to recognize the digit. To save time, I ran the neuralnet_training.ipynb for the 45 minutes and saved the hyper-parameters as: "W1_pretrained.dat", "W2_pretrained.dat", "B1_pretrained.dat" and "B2_pretrained.dat". Thus, it is possible to run the GUI without having to train the network every time and having to wait the 45 minutes. The GUI is quite intuitive as once ran, a white box appears where the user should draw a single digit number or single letter (drawing as slowly and nicely as possible and not too small). Then, if we click on the predict button the network returns a prediction and a confidence in the prediction (this is where the softmax function returning probabilities becomes so useful). The clear button can be used to clear the canvas, so that a new number/letter can be drawn. This can be found in GUI.ipynb