Computer Vision Assignment 5
Luka Eerens

Q1.1
Answer:

The softmax equation is equal to:

$$softmax_{x_i} = \frac{e^{x_i}}{\sum_j e^{x_i}}$$

For softmax to be invariant to translation, any addition or subtraction by a scalar constant (denoted by c) should result in the same $\frac{e^{x_i}}{\sum_j e^{x_i}}$.

$$\therefore softmax_{x_i} = \frac{e^{x_i+c_i}}{\sum_j e^{x_i+c_i}}$$

And since c_i is a constant, it is c no matter what i, therefore:

$$softmax_{x_i} = \frac{e^{x_i+c}}{\sum_j e^{x_i+c}}$$

$$softmax_{x_i} = \frac{e^{x_i}e^{c}}{\sum_j e^{x_i}e^{c}}$$

Now e^c is in the numerator and the denominator therefore they can be cancelled out, leaving us with:

$$softmax_{x_i} = \frac{e^{x_i}}{\sum_j e^{x_i}}$$

Since the softmax is unchanging despite adding a constant, we show that softmax is invariant to translation.

The motivation behind using $softmax(x - \max(x))$ is in fact a normalization trick. It comes in handy because intermediate terms have the possibility of being large in value because of the exponentials. Dividing these large numbers can lead to unstable outcomes so this normalization handles this and provides a more numerically stable output.

Q1.2
Answer:

Each element is any number in between and including 0 and 1. This is why softmaxes are used in outputs in order to find probabilities of classifications belong to a certain class of output.

It will still be a vector but the real values get transformed into a value between and including 0 and 1. Therefore it has unit length so you could call it a unit length vector.

Step 1: Upscale vector
Step 2: Normalize vector
Step 3: Thin vector into unit length vector

Q1.3
Answer:
Consider this linear equation which outputs a weight of the input

$$y = W_1 x$$

Where W is the weights, x is the input. Layering these connections to each other has the following output:

$$y = W_2 x + W_1 x$$

As a new layer is added, we still retain the linear properties while increase the expressive power of the linear model.

Growing this model further until it has n number of layers gives:

$$y = W_n x + W_{n-1} x + \cdots + W_1 x$$

Which is equal to:

$$y = \sum_n W_n x$$

Which is the weighted sum of neural network inputs.

Q1.4

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

$$\frac{d}{dx}\sigma(x) = \frac{d}{dx}\frac{1}{1+e^{-x}}$$

$$\frac{d}{dx}\sigma(x) = -(1+e^{-x})^{-2}(-e^{-x})$$

$$\frac{d}{dx}\sigma(x) = -(1+e^{-x})^{-2}(-e^{-x})$$

$$\frac{d}{dx}\sigma(x) = \frac{e^{-x}}{(1+e^{-x})^2}$$

$$\frac{d}{dx}\sigma(x) = \frac{1}{1+e^{-x}} * \frac{e^{-x}}{1+e^{-x}}$$

$$\frac{d}{dx}\sigma(x) = \frac{1}{1+e^{-x}} * \frac{(1+e^{-x})-1}{1+e^{-x}}$$

$$\frac{d}{dx}\sigma(x) = \frac{1}{1+e^{-x}} * \left(1 - \frac{1}{1+e^{-x}}\right)$$

$$\frac{d}{dx}\sigma(x) = \sigma(x) * \big(1 - \sigma(x)\big)$$

So at no point during this derivation did we need to know the value of x.

Q1.5
Answer:

Deriving the loss with respect to the input through the chain rule gives:

$$\frac{dJ}{dx} = \frac{dJ}{dy}\frac{dy}{dx}$$

Now since $\frac{dJ}{dy} = \delta$, representing y as $x^T W$ gives:

$$\frac{dJ}{dx} = \delta \frac{dx^T W}{dx}$$

And so:

$$\frac{dJ}{dx} = \delta W^T$$

Where $W \epsilon R^{dxk}$

Deriving the loss with respect to the weights through the chain rule gives:

$$\frac{dJ}{dW} = \frac{dJ}{dy}\frac{dy}{dW}$$

Now since $\frac{dJ}{dy} = \delta$, representing y as $x^T W$ gives:

$$\frac{dJ}{dW} = \delta \frac{dx^T W}{dW}$$

And so:

$$\frac{dJ}{dW} = \delta x$$

Q1.6
Answer:

1) As the signal propagates through a deeper neural network, the cascading weighted sum of activations (which are all very small numbers) becomes smaller and smaller very quickly. You are adding and multiplying around numbers that are smaller in magnitude than 1 and so you could have an implosion of signal magnitude that is outputted by the neural network.

2 and 3) Sigmoid is between 0 and 1. Tanh is between 1 and -1. The reason why Tanh may be considered ahead of sigmoid is because in multilayer neural networks, during training, the sigmoids saturate at 0 or 1, and if saturation is at 0, then the gradient of the sigmoid activation implodes to 0 (vanishing gradient), while this issue doesn't exist as much for Tanh where saturation is at +1 or -1. The other thing is that the shallower gradient profile of the sigmoid means that the largest gradient magnitude is still smaller than 1, which adds to the vanishing gradient problem.

4)

$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad \therefore \sigma(x) = \frac{e^x}{1+e^x} \qquad \therefore \sigma(x) + \sigma(x)e^x = e^x$$

$$\therefore \frac{\sigma(x)}{1-\sigma(x)} = e^x$$

Now

$$tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$$

Therefore by plugging in for e^x we get:

$$tanh(x) = \frac{\left[\frac{\sigma(x)}{1-\sigma(x)}\right]^2 - 1}{\left[\frac{\sigma(x)}{1-\sigma(x)}\right]^2 + 1}$$

$$\therefore \tanh(x) = \frac{2\sigma(x)-1}{2\sigma(x)^2 - 2\sigma(x) + 1}$$

Q2.1.1
A neural network that is initialized with zeros will only propagate a signal of zeros through the network. You do also sorts of weighted sums, all of which are zeroed by the 0 initialization and so your network won't learn anything as no matter what input you feed into the network the activations will only output a sum of zeros no matter what.

Q2.1.2
Done in the code

Q2.1.3
We initialize with random numbers so that we can offer from the beginning as much a diversity of activation output for all neurons so that the network through training can learn as much a diversity of features. A network may learn the same feature for example from many neurons, while random initialization allows each neuron to express a different feature, and this is more useful. The scaling is done to make it converge faster and learn more efficiently.

Q2.2.1
Done in the code

Q2.2.2
Done in the code

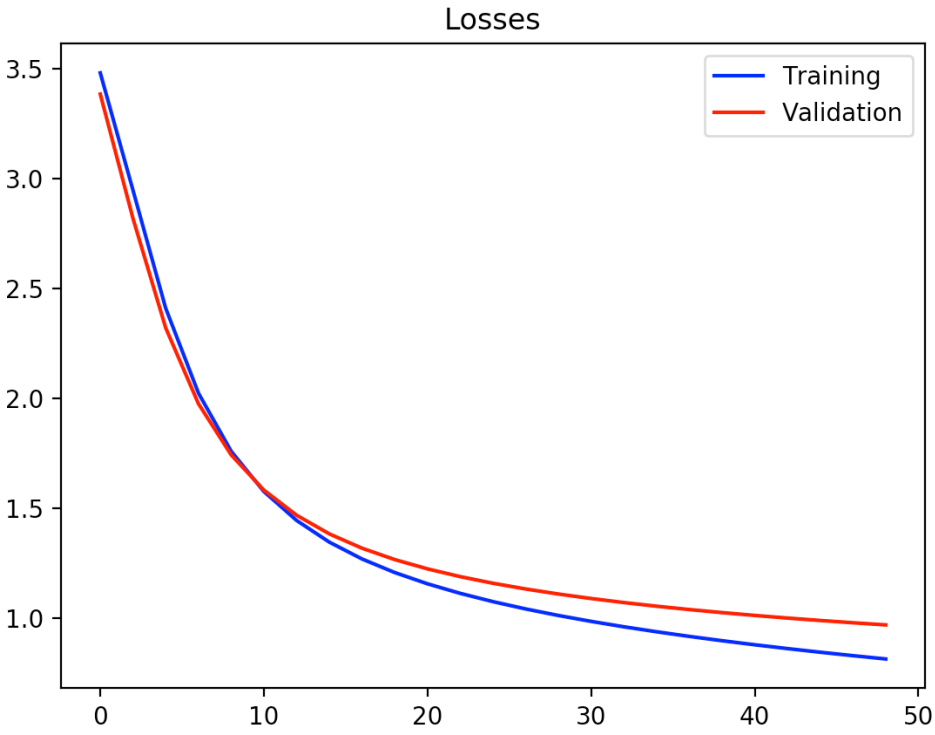
Q2.2.3
Done in the code

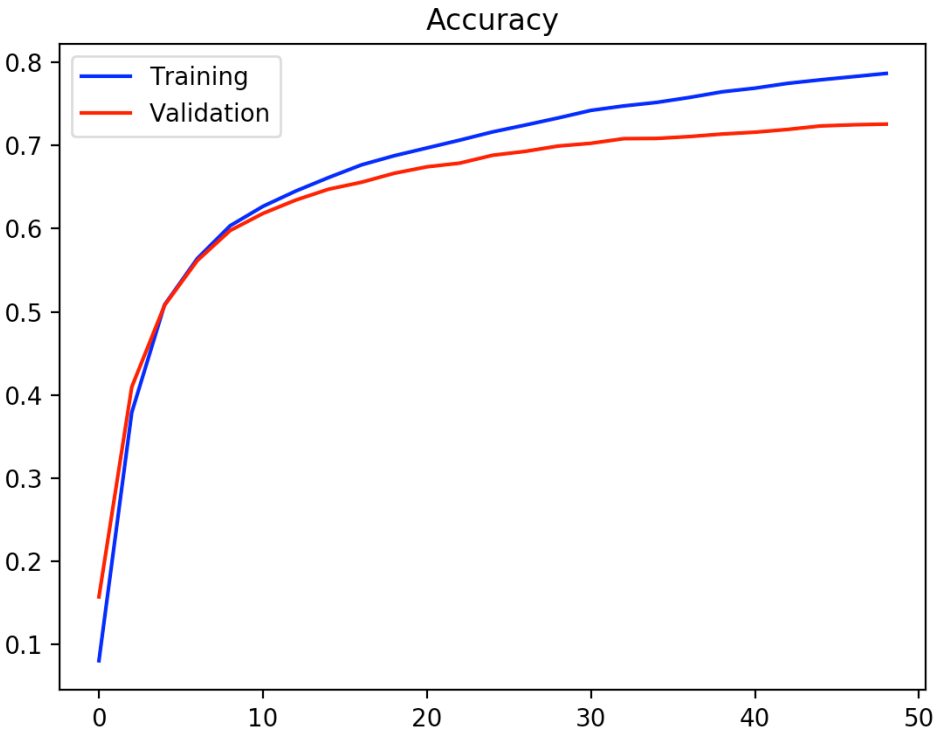
Q2.3.1
Done in the code

Q2.4.1
Done in the code

Q2.5.1
Done in the code

Q3.1.1
Answer:

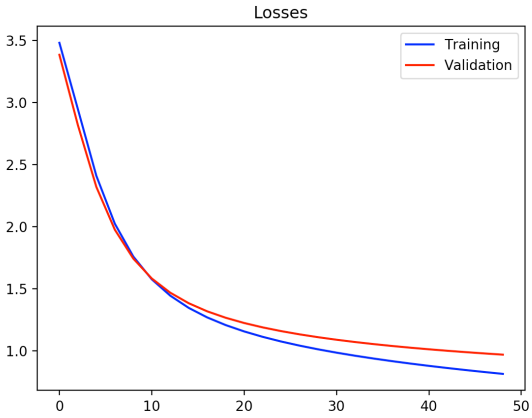

Losses
Training
Validation
3.5
3.0
2.5
2.0
1.5
1.0
0
10
20
30
40
50

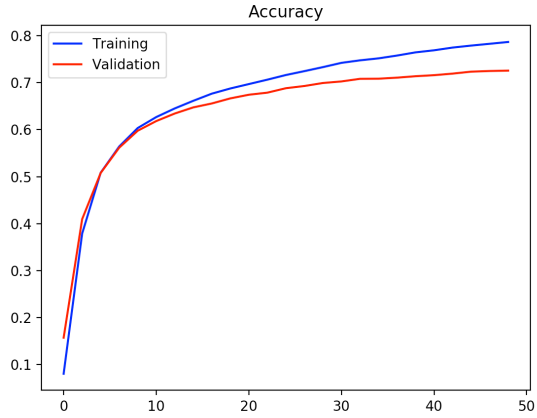

Accuracy
Training
Validation
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
10
20
30
40
50

Q3.1.2

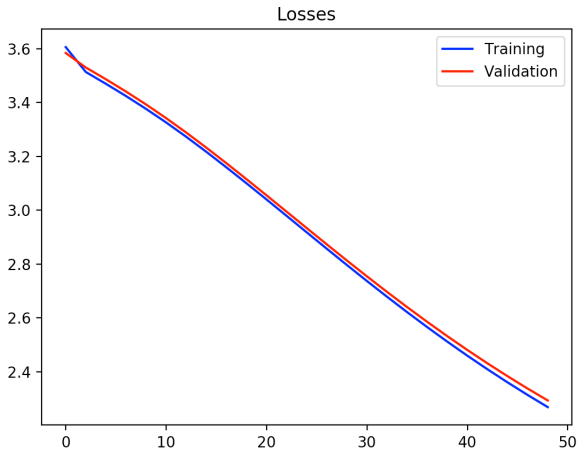
Answer:

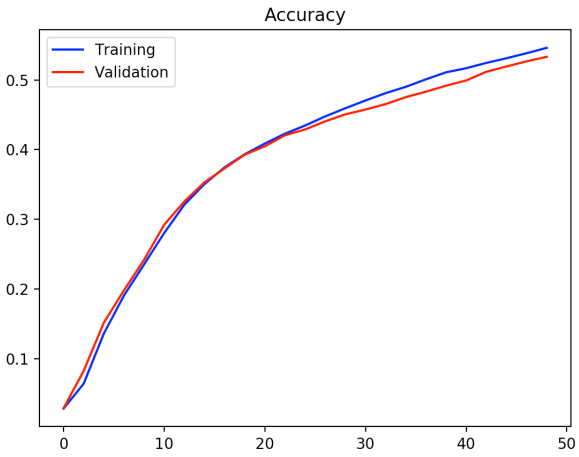
Learning Rate: 0.001



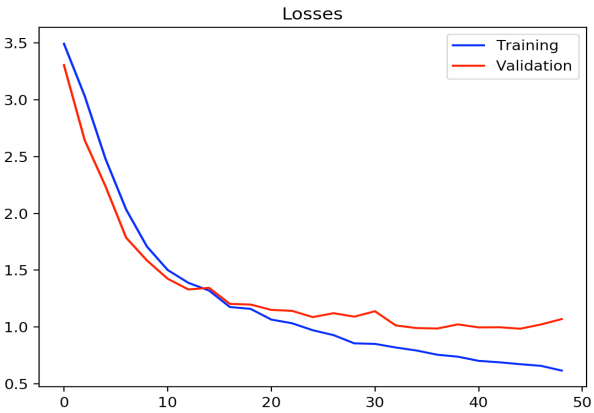


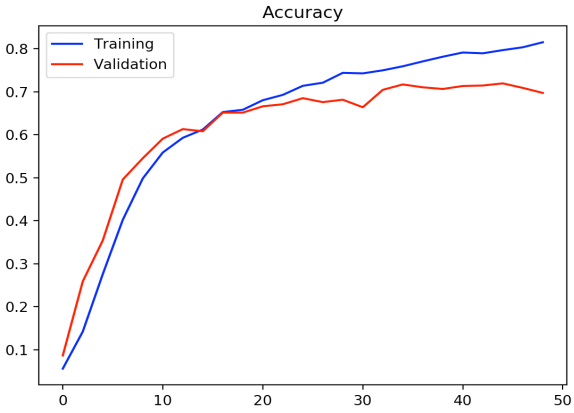
Learning Rate: 0.0001





Learning Rate: 0.01

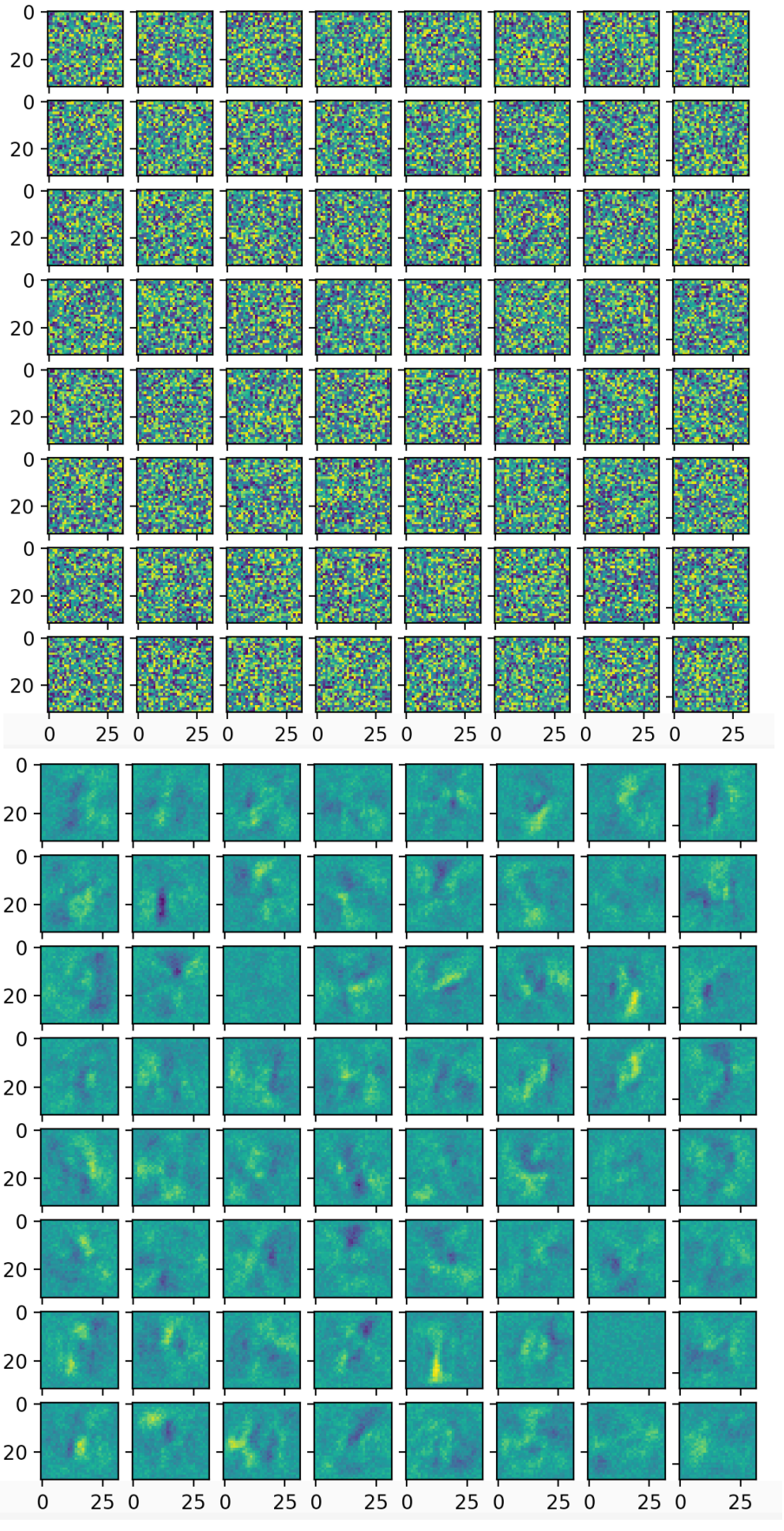




Small learning rates have smoother accuracy curves, which climb more slowly, while faster learning rates have more jagged profiles and climb faster and more irregularly. The best accuracy that I obtained on the test set was 0.74%.

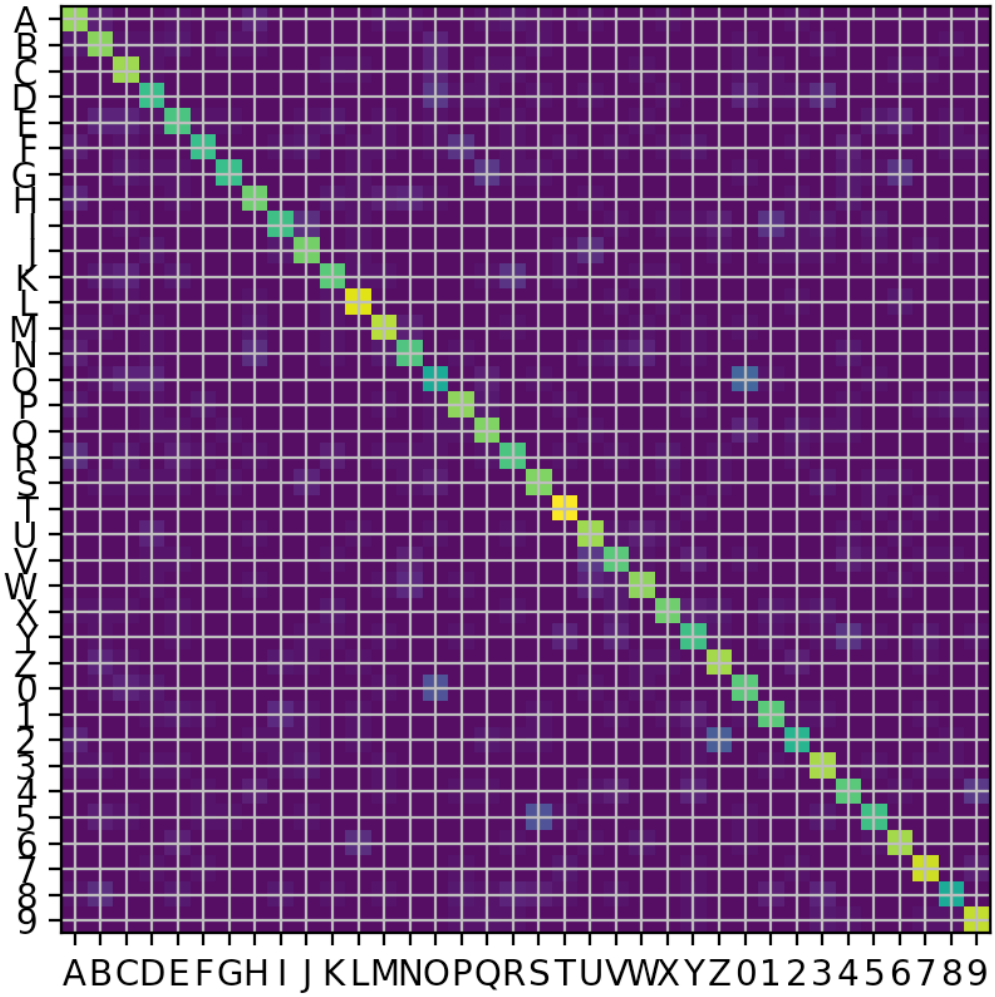
Q3.1.3

Answer:



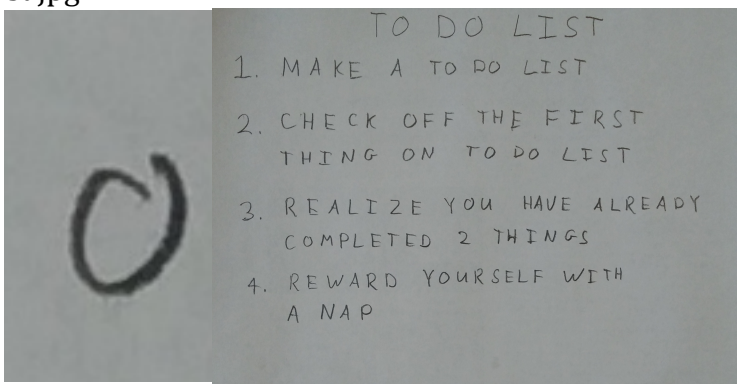
The first layer weight is the first grid, and the subsequent layer is the grid at the bottom. One thing that can be noticed is that the first layer appears to be more noisy and random in terms of filter profiles, while the there seems to be the nascent emergence of structure from this noise in the subsequent layer.

Q3.1.4
Answer:



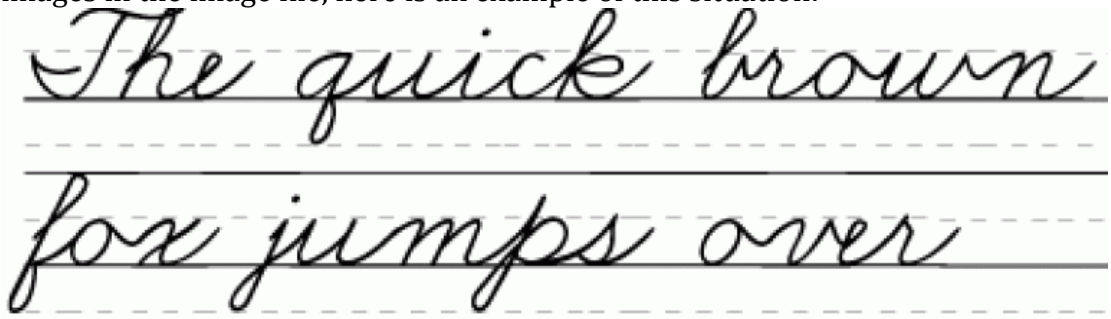
By looking at the confusion matrix, we can find that S and 5 are commonly confused, the letter O and the digit 0 are commonly confused, Z and 2 are also confused. All of this sounds right as these all share very similar features and even for humans can easily be mistaken for one another.

Q4.1
Answer:
Any character whose parts are disjointed from the rest of the joined marking of that character may affect the performance of this system. What I mean by that is if for example there is a capital E, and one of the – horizontal line segments of the E are separated from the rest of the letter. Certain people write like this and an example of this is shown here:
In image 01_list.jpg



This "o" which is not fully joined is one such candidate in the entire image show on the right.

Another example is if there is cursive writing where letters build and join into each other, thereby creating a concatenated symbol. This will also affect the performance of this algorithm. Though there are no examples from the list of images in the image file, here is an example of this situation:



Where the computer would not instinctively parse through this letter by letter, but rather continue of a loop that checks for neighbouring pixels of the letter which in this case would follow the cursive paths like a dynamite fuse meandering between rocks.

Q4.2
Answer:
Done in the code

Q4.3
Answer:

Q4.4
Answer:

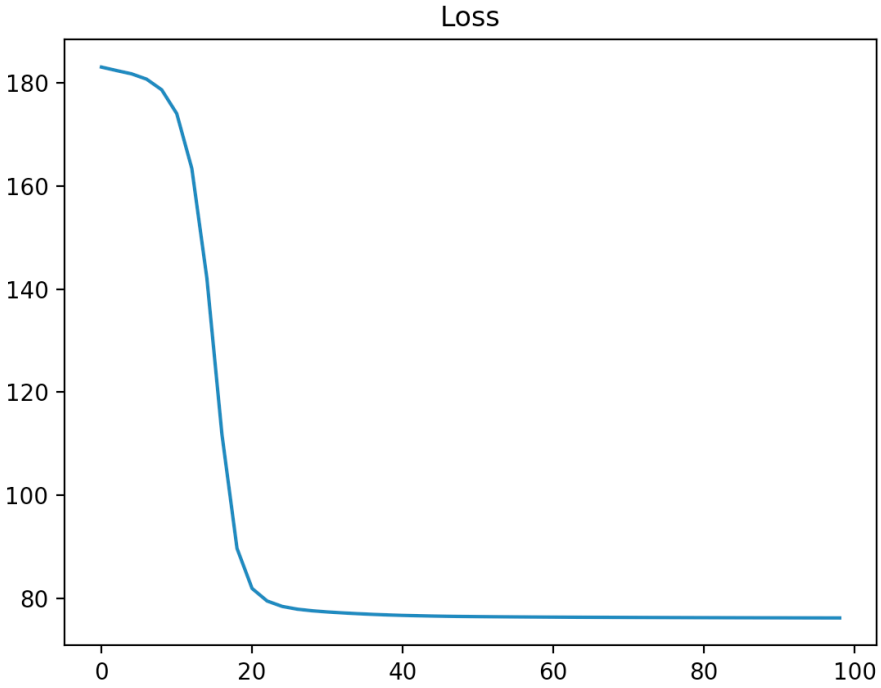
Q5.1.1
Answer:
Done in the code

Q5.1.2
Answer:
Done in the code

Q5.2

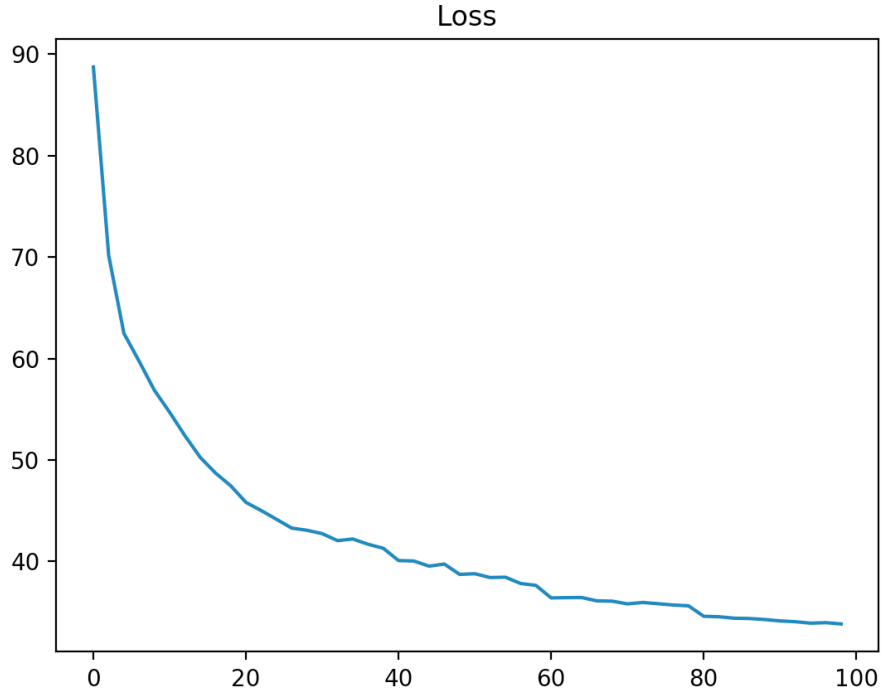
Answer:

This is what it looks like for a 3e-8 learning, rate:



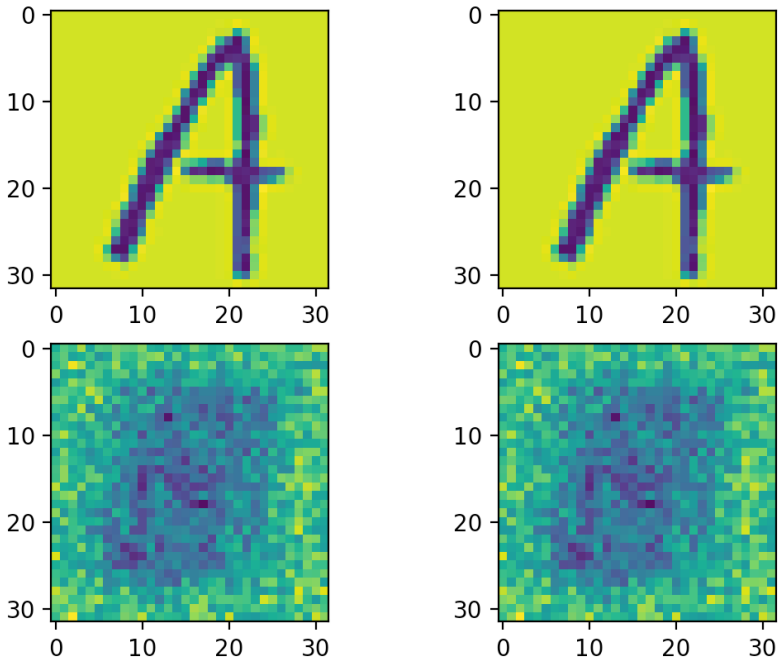
If falls steeply, followed by a very aggressive asymptote in loss after around 20 epochs towards a loss of 80. (Stuck in a non-important local minimum)

I then decided to redo the test but with a faster learning rate: 0.00001 and obtained:

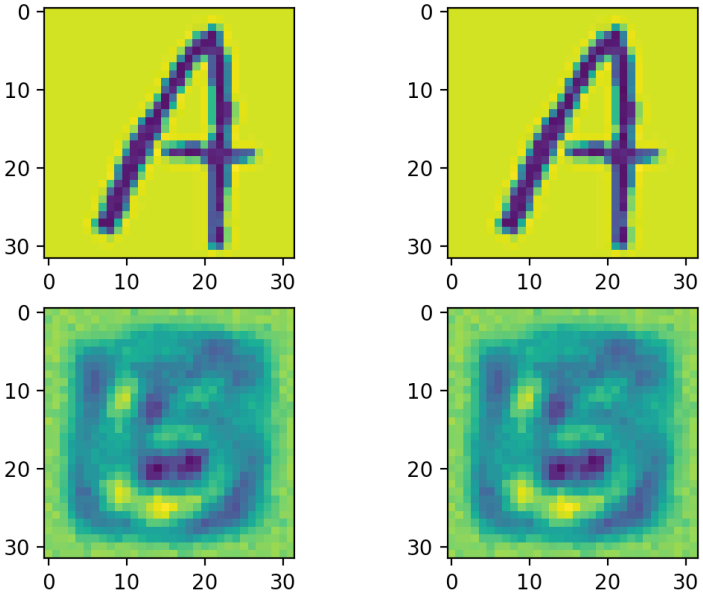


Q5.3.1
Answer

With the initial learning rate we obtain something that looks like:



Clearly nothing much is going on right, the filters seem to be in their nascent early phases wrt structuring the filter to classify input images correctly. So with the faster learning rate (1e-5) we get this instead:



Q5.3.2
Answer:

I obtained: `1.6770842016232048`

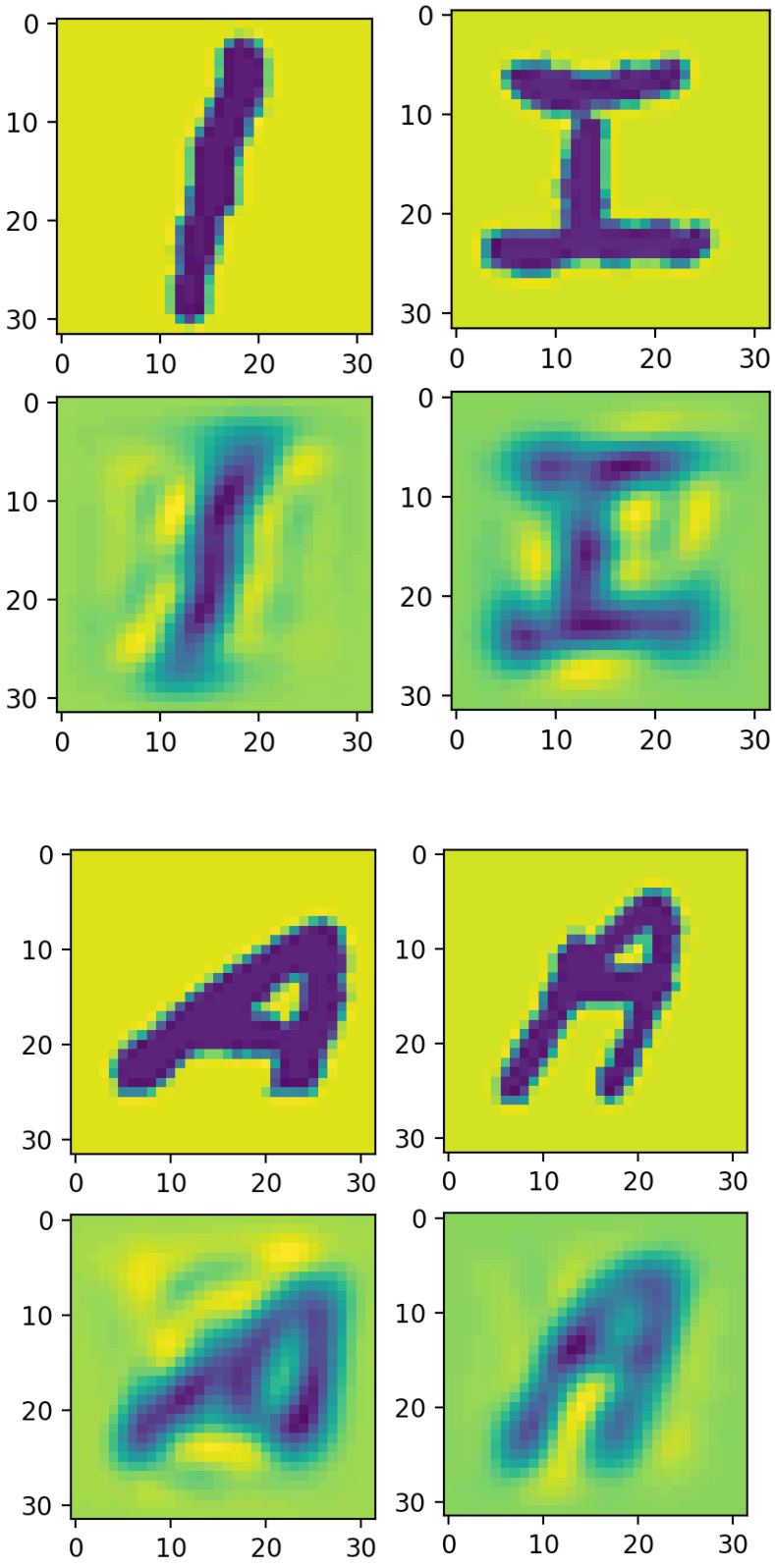
Q6.1

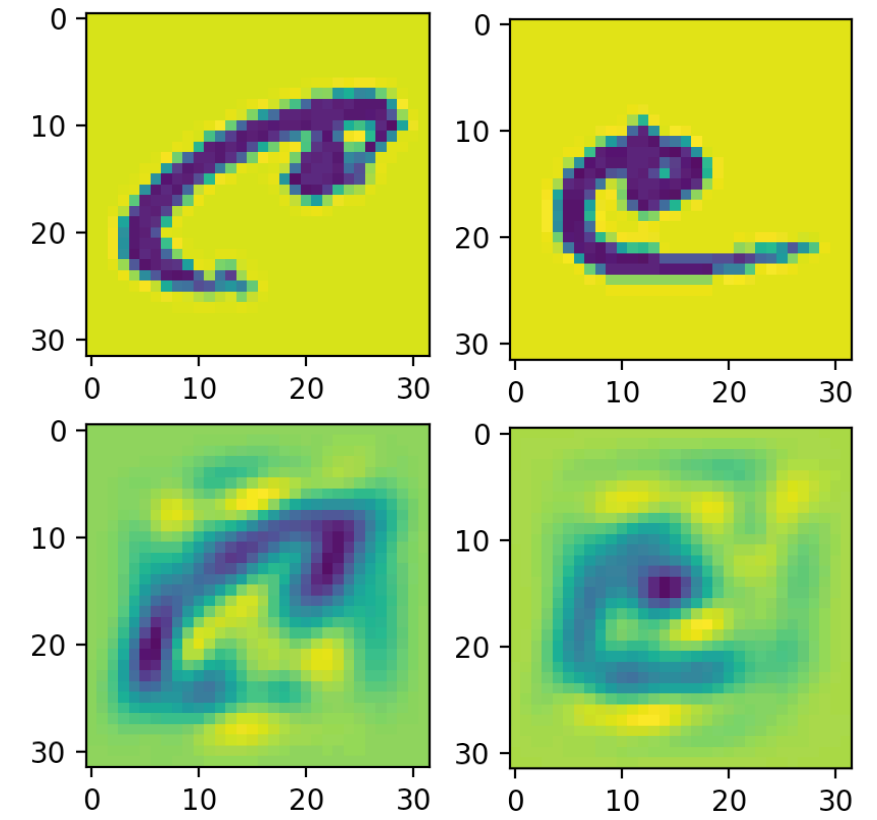
Answer:

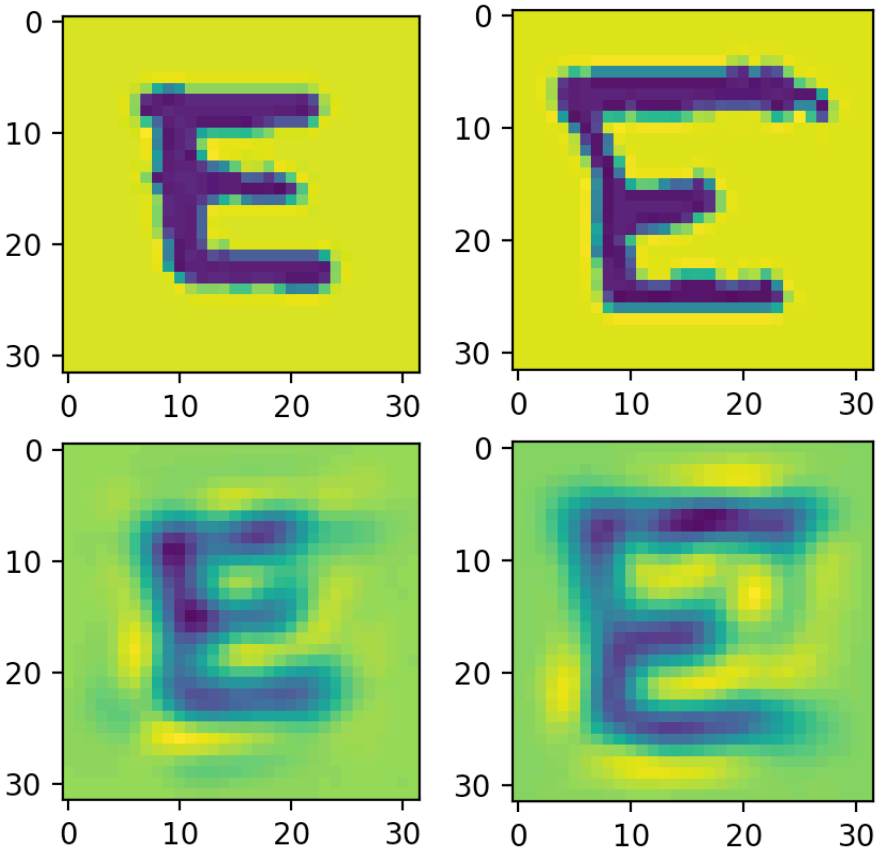
The rank is 32, and the dimensions of the projection matrix are 1024x32. The rest of the instructions to this question have been implemented in the code.

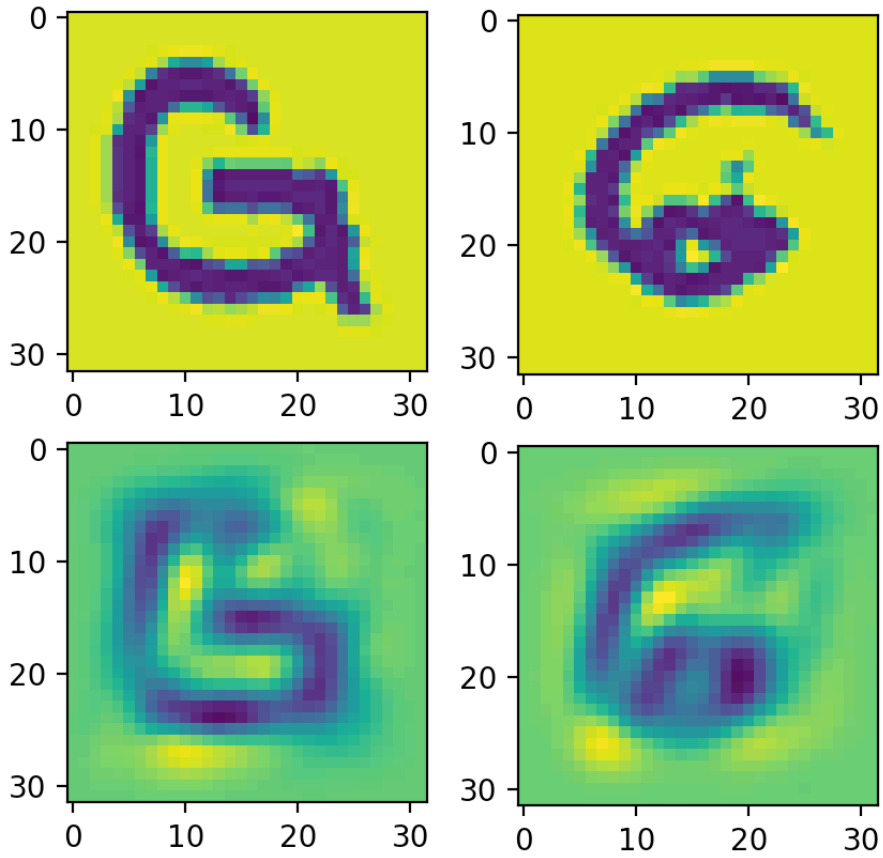
Q6.2

Answer:




0
10
20
30
0
10
20
30


0
10
20
30
0
10
20
30



Just like the autoencoder, the output images are blurry and have a pixelated texture. This is due to the high degree of detail compression by funnelling low level features through the algorithm. The background color of the output is also green because of the compressed combination of blue characters with yellow backgrounds (=green).

Q6.3
Answer:
16,284 which is higher than that of the auto-encoder. It is like this because the principal component analysis expose the "principal" parts that have the biggest impact, while auto-encoders intrinsically minimize across the board, the difference between input and output image. Therefore you have more refined detail that is preserved and so a lower peak signal to noise ratio.

Q6.4
Answer:

Auto-encoder = 67712
[(1024*32 + 32*32)*2 + (32+32)*2]
PCA = 32768
[1024*32]

The difference is attributed to PCA trying to find the most fundamentally simplified representation of the data.