

16720: Computer Vision

HW5

Andrew ID: hdhruva

Q1.1

Q1.1 Theory [3 points] Prove that softmax is invariant to translation, that is

$$\text{softmax}(x) = \text{softmax}(x + c) \quad \forall c \in \mathbb{R}$$

Softmax is defined as below, for each index i in a vector x .

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Often we use $c = -\max x_i$. Why is that a good idea? (Tip: consider the range of values that numerator will have with $c = 0$ and $c = -\max x_i$)

$$\begin{aligned} \text{softmax}(x_i + c) &= \frac{e^{(x_i + c)}}{\sum_j e^{(x_j + c)}} \\ &= \frac{e^{x_i} \cdot e^c}{\sum_j e^{x_j} \cdot e^c} = \frac{e^{x_i} \cdot \cancel{e^c}}{\cancel{e^c} \sum_j e^{x_j}} \\ &= \frac{e^{x_i}}{\sum_j e^{x_j}} \\ &= \text{softmax}(x_i) \end{aligned}$$

\therefore Softmax is invariant to translation.

When $c = -\max(x_i) \rightarrow \text{numerator} = e^{(x_i - \max(x_i))} = \frac{e^{x_i}}{e^{\max(x_i)}} \left\{ \begin{array}{l} e^{x_i} \in [0, \infty) \\ \in [0, 1] \end{array} \right.$

Range of numerator is constrained, could help by avoiding overflow while coding.

Q1.2

Q1.2 Theory [3 points] Softmax can be written as a three step processes, with $s_i = e^{x_i}$, $S = \sum s_i$ and $\text{softmax}(x_i) = \frac{1}{S}s_i$.

- As $x \in \mathbb{R}^d$, what are the properties of $\text{softmax}(x)$, namely what is the range of each element? What is the sum over all elements?
- One could say that "softmax takes an arbitrary real valued vector x and turns it into a _____".
- Can you see the role of each step in the multi-step process now? Explain them.

- Range of each element is $[0, 1]$

Sum over all elements is 1

- Probability distribution function

- Step 1: $s_i = e^{x_i} \Rightarrow$ Calculates the outcome of x_i by calculating exponential value

Step 2: $S = \sum s_i \Rightarrow$ Sum of all exponential values or the total outcome

Step 3: $\text{softmax}(x_i) = \frac{1}{S}s_i \Rightarrow$ Calculates the probability of x_i

Q1.3

Q1.3 Theory [3 points] Show that multi-layer neural networks without a non-linear activation function are equivalent to linear regression.

Output of a multi-layer neural network with a non-linear activation function $\rightarrow y = f(Wx + b)$

$y = f(Wx + b)$

↑ some non-linear function ↑ weight matrix ↑ bias

$y = f(W_n x_n + b_n) = f(W_n f(W_{n-1} x_{n-1} + b_{n-1}) + b_n)$

↑ n layer

Say we remove the non-linear activation function

$$\begin{aligned} y &= W_n x_n + b_n = W_n (W_{n-1} x_{n-1} + b_{n-1}) + b_n \\ &= W_n W_{n-1} x_{n-1} + W_n b_{n-1} + b_n \\ &= W_n W_{n-1} (W_{n-2} x_{n-2} + b_{n-2}) + W_n b_{n-1} + b_n \\ &= W_n W_{n-1} W_{n-2} x_{n-2} + W_n W_{n-1} b_{n-2} + W_n b_{n-1} + b_n \\ &= \vdots \end{aligned}$$

$$y = \prod_{i=1}^n W_i x + \sum_{j=1}^n \left(\prod_{k=j+1}^n W_k \right) b_j$$

$$y = W'x + b'$$

\therefore Without a non-linear activation function this is equivalent to solving a single layer with weight w' & bias b' ie. linear regression

Q1.4

Q1.4 Theory [3 points] Given the sigmoid activation function $\sigma(x) = \frac{1}{1+e^{-x}}$, derive the gradient of the sigmoid function and show that it can be written as a function of $\sigma(x)$ (without having access to x directly)

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

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

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

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

$$= \sigma(x) - \sigma(x)^2$$

$$\boxed{\sigma'(x) = \sigma(x)(1 - \sigma(x))}$$

Q1.5

Q1.5 Theory [12 points] Given $y = Wx + b$ (or $y_i = \sum_{j=1}^d x_j W_{ij} + b_i$), and the gradient of some loss J with respect y , show how to get $\frac{\partial J}{\partial W}$, $\frac{\partial J}{\partial x}$ and $\frac{\partial J}{\partial b}$. Be sure to do the derivatives with scalars and re-form the matrix form afterwards. Here is some notional suggestions.

$$\frac{\partial J}{\partial y} = \delta \in \mathbb{R}^{k \times 1} \quad W \in \mathbb{R}^{k \times d} \quad x \in \mathbb{R}^{d \times 1} \quad b \in \mathbb{R}^{k \times 1}$$

We won't grade the derivative with respect to b but you should do it anyways, you will need it later in the assignment.

$$y_i = \sum_{j=1}^d x_j W_{ij} + b_i \quad , \quad \frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial y_i} \cdot \frac{\partial y_i}{\partial W_{ij}} = \delta_k \cdot x_j$$

$$\therefore \frac{\partial J}{\partial W} = \begin{bmatrix} \frac{\partial J}{\partial W_{11}} & \dots & \frac{\partial J}{\partial W_{k1}} \\ \vdots & & \vdots \\ \frac{\partial J}{\partial W_{1d}} & \dots & \frac{\partial J}{\partial W_{kd}} \end{bmatrix} = \begin{bmatrix} \frac{\partial J}{\partial y_1} x_1 & \dots & \frac{\partial J}{\partial y_k} x_1 \\ \vdots & & \vdots \\ \frac{\partial J}{\partial y_1} x_d & \dots & \frac{\partial J}{\partial y_k} x_d \end{bmatrix} = \begin{bmatrix} \delta_1 x_1 & \dots & \delta_k x_1 \\ \vdots & & \vdots \\ \delta_1 x_d & \dots & \delta_k x_d \end{bmatrix} = x \delta^T \in \mathbb{R}^{d \times k}$$

$$\frac{\partial J}{\partial x_j} = \frac{\partial J}{\partial y_i} \cdot \frac{\partial y_i}{\partial x_j} = \delta_k \sum_{j=1}^k W_{ij}$$

$$\therefore \frac{\partial J}{\partial x} = \begin{bmatrix} \frac{\partial J}{\partial x_1} \\ \vdots \\ \frac{\partial J}{\partial x_d} \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^k \delta_k W_{1j} \\ \vdots \\ \sum_{j=1}^k \delta_k W_{dj} \end{bmatrix} = W \delta \in \mathbb{R}^{d \times 1}$$

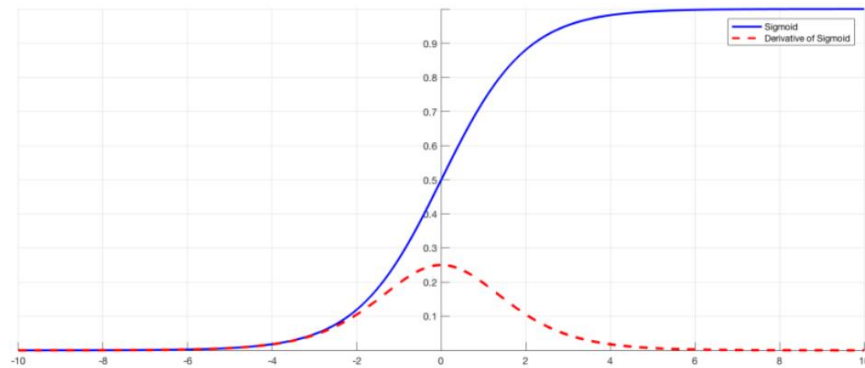
$$\frac{\partial J}{\partial b} = \begin{bmatrix} \frac{\partial J}{\partial b_1} \\ \vdots \\ \frac{\partial J}{\partial b_k} \end{bmatrix} = \begin{bmatrix} \frac{\partial J}{\partial y_1} \cdot \frac{\partial y_1}{\partial b_1} \\ \vdots \\ \frac{\partial J}{\partial y_k} \cdot \frac{\partial y_k}{\partial b_k} \end{bmatrix} = \begin{bmatrix} \frac{\partial J}{\partial y_1} \\ \vdots \\ \frac{\partial J}{\partial y_k} \end{bmatrix} = \delta \in \mathbb{R}^{k \times 1}$$

Q1.6

Q1.6 Theory [4 points] When the neural network applies the elementwise activation function (such as sigmoid), the gradient of the activation function scales the backpropagation update. This is directly from the chain rule, $\frac{d}{dx}f(g(x)) = f'(g(x))g'(x)$.

1. Consider the sigmoid activation function for deep neural networks. Why might it lead to a "vanishing gradient" problem if it is used for many layers (consider plotting Q1.4)?
2. Often it is replaced with $\tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$. What are the output ranges of both tanh and sigmoid? Why might we prefer tanh?
3. Why does tanh(x) have less of a vanishing gradient problem? (plotting the derivatives helps! for reference: $\tanh'(x) = 1 - \tanh(x)^2$)
4. tanh is a scaled and shifted version of the sigmoid. Show how $\tanh(x)$ can be written in terms of $\sigma(x)$.

1.



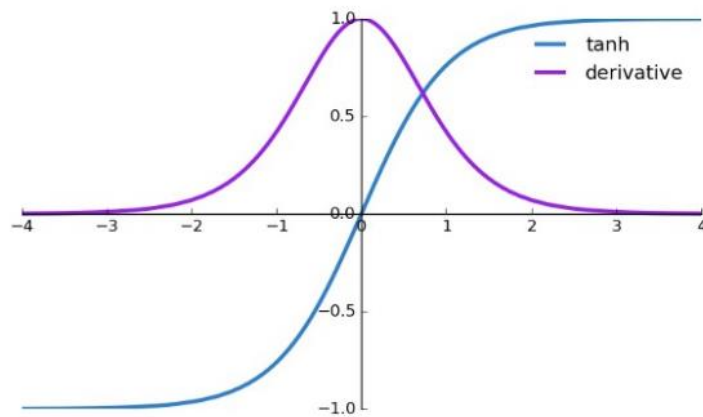
We see from the plot above that the maximum value of $\text{sigmoid}'(x)$ is 0.25, and the $\text{sigmoid}'(x)$ value is smaller if inputs of the sigmoid function become larger or smaller. For many layers, there will be many small derivative values multiplied together and therefore the gradient decreases significantly as we propagate down the initial layers in accordance to chain rule, which will lead to the "vanishing gradient" problem.

2. Output range of sigmoid $\Rightarrow (0, 1)$

$$\tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} = 1 - \frac{2e^{-2x}}{1 + e^{-2x}} = 1 - \frac{2}{1 + e^{2x}} \Rightarrow \text{output range} \Rightarrow (-1, 1)$$

$\tanh(x)$ is preferred since it retains the sign of the input, the output range being symmetric. This avoids bias in the gradients and therefore it is less likely to have "gradient vanish"

$$3. \quad \tanh'(x) = 1 - \tanh(x)^2$$



Here the max value of the derivative of tanh as seen in the graph above is 1.0

\therefore tanh has a higher value for gradient around zero, therefore it is less likely to have a vanishing gradient problem

$$4. \quad \sigma(x) = \frac{1}{1+e^{-x}} \quad , \quad \tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$$

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

$$\boxed{\tanh(x) = 2\sigma(2x) - 1}$$

Q2.1.1

Q2.1.1 Theory [3 points] Why is it not a good idea to initialize a network with all zeros? If you imagine that every layer has weights and biases, what can a zero-initialized network output after training?

When you initialize all the weights to zero, then every hidden layer in the network will be propagated by zeros independent of the input. So when propagated backwards, then while updating the weights, all of the weights will get updated by the same gradient value, which will only affect the scale of the weight vector. This will prevent the network from learning, since the error values that will be backpropagated will be the same, and then all the weights will be updated by the same amount. We will not be able to converge to a solution this way.

Q2.1.2

Q2.1.2 Code [3 points] In `python/nn.py`, implement a function to initialize neural network weights with Xavier initialization [1], where $Var[w] = \frac{2}{n_{in} + n_{out}}$ where n is the dimensionality of the vectors and you use a **uniform distribution** to sample random numbers (see eq 16 in the paper).

Q2.1.3

Q2.1.3 Theory [2 points] Why do we initialize with random numbers? Why do we scale the initialization depending on layer size (see near Fig 6 in the paper)?

When we initialize the weights with random numbers, we are avoiding the symmetry problem we would run into if all the weights are initialized by the same value. This would yield a better chance of converging to an optimal solution.

We scale the initialization depending on the layer size since it will help keep the variance of the weights to a minimum. Therefore, the output through forward propagation or the gradients through back propagation will also be controlled, therefore avoiding vanishing problem or gradient explosion.

Q2.2.1

Q2.2.1 Code [4 points] In `python/nn.py`, implement sigmoid, along with forward propagation for a single layer with an activation function, namely $y = \sigma(XW + b)$, returning the output and intermediate results for an $N \times D$ dimension input X , with examples along the rows, data dimensions along the columns.

Q2.2.2

Q2.2.2 Code [3 points] In `python/mn.py`, implement the softmax function. Be sure to use the numerical stability trick you derived in Q1.1 softmax.

Q2.2.3

Q2.2.3 Code [3 points] In `python/nn.py`, write a function to compute the accuracy of a set of labels, along with the scalar loss across the data. The loss function generally used for classification is the cross-entropy loss.

$$L_f(\mathbf{D}) = - \sum_{(\mathbf{x}, \mathbf{y}) \in \mathbf{D}} \mathbf{y} \cdot \log(\mathbf{f}(\mathbf{x}))$$

Here \mathbf{D} is the full training dataset of data samples \mathbf{x} ($N \times 1$ vectors, N = dimensionality of data) and labels \mathbf{y} ($C \times 1$ one-hot vectors, C = number of classes).

Q2.3.1

Q2.3.1 Code [7 points] In `python/mn.py`, write a function to compute backpropagation for a single layer, given the original weights, the appropriate intermediate results, and given gradient with respect to the loss. You should return the gradient with respect to X so you can feed it into the next layer. As a size check, your gradients should be the same dimensions as the original objects.

Q2.4

Q2.4 Code [5 points] In `python/nn.py`, write a function that takes the entire dataset (`x` and `y`) as input, and then split the dataset into random batches. In `python/run_q2.py`, write a training loop that iterates over the random batches, does forward and backward propagation, and applies a gradient update step.

Q2.5

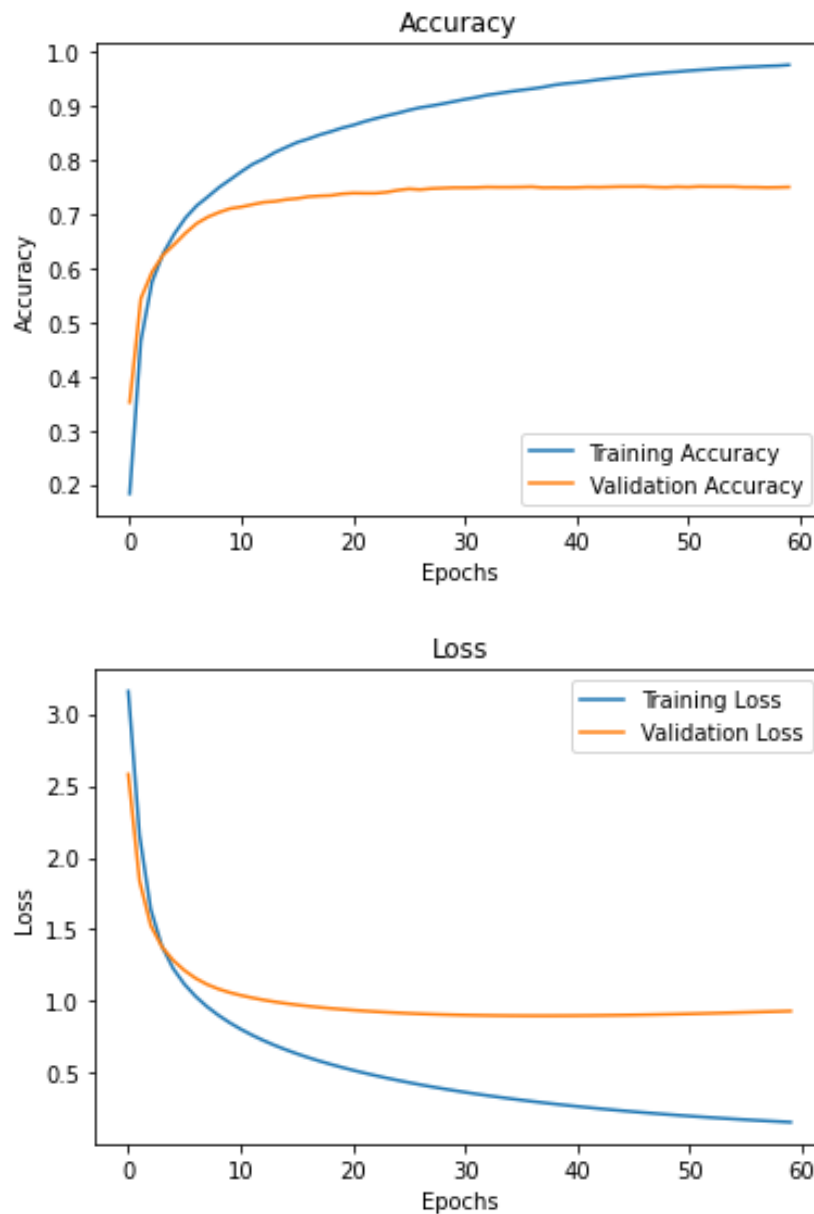
Q2.5 [5 points] In `python/run_q2.py`, implement a numerical gradient checker. Instead of using the analytical gradients computed from the chain rule, add ϵ offset to each element in the weights, and compute the numerical gradient of the loss with central differences. Central differences is just $\frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}$. Remember, this needs to be done for each scalar dimension in all of your weights independently.

Q3.1

Q3.1 Code [5 points] Train a network from scratch. Use a single hidden layer with 64 hidden units, and train for at least 30 epochs. Modify the script to generate two plots: one showing the accuracy on both the training and validation set over the epochs, and the other showing the cross-entropy loss averaged over the data. The x-axis should represent the epoch number, while the y-axis represents the accuracy or loss. With these settings, you should see an accuracy on the validation set of at least 75%.

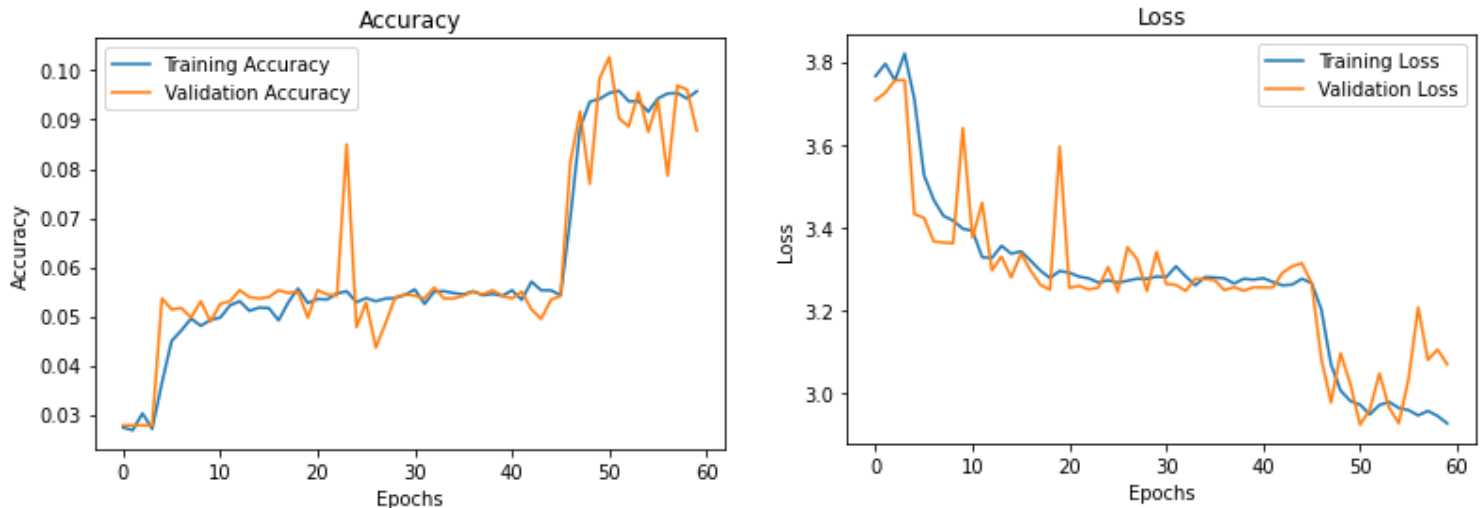
Learning Rate = 0.006, batch_size = 32, iterations = 60

Validation accuracy = 75.6 %, Test Accuracy = 76 %



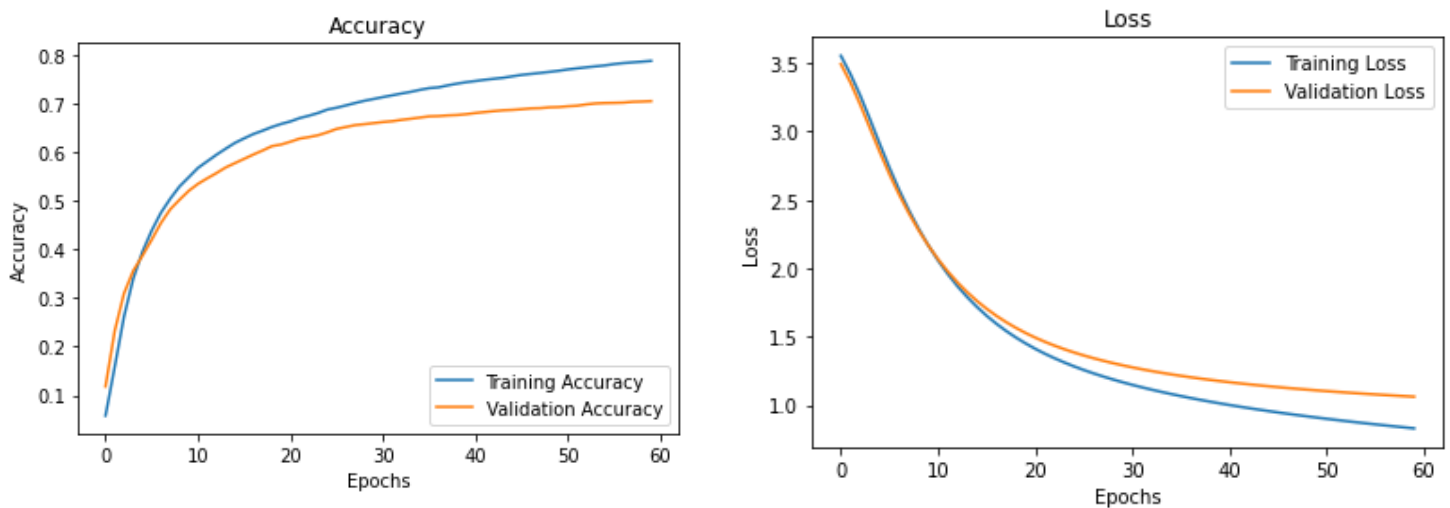
Q3.2 **Q3.2 Writeup [3 points]** Use your modified training script to train three networks, one with your best learning rate, one with 10 times that learning rate and one with one tenth that learning rate. Include all 4 plots in your writeup. Comment on how the learning rates affect the training, and report the final accuracy of the best network on the test set.

Learning Rate = 0.06, Test Accuracy = 8.8 %



Higher loss and lower accuracy was achieved above, and due to the large step size, the curve is not smooth.

Learning Rate = 0.0006, Test Accuracy = 71 %

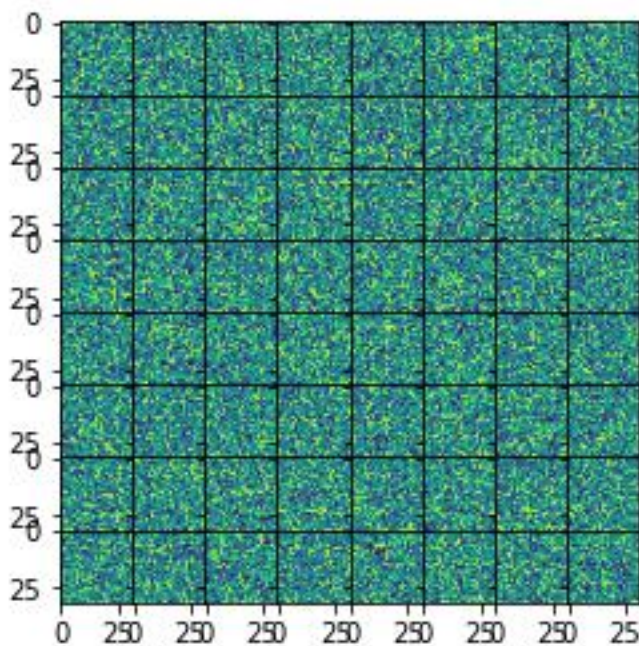


Here the curves are quite smooth due to the smaller step size, although we did not converge to a lower loss within the number of iterations (60). The accuracy is good but it is not better than the original model (learning rate = 0.006).

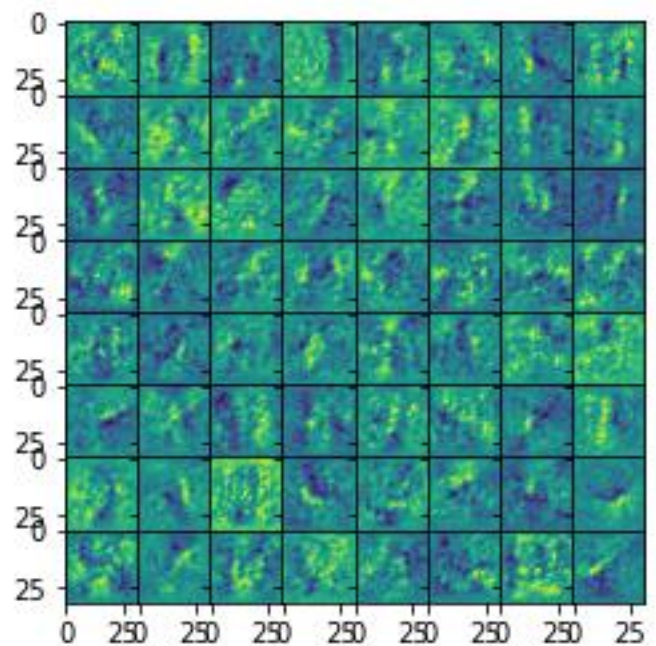
Model with learning rate = 0.006 has the best test accuracy.

Q3.3

Q3.3 Writeup [3 points] Visualize the first layer weights that your network learned (using `reshape` and `ImageGrid`). Compare these to the network weights immediately after initialization. Include both visualizations in your writeup. Comment on the learned weights. Do you notice any patterns?



Initialization Weights



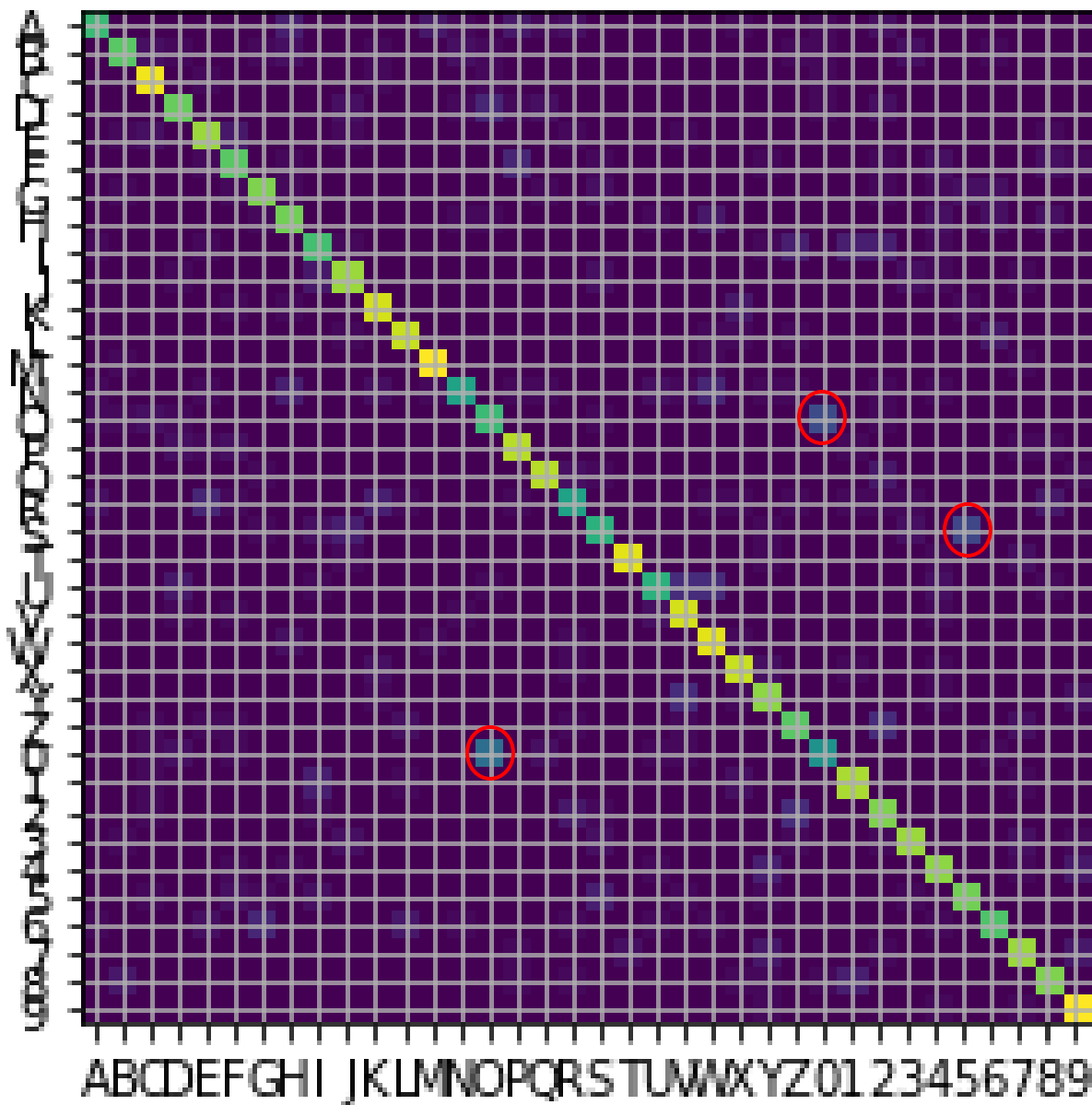
Training Weights

From the figures above it is evident that with the initialization weights, we see a random noisy pattern in the data, whereas after some training, the weights seem to establish some patterns. In the training weights figure, most of the data looks less noisy or less random.

Q3.4

Q3.4 Writeup [3 points] Visualize the confusion matrix for your best model. Comment on the top few pairs of classes that are most commonly confused.

The brighter the squares are, the larger the number in that grid. Since the main diagonal looks to be the heaviest, the results look reasonably good. I have highlighted some pairs that seem the most confused in the figure below. Most confused looks like pair (O, 0).



Q4.1

Q4.1 Theory [3 points] The method outlined above is pretty simplistic, and makes several assumptions. What are two big assumptions that the sample method makes. In your writeup, include two example images where you expect the character detection to fail (either miss valid letters, or respond to non-letters).

1. The letters do not overlap, they are isolated

✗  ✓ 

✗  ✓ 

2. Letters should be of similar size

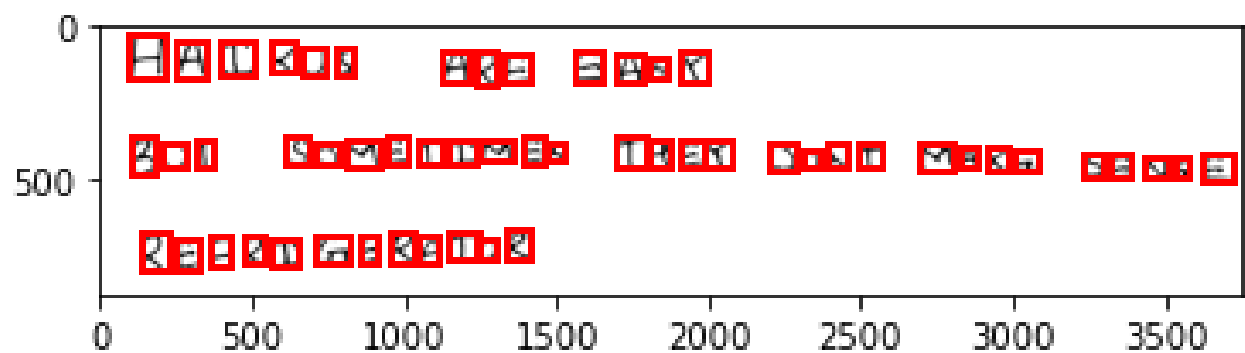
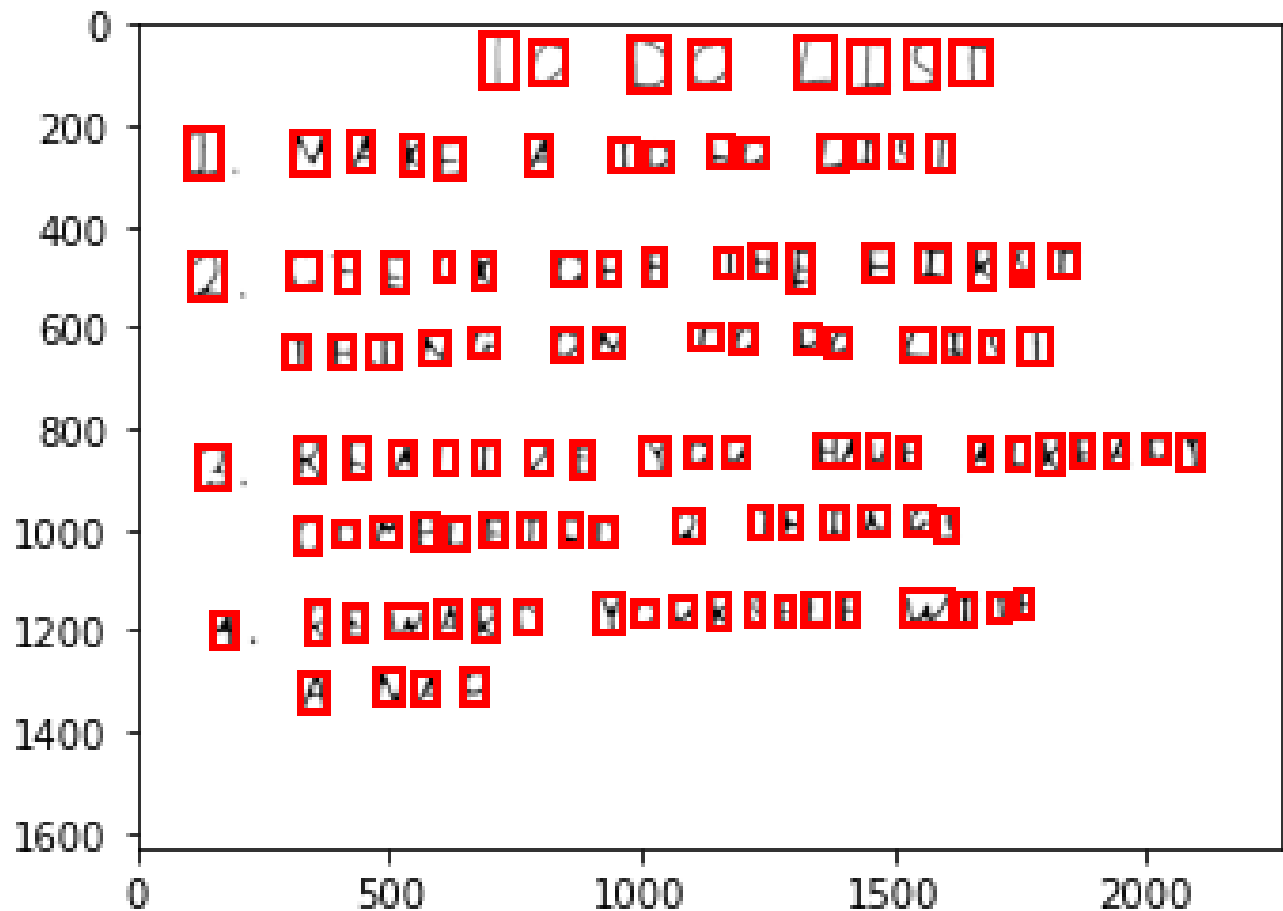
✗  ✓ 

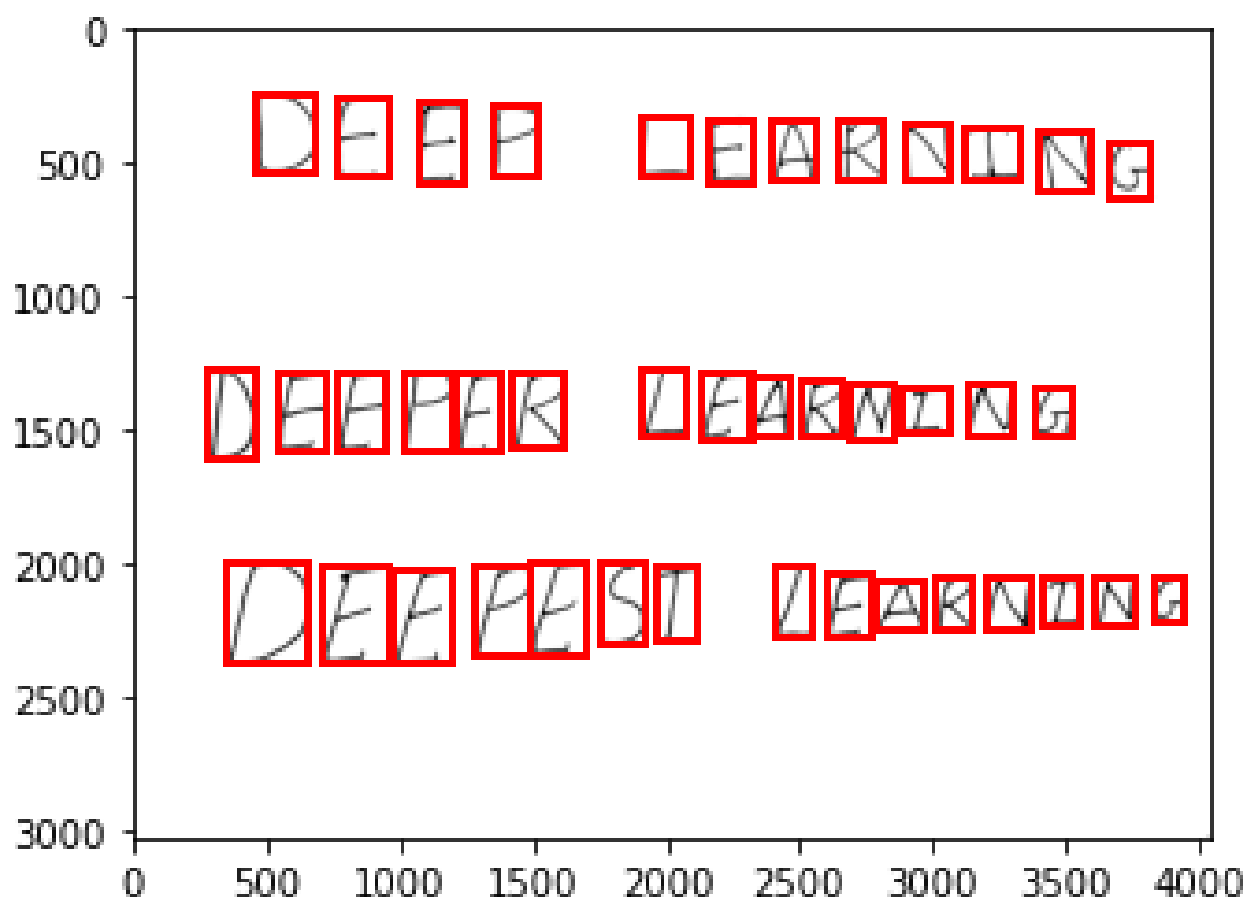
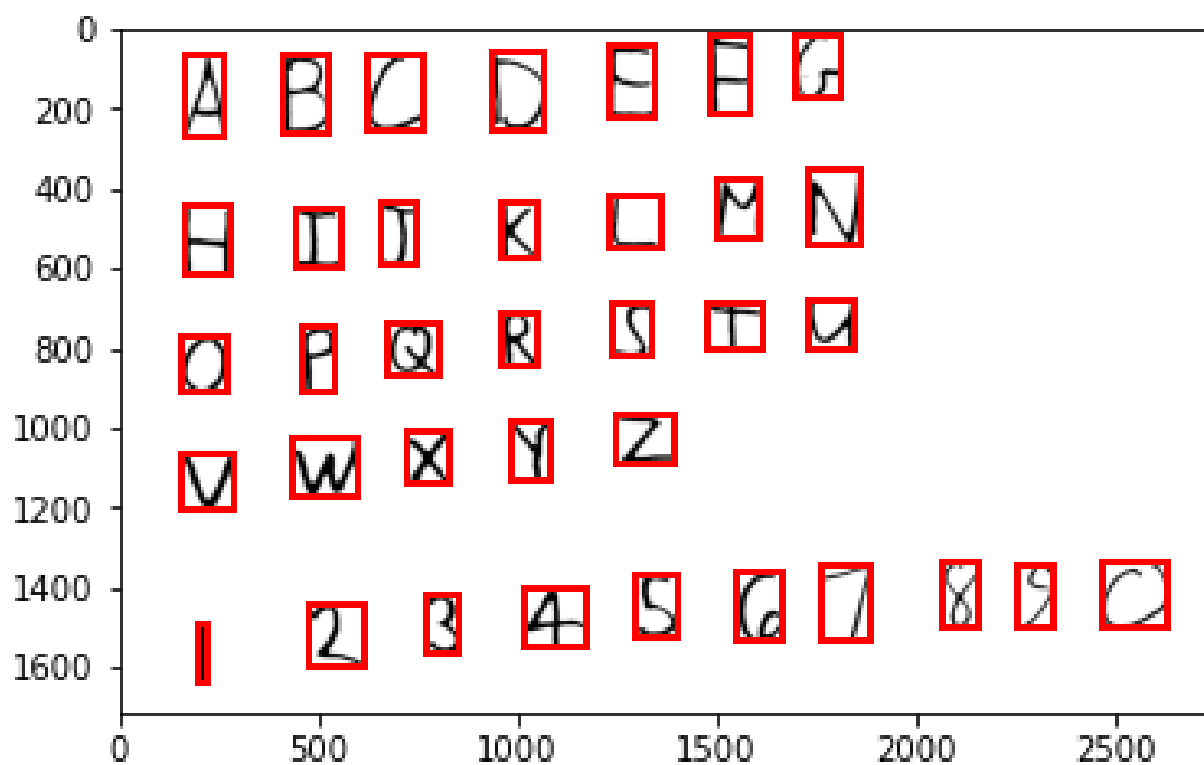
Q4.2

Q4.2 Code [10 points] In `python/q4.py`, implement the function to find letters in the image. Given an RGB image, this function should return bounding boxes for all of the located handwritten characters in the image, as well as a binary black-and-white version of the image `im`. Each row of the matrix should contain `[y1,x1,y2,x2]` the positions of the top-left and bottom-right corners of the box. The black and white image should be floating point, 0 to 1, with the characters in black and background in white.

Q4.3

Q4.3 Writeup [5 points] Run `findLetters(..)` on all of the provided sample images in `images/`. Plot all of the located boxes on top of the image to show the accuracy of your `findLetters(..)` function. Include all the result images in your writeup.





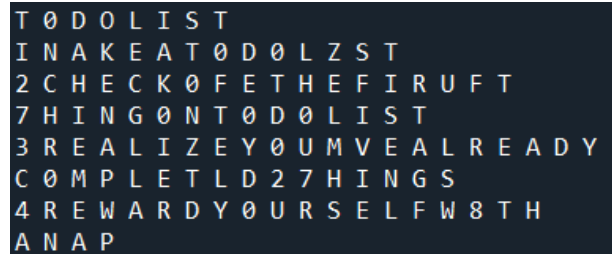
Q4.4

Q4.4 Code/Writeup [8 points] Use `python/run_q4.py` for this question. Now you will load the image, find the character locations, classify each one with the network you trained in **Q3.1**, and return the text contained in the image. Be sure you try to make your detected images look like the images from the training set. Visualize them and act accordingly.

Run your `run_q4` on all of the provided sample images in `images/`. Include the extracted text in your writeup.

“01_list.jpg”

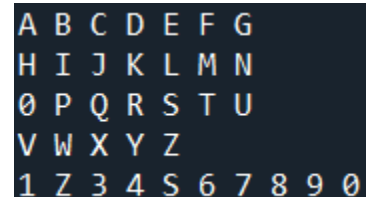
T O D O L I S T
I N A K E A T O D O L Z S T
2 C H E C K O F E T H E F I R U F T
7 H I N G O N T O D O L I S T
3 R E A L I Z E Y O U M V E A L R E A D Y
C O M P L E T L D 2 7 H I N G S
4 R E W A R D Y O U R S E L F W 8 T H
A N A P



T O D O L I S T
I N A K E A T O D O L Z S T
2 C H E C K O F E T H E F I R U F T
7 H I N G O N T O D O L I S T
3 R E A L I Z E Y O U M V E A L R E A D Y
C O M P L E T L D 2 7 H I N G S
4 R E W A R D Y O U R S E L F W 8 T H
A N A P

“02_letters.jpg”

A B C D E F G
H I J K L M N
O P Q R S T U
V W X Y Z
1 Z 3 4 S 6 7 8 9 0



A B C D E F G
H I J K L M N
O P Q R S T U
V W X Y Z
1 Z 3 4 S 6 7 8 9 0

“03_haiku.jpg”

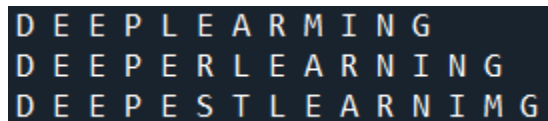
H A I K U S A R E E A S Y
B U T S O M E T I M E S T R E Y D D N T M A K R S E N 5 E
R E F R I G E R A T O R



H A I K U S A R E E A S Y
B U T S O M E T I M E S T R E Y D D N T M A K R S E N 5 E
R E F R I G E R A T O R

“04_deep.jpg”

D E E P L E A R M I N G
D E E P E R L E A R N I N G
D E E P E S T L E A R N I M G



D E E P L E A R M I N G
D E E P E R L E A R N I N G
D E E P E S T L E A R N I M G

Q5.1.1

Q5.1.1 Code [5 points] Due to the difficulty in training auto-encoders, we have to move to the $\text{relu}(x) = \max(x, 0)$ activation function. It is provided for you in `util.py`. Implement a 2 hidden layer autoencoder where the layers are

- 1024 to 32 dimensions, followed by a ReLU
- 32 to 32 dimensions, followed by a ReLU
- 32 to 32 dimensions, followed by a ReLU
- 32 to 1024 dimensions, followed by a sigmoid (this normalizes the image output for us)

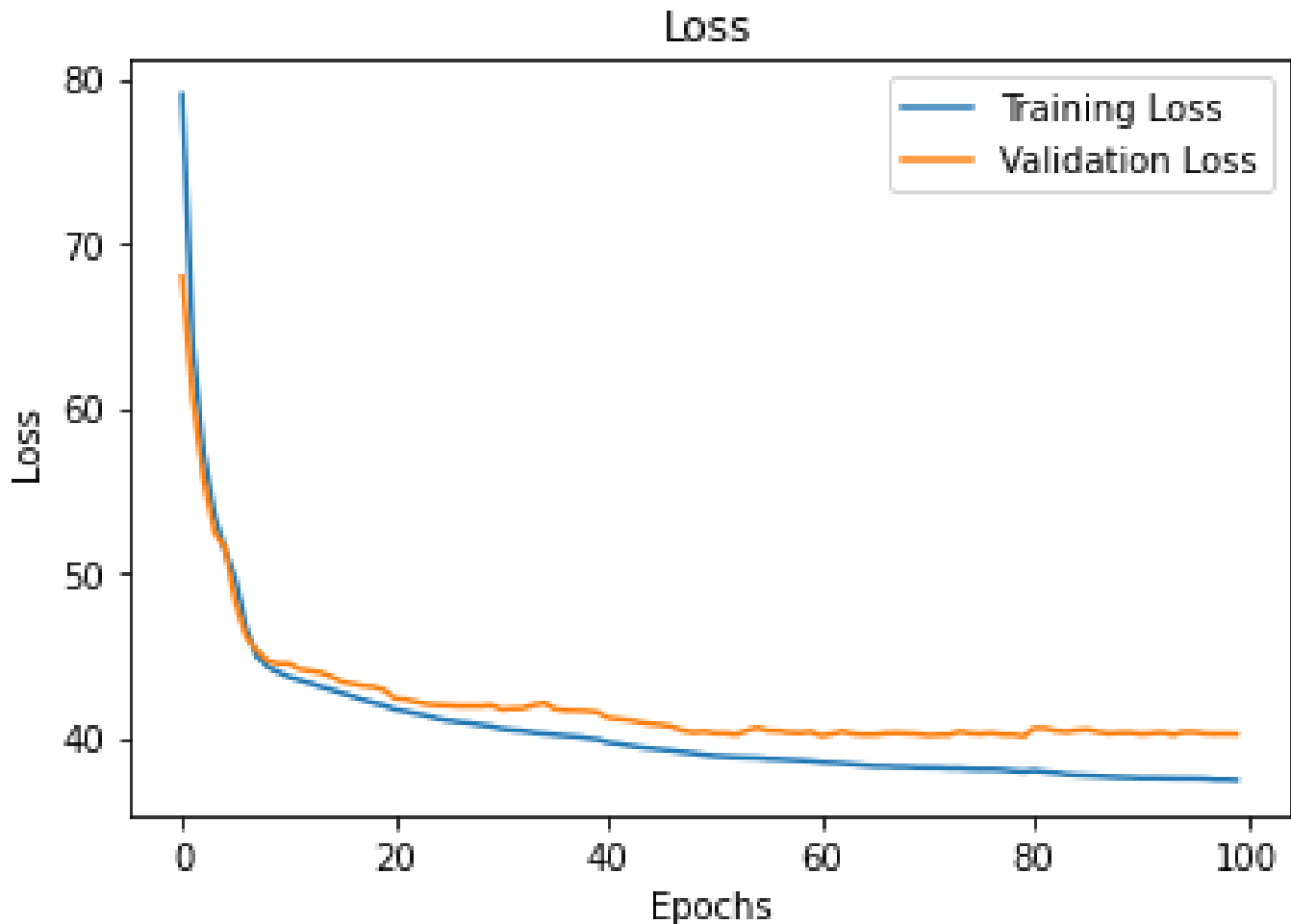
The loss function that you're using is total squared error for the output image compared to the input image (they should be the same!).

Q5.1.2

Q5.1.2 Code [5 points] To help even more with convergence speed, we will implement [momentum](#). Now, instead of updating $W = W - \alpha \frac{\partial J}{\partial W}$, we will use the update rules $M_W = 0.9M_W - \alpha \frac{\partial J}{\partial W}$ and $W = W + M_W$. To implement this, populate the parameters dictionary with zero-initialized momentum accumulators, one for each parameter. Then simply perform both update equations for every batch.

Q5.2

Q5.2 Writeup/Code [5 points] Using the provided default settings, train the network for 100 epochs. What do you observe in the plotted training loss curve as it progresses?

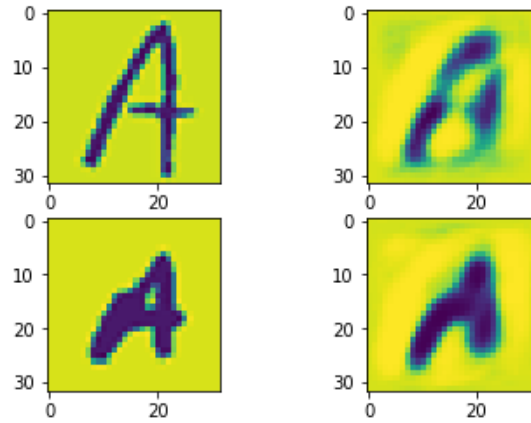


With the default settings, the data seems to converge well, and the training and validation loss follow a similar trajectory, therefore indicating a consistent model. The curves are slightly bumpy, but smooth overall.

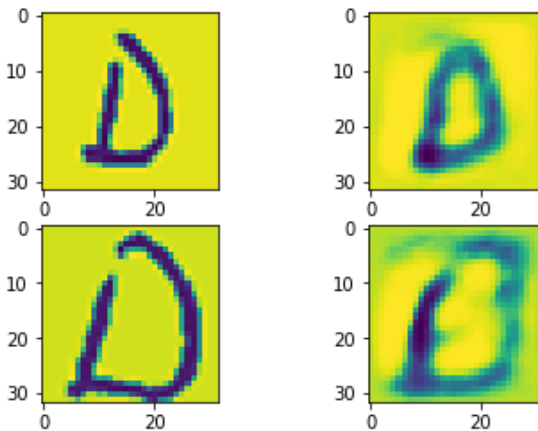
Q5.3.1

Q5.3.1 Writeup/Code [5 points] Now let's evaluate how well the autoencoder has been trained. Select 5 classes from the total 36 classes in the validation set and for each selected class include in your report 2 validation images and their reconstruction. What differences do you observe that exist in the reconstructed validation images, compared to the original ones?

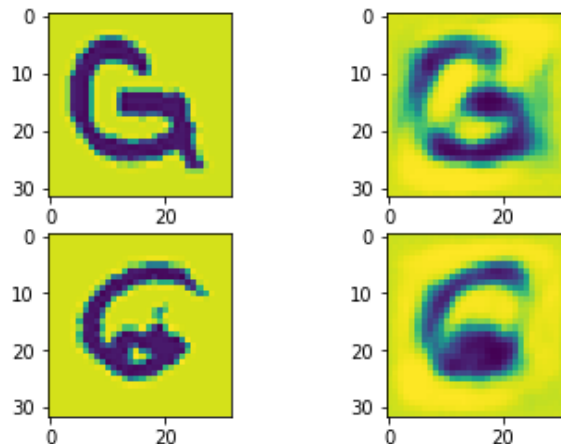
Class "A"



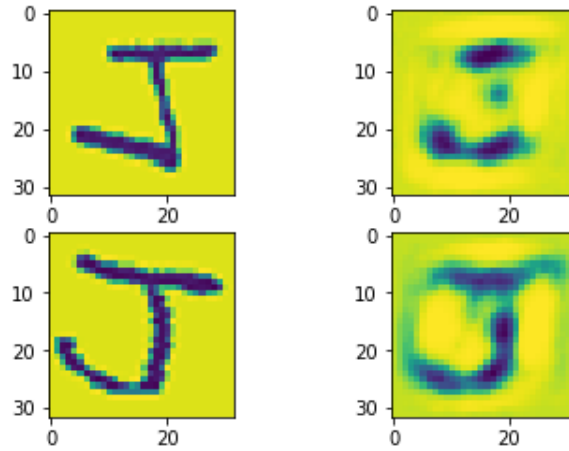
Class "D"



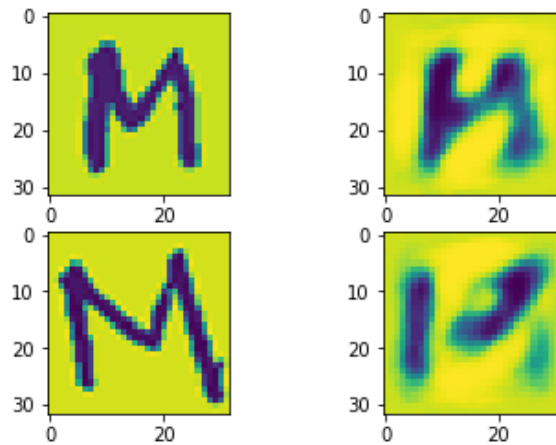
Class "G"



Class "J"



Class "M"



The reconstruction by the autoencoder is a blurry version of the clearer original image, which confirms that the autoencoder can only provide an approximate version of the original image. The reconstructed image edges are less distinct and more blurred into the background but some of them can still be identified as the appropriate letters by the human eye.

Q5.3.2

Q5.3.2 Writeup [5 points] Let's evaluate the reconstruction quality using Peak Signal-to-noise Ratio (PSNR). PSNR is defined as

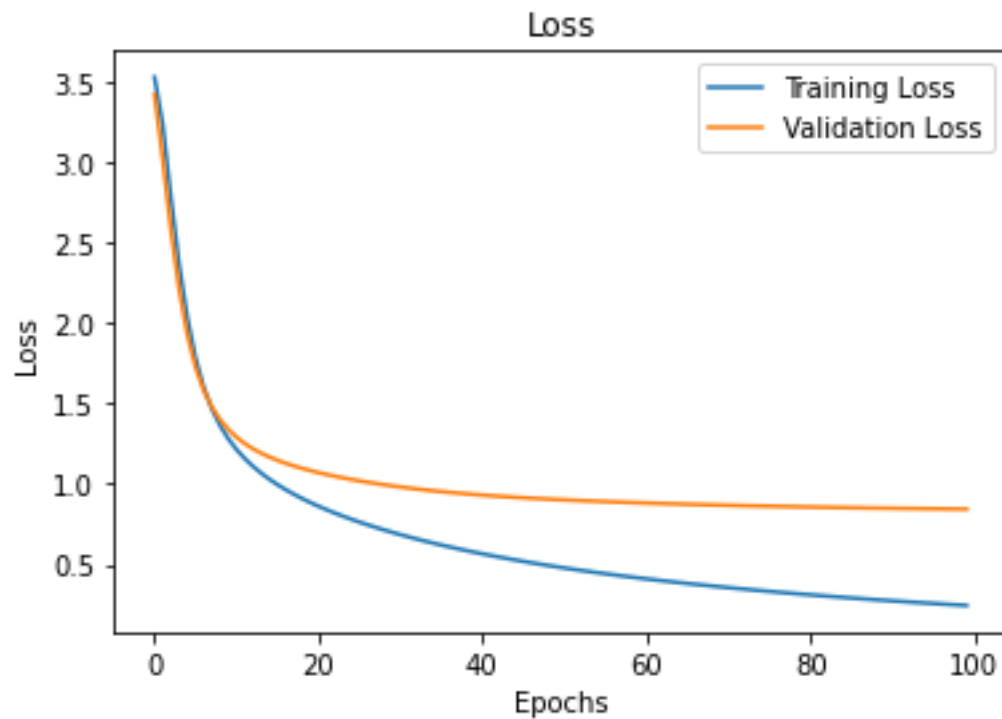
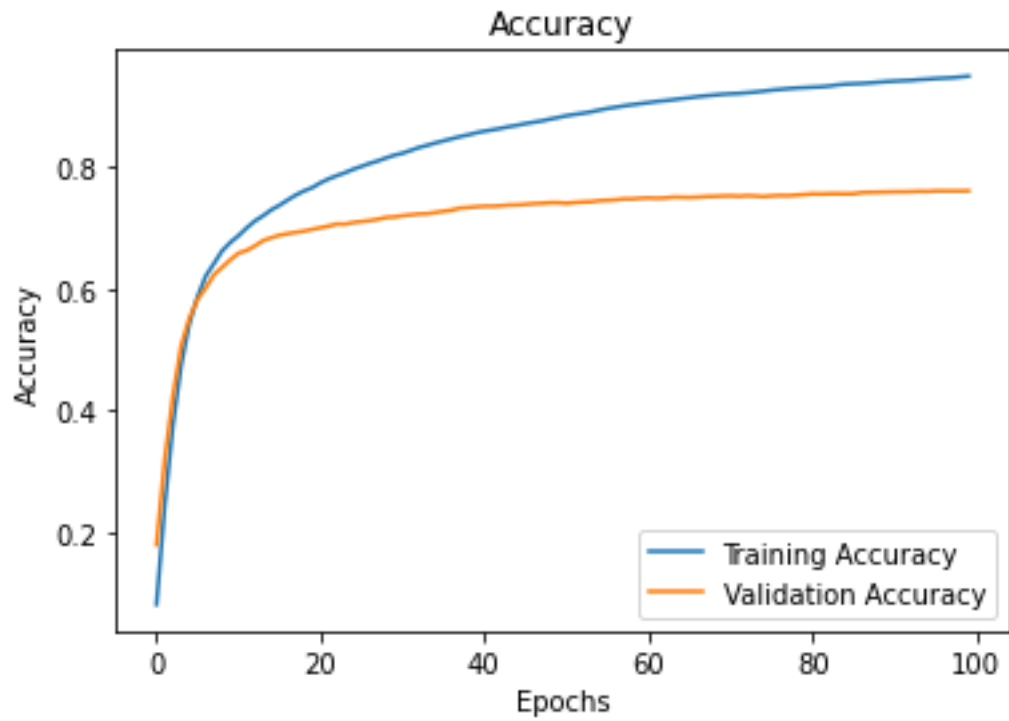
$$\text{PSNR} = 20 \times \log_{10}(\text{MAX}_I) - 10 \times \log_{10}(\text{MSE}) \quad (1)$$

where MAX_I is the maximum possible pixel value of the image, and MSE (mean squared error) is computed across all pixels. You may use [skimage.measure.compare_psnr](#) for convenience. Report the average PSNR you get from the autoencoder across all images in the validation set.

Avg PSNR = 15.680482369019678

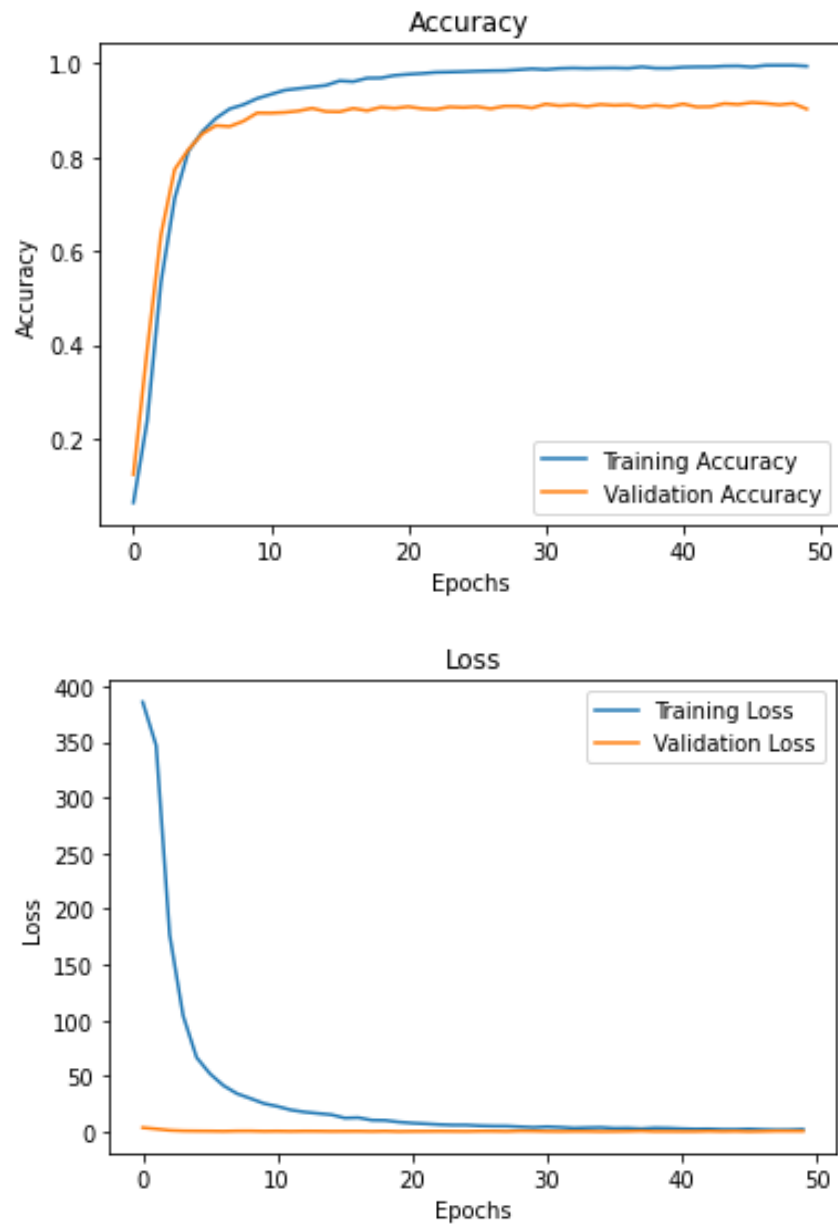
Q6.1.1

Q6.1.1 Code/Writeup [5 points] Re-write and re-train your fully-connected network on the included NIST36 in PyTorch. Plot training accuracy and loss over time.



Q6.1.2

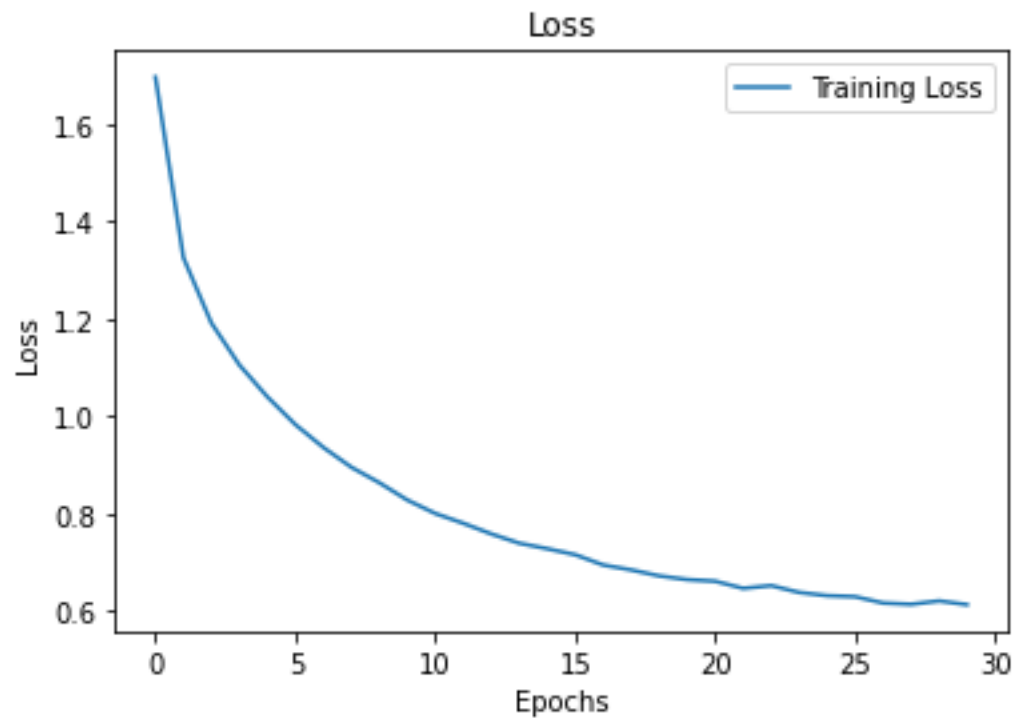
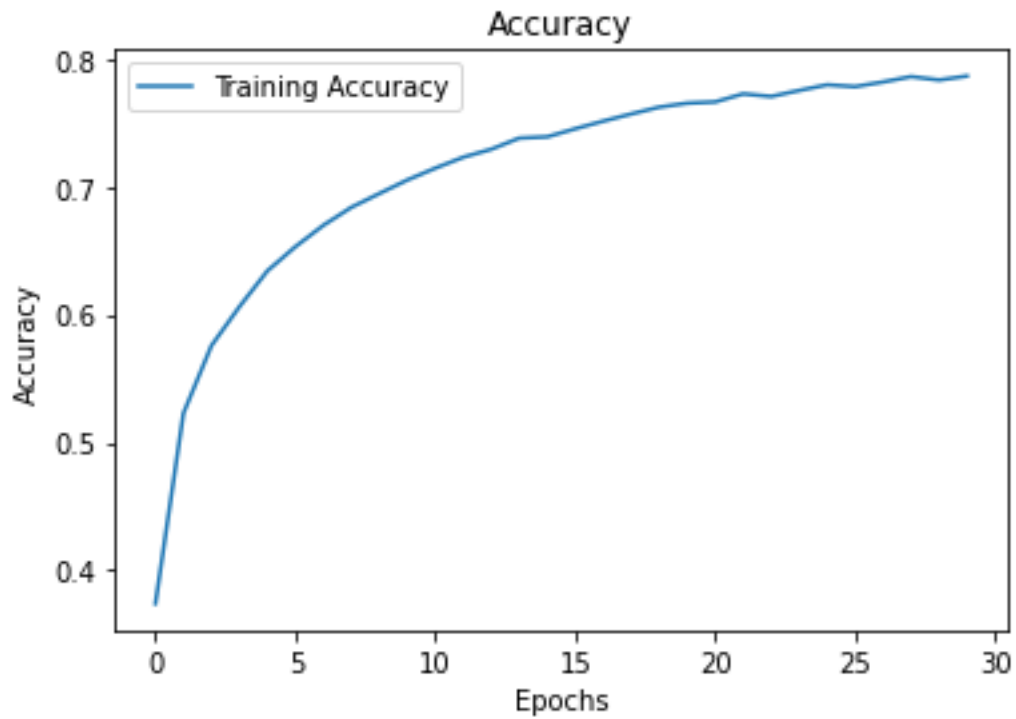
Q6.1.2 Code/Writeup [5 points] Train a convolutional neural network with PyTorch on the included NIST36 dataset. Compare its performance with the previous fully-connected network.



The CNN performance is much better than the previously fully connected network. The validation accuracy is 90.2% which is a lot higher than the 76% accuracy achieved in the fully connected network.

Q6.1.3

Q6.1.3 Code/Writeup [5 points] Train a convolutional neural network with PyTorch on CIFAR-10 (`torchvision.datasets.CIFAR10`). Plot training accuracy and loss over time.



Q6.1.4

Q6.1.4 Code/Writeup [10 points] In Homework 1, we tried scene classification with the bag-of-words (BoW) approach on a subset of the SUN database. Use the same dataset in HW1, and implement a convolutional neural network with PyTorch for scene classification. Compare your result with the one you got in HW1, and briefly comment on it.

Q6.2.1

Q6.2.1 Code/Writeup [5 points] Fine-tune a single layer classifier using pytorch on the [flowers 17](#) (or [flowers 102!](#)) dataset using [squeezeNet1.1](#), as well as an architecture you've designed yourself (*3 conv layers, followed 2 fc layers, it's standard [slide 6](#)*) and trained from scratch. How do they compare?

We include a script in `scripts/` to fetch the flowers dataset and extract it in a way that `PyTorch ImageFolder` can consume it, see [an example](#), from `data/oxford-flowers17`. You should look at how SqueezeNet is [defined](#), and just replace the classifier layer. There exists a pretty good example for [fine-tuning](#) in PyTorch.