16720 — PS5

Connor W. Colombo (cwcolomb)

November 24th 2020

1 Theory

Q1.1

$$\text{let: } x \text{ be a } d \times 1 \text{ vector, and } i, j \in [1, d] \in \mathbb{Z}$$
$$\text{let: } \exists\, c \in \mathbb{R}$$
$$\because \text{softmax}(x) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$
$$\text{softmax}(x+c) = \frac{e^{x_i+c}}{\sum_j e^{x_j+c}} = \frac{e^{x_i}e^{c}}{\sum_j e^{x_j}e^{c}} = \frac{e^{x_i}e^{c}}{e^{c}\sum_j e^{x_j}}$$
$$\rightarrow \text{softmax}(x+c) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$
$$\therefore \text{softmax}(x+c) = \text{softmax}(x)$$

Since $\text{softmax}(x)$ involves using each x_i as a power, it has the potential for becoming numerically unstable by involving numerically impractical large or numerically impractical small values as intermediates in the calculation, especially since division is involved and one could end up dividing a very large number by a very small number or vice versa, even though all results will fall in the range from 0 to 1. To prevent this, subtracting the largest element of the vector from every element will ensure that all exponents are at most 0 and, thus, that the numerator (e^{x_i}) is always between 0 and 1 (instead of 0 to ∞ for $c = 0$ and $x_i \in \mathbb{R}$) and that the denominator is always between 0 and d, thereby increasing the numerical stability of the algorithm.

Q1.2

$$\text{let: } x \in \mathbb{R}^d$$
$$\text{let: } s_i = e^{x_i}$$
$$\text{let: } S = \sum s_i$$
$$\text{let: } \operatorname{softmax}(x_i) = \frac{1}{S} s_i$$

- The range of each element is between 0 and 1 **and** the sum of all elements is 1.
  This can be seen since, as described in Q1.1, $\operatorname{softmax}(x) = \operatorname{softmax}(x - \max x_i)$ in which case each value can be thought of as a number between 0 and 1 being divided a number between 0 and d; thus, all $\operatorname{softmax}(x_i)$ will be between 0 and 1. Furthermore, it can be seen that the sum is 1 since:

$$\sum_i \operatorname{softmax}(x_i) = \sum_i \frac{1}{S} s_i = \frac{1}{S} \sum_i s_i = \frac{1}{\sum s_i} \sum s_i = 1$$

- One could say that "softmax takes an arbitrary real valued vector x and turns it into a **normalized vector of probabilities (values between 0 and 1 which sum to 1)**".

-
  1. The first $s_i = e^{x_i}$ step maps each element from $(-\infty, \infty)$ to a positive number $(0, \infty)$.
  2. The second $S = \sum s_i$ step sums up all these remapped values to act as a normalizing constant.
  3. The third $\operatorname{softmax}(x_i) = \frac{1}{S} s_i$ step normalizes the vector of remapped values, mapping each of them from $(0, \infty)$ to $(0, 1]$ while ensuring that $\sum_i \operatorname{softmax}(x_i) = 1$.
     (note: $\operatorname{softmax}(x_i) = 1$ iff $d = 1$).

Q1.3

Let $\mathbf{x}_j$ be the vector of all neuron states in layer j, $x_{j,k}$ be the state of the k^{th} neuron in layer j, $\mathbf{w}_{j,k}$ be the vector of weights from all neurons in layer $j-1$ to the k^{th} neuron in layer j, and $b_{j,k}$ be the bias for neuron k in layer j.

Without a nonlinear activation function, the state of k^{th} neuron in layer j can be expressed as:

$$x_{j,k} = f(\mathbf{w}_{j,k}^T \mathbf{x}_{j-1} + b_{j,k}) = \mathbf{w}_{j,k}^T \mathbf{x}_{j-1} + b_{j,k}$$

Thus, one could express the state of all neurons in layer j with respect to the state of all neurons in layer $j-1$ as:

$$\mathbf{x}_j = \mathbf{W}_j \mathbf{x}_{j-1} + \mathbf{b}_j$$

where $\mathbf{W}_j$ is a matrix of vertically stacked $\mathbf{w}_{j,k}^T$ vectors and $\mathbf{b}_j$ is the vector of biases for every neuron in layer j as:

$$\mathbf{W}_j = \begin{bmatrix} \mathbf{w}_{j,1}^T \\ \mathbf{w}_{j,2}^T \\ \vdots \\ \mathbf{w}_{j,N}^T \end{bmatrix}, \quad \mathbf{b}_j = \begin{bmatrix} b_{j,1} \\ b_{j,2} \\ \vdots \\ b_{j,N} \end{bmatrix}$$

Accordingly, the state of all neurons in layer $j+1$ with respect to the state of all neurons in layer $j-1$ can be found to be:

$$\mathbf{x}_{j+1} = \mathbf{W}_{j+1}(\mathbf{W}_j \mathbf{x}_{j-1} + \mathbf{b}_j) + \mathbf{b}_{j+1} = \mathbf{W}_{j+1}\mathbf{W}_j \mathbf{x}_{j-1} + \mathbf{W}_{j+1}\mathbf{b}_j + \mathbf{b}_{j+1}$$

For the sake of completeness, the subsequent layer would then be:

$$\mathbf{x}_{j+2} = \mathbf{W}_{j+2}(\mathbf{W}_{j+1}\mathbf{W}_j \mathbf{x}_{j-1} + \mathbf{W}_{j+1}\mathbf{b}_j + \mathbf{b}_{j+1}) + \mathbf{b}_{j+2} = \mathbf{W}_{j+2}\mathbf{W}_{j+1}\mathbf{W}_j \mathbf{x}_{j-1} + \mathbf{W}_{j+2}\mathbf{W}_{j+1}\mathbf{b}_j + \mathbf{W}_{j+2}\mathbf{b}_{j+1} + \mathbf{b}_{j+2}$$

Thus, one could generalize this expression and restructure it relative to the first layer ($j-1=0$) and find the neuron states in layer n as:

$$\mathbf{x}_n = \mathbf{A}_n \mathbf{x}_0 + \beta_n \quad | \quad \mathbf{A}_n = \prod_{i=0}^{n} \mathbf{W}_i, \quad \beta_n = \sum_{i=1}^{n-1} \left(\prod_{j=i+1}^{n} \mathbf{W}_j \right) \mathbf{b}_i + \mathbf{b}_n$$

Thus the output (state of layer n) of a multi-layer neural network with any number of layers n without a non-linear activation function is equivalent to a linear expression ($\mathbf{x}_n = \mathbf{A}_n \mathbf{x}_0 + \beta_n$) and, as such, the process of solving for the network parameters would be equivalent to linear regression.

Q1.4

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

$$\therefore \nabla\sigma(x) = \left[\frac{d\sigma(x)}{dx}\right] = \frac{-1}{(1+e^{-x})^2}(-e^{-x})$$

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

$$\rightarrow \nabla\sigma(x) = \left(1 - \frac{1}{1+e^{-x}}\right)\frac{1}{1+e^{-x}}$$

$$\rightarrow \boldsymbol{\nabla\sigma(x) = (1-\sigma(x))\sigma(x)}$$

Q1.5

$$\text{let: } \left[\frac{\partial J}{\partial y}\right] = \delta$$

$$\text{let: } y \in \mathbb{R}^{k\times 1},\ x \in \mathbb{R}^{d\times 1},\ \delta \in \mathbb{R}^{k\times 1},\ W \in \mathbb{R}^{k\times d},\ b \in \mathbb{R}^{k\times 1}$$

$$\text{let: } \left[\frac{\partial J}{\partial x}\right] \in \mathbb{R}^{d\times 1},\ \left[\frac{\partial J}{\partial W}\right] \in \mathbb{R}^{k\times d},\ \left[\frac{\partial J}{\partial b}\right] \in \mathbb{R}^{k\times 1}$$

$$\text{let: } y = Wx + b \ \rightarrow\ y_i = \sum_{j=1}^{d} x_j W_{ij} + b_i$$

Notation note: since these gradients are being computed for the sake of gradient descent update, it is desirable for them to have the shape of the denominator (entity with respect to which the partial is being computed). Even though this doesn't necessarily conform to the standards for taking the derivative of a scalar with respect to a vector used in some other disciplines for Jacobians of vectors, it is consistent with the notation used for taking the gradient of a scalar-valued function of a vector. In order to be consistent with this notation, any derivatives of vectors taken with respect to a scalar used in the intermediate steps below will be considered row vectors (using the transpose of the shape of the numerator).

Solve for $\left[\frac{\partial J}{\partial x}\right]$:

$$\left[\frac{\partial J}{\partial x}\right] = \begin{bmatrix} \left[\frac{\partial J}{\partial x_1}\right] \\ \left[\frac{\partial J}{\partial x_2}\right] \\ \vdots \\ \left[\frac{\partial J}{\partial x_d}\right] \end{bmatrix}$$

$$\because \left[\frac{\partial J}{\partial x_j}\right] = \left[\frac{\partial J}{\partial y}\right]\left[\frac{\partial y}{\partial x_j}\right] = \delta_1\left[\frac{\partial y_1}{\partial x_j}\right] + \delta_2\left[\frac{\partial y_2}{\partial x_j}\right] + \ldots \delta_k\left[\frac{\partial y_k}{\partial x_j}\right]$$

$$\because \left[\frac{\partial y_i}{\partial x_j}\right] = W_{ij}$$

$$\left[\frac{\partial J}{\partial x_j}\right] = \sum_{i=1}^{k} \delta_i W_{ij}$$

$$\therefore \left[\frac{\partial J}{\partial x}\right] = \begin{bmatrix} \sum_{i=1}^{k} \delta_i W_{i1} \\ \sum_{i=1}^{k} \delta_i W_{i2} \\ \ldots \\ \sum_{i=1}^{k} \delta_i W_{id} \end{bmatrix}$$

$$\rightarrow \left[\frac{\partial J}{\partial x}\right] = W^T \delta$$

Solve for $\left[\frac{\partial J}{\partial W}\right]$:

$$\left[\frac{\partial J}{\partial W}\right] = \begin{bmatrix} \left[\frac{\partial J}{\partial W_{11}}\right] & \left[\frac{\partial J}{\partial W_{12}}\right] & \cdots & \left[\frac{\partial J}{\partial W_{1d}}\right] \\ \vdots & \vdots & \ddots & \vdots \\ \left[\frac{\partial J}{\partial W_{k1}}\right] & \left[\frac{\partial J}{\partial W_{k2}}\right] & \cdots & \left[\frac{\partial J}{\partial W_{kd}}\right] \end{bmatrix}$$

$$\because \left[\frac{\partial J}{\partial W_{ij}}\right] = \left[\frac{\partial J}{\partial y}\right]\left[\frac{\partial y}{\partial W_{ij}}\right] = \delta_1\left[\frac{\partial y_1}{\partial W_{ij}}\right] + \delta_2\left[\frac{\partial y_2}{\partial W_{ij}}\right] + \ldots \delta_k\left[\frac{\partial y_k}{\partial W_{ij}}\right]$$

$$\because \left[\frac{\partial y_m}{\partial W_{ij}}\right] = \begin{cases} x_j & \text{if } m = i, \\ 0 & \text{else} \end{cases}$$

$$\left[\frac{\partial J}{\partial W_{ij}}\right] = \delta_i x_j$$

$$\therefore \left[\frac{\partial J}{\partial W}\right] = \begin{bmatrix} \delta_1 x_1 & \delta_1 x_2 & \ldots & \delta_1 x_d \\ \vdots & \vdots & \ddots & \vdots \\ \delta_k x_1 & \delta_k x_2 & \ldots & \delta_k x_d \end{bmatrix}$$

$$\rightarrow \left[\frac{\partial J}{\partial W}\right] = \delta x^T$$

Solve for $\left[\frac{\partial J}{\partial b}\right]$:

$$\left[\frac{\partial J}{\partial b}\right] = \begin{bmatrix} \left[\frac{\partial J}{\partial b_1}\right] \\ \left[\frac{\partial J}{\partial b_2}\right] \\ \vdots \\ \left[\frac{\partial J}{\partial b_k}\right] \end{bmatrix}$$

$$\because \left[\frac{\partial J}{\partial b_j}\right] = \left[\frac{\partial J}{\partial y}\right]\left[\frac{\partial y}{\partial b_j}\right] = \delta_1\left[\frac{\partial y_1}{\partial b_j}\right] + \delta_2\left[\frac{\partial y_2}{\partial b_j}\right] + \ldots \delta_k\left[\frac{\partial y_k}{\partial b_j}\right]$$

$$\because \left[\frac{\partial y_i}{\partial b_j}\right] = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{else} \end{cases}$$

$$\left[\frac{\partial J}{\partial b_j}\right] = \delta_j$$

$$\therefore \left[\frac{\partial J}{\partial b}\right] = \begin{bmatrix} \delta_1 \\ \delta_2 \\ \vdots \\ \delta_k \end{bmatrix}$$

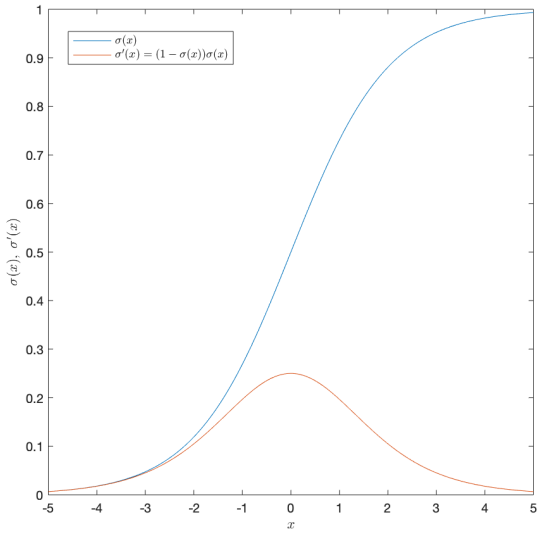
$$\rightarrow \left[\frac{\partial J}{\partial b}\right] = \delta$$

Q1.6

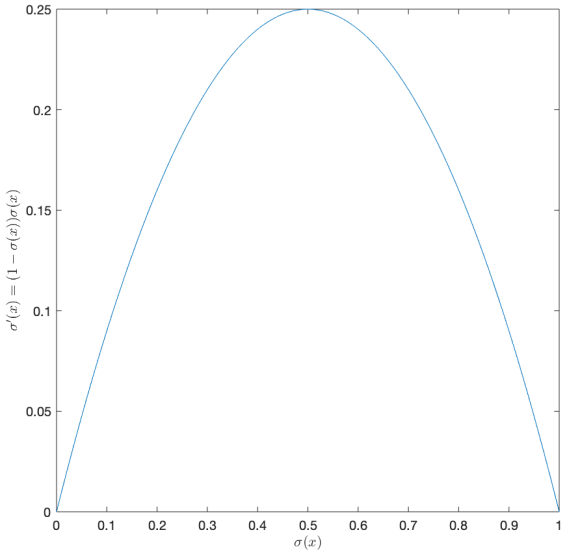
1. Consider the sigmoid activation function for deep neural networks. Why might it lead to a "vanishing gradient" problem if it is used for many layers (consider plotting Q1.4)?

As can be seen in Figure 1(a), for all input values, the sigmoid gradient is at most equal to and, at worst significantly smaller than the sigmoid function. Figure 1(b) shows that the sigmoid only peaks at 0.25 whereas the sigmoid peaks at 1. Lastly, Figure 1(c) shows that the ration of the sigmoid gradient to the sigmoid itself drops linearally to zero as the sigmoid value increases.

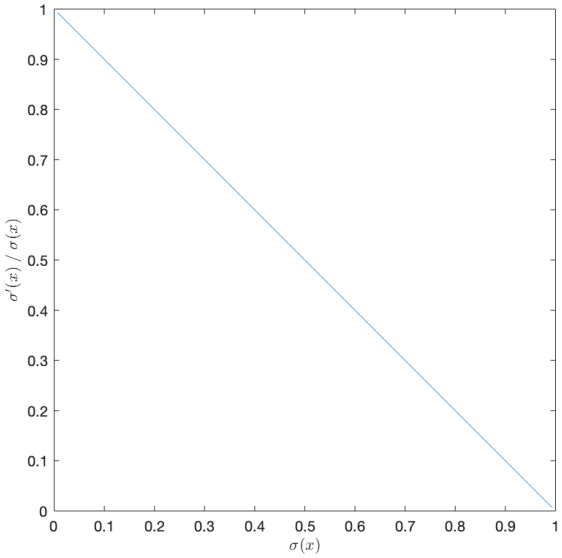
In deep neural networks where, during the back propagation step of learning, this gradient is successively multiplied, working backwards from the last layer as part of the larger gradient descent calculation. Considering all the above observations together, it seems that, due to the high likelihood of the magnitude of the sigmoid gradient being $\ll 1$, the gradient used to update early layers in the network would become significantly small and "vanish". As a result, one would expect this to cause the learning for these earlier layers to be significantly hampered.



(a) Plot of sigmoid function and gradient of sigmoid function.



(b) Gradient of sigmoid function as a function of sigmoid function.

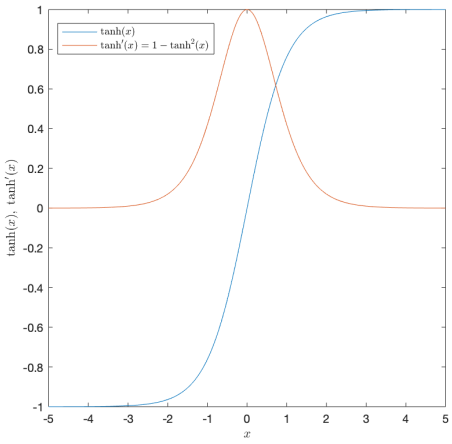


(c) Size of sigmoid gradient relative to size of sigmoid versus size of sigmoid.

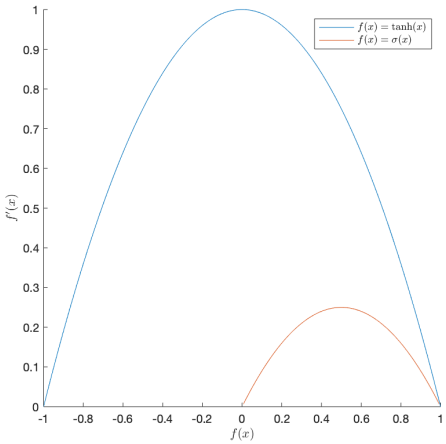
Figure 1: Analysis of Sigmoid Function Gradient with respect to Sigmoid Function.

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 ?**

$\sigma(x)$ has an output range of 0 to 1, whereas $\tanh(x)$ has an output range of -1 to 1. $\tanh(x)$ would often be preferable to $\sigma(x)$ since it exhibits the same desirable feature of $\sigma(x)$ — that its derivative can be written as a function of itself ($\tanh'(x) = 1 - \tanh^2(x)$) — but is less likely to exhibit the undesirable feature of $\sigma(x)$ — the vanishing gradient — since, as can be seen in Figure 2(b) below, its gradient maintains a considerably larger value for a much wider range of function values (its gradient value only drops below the largest value of the sigmoid gradient for $|\tanh(x)| > 0.866$).



(a) Plot of tanh function and gradient of tanh function.



(b) Function gradient as a function of the function for tanh and sigmoid functions.

Figure 2: Analysis of tanh Function Gradient with respect to tanh Function compared to Sigmoid Function.

3.Why does tanh(x) have less of a vanishing gradient problem? (plotting the derivatives helps! for reference: $\tanh'(x) = 1 - \tanh^2(x)$**)**

The gradient of $\tanh(x)$ is plotted in Figure 2(a) above. $\tanh(x)$ hass less of the vanishing gradient problem that $\sigma(x)$ has since, as can be seen in Figure 3 below, its gradient maintains a considerably larger value for a much wider range of function values. Specifically, its gradient value only drops below the largest value of the sigmoid gradient for $|\tanh(x)| > 0.866$. As a result, only very high or very low neuron activation states (over or under saturated) will lead to a sufficiently small gradient for the "vanishing gradient" problem to occur. That said, $\tanh(x)$ would still exhibit the vanishing gradient problem in very deep neural networks or in conditions where some of the neurons, especially the deeper neurons, are highly saturated.

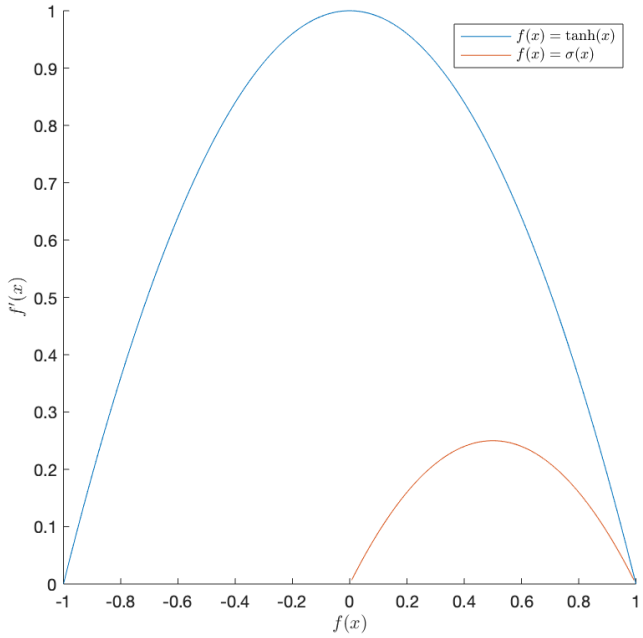


Figure 3: Function gradient as a function of the function for tanh and sigmoid functions (reproduction of Figure 2(b) above).

4.tanh is a scaled and shifted version of the sigmoid. Show how $tanh(x)$ can be written in terms of $\sigma(x)$‘

The following shows how $\tanh(x)$ is purely a version of $\sigma(x)$ that has been scaled down by 2 on the x-axis, scaled up by 2 on the y-axis, and then shifted down by 1 on the y axis.

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

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

However, this only represents $\tanh(x)$ as function of $\sigma(2x)$. Alternatively, $\tanh(x)$ can be written as function of purely $\sigma(x)$ as follows:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$
$$\rightarrow e^{-x} = \frac{1}{\sigma(x)} - 1$$
$$\therefore e^{-2x} = \left(\frac{1}{\sigma(x)} - 1\right)^2 = \frac{1}{\sigma^2(x)} - \frac{2}{\sigma(x)} + 1$$
$$\because \tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$$
$$\tanh(x) = \frac{\frac{2}{\sigma(x)} - \frac{1}{\sigma^2(x)}}{\frac{1}{\sigma^2(x)} - \frac{2}{\sigma(x)} + 2}$$
$$\therefore \boldsymbol{\tanh(x) = \frac{2\sigma(x) - 1}{2\sigma^2 x - 2\sigma(x) + 1}}$$

2 Implement a Fully Connected Network

2.1 Network Initialization

Q2.1.1 Theory

If all neuron weights are initialized to zero, no input state will have any effect on the cost, thus all the gradients will be equivalent (likely 0 for most activation functions), the cost will remain unchanged, and the network will not learn. Moreover, by simply initializing all weights to the same value (0 or otherwise), all the neurons will learn the same features as they will follow the same gradient descent trajectory (assuming they all share the same activation function).

As a result, after training, a zero-initialized network can output whatever it output before training since it will fail to learn as described in more detail above.

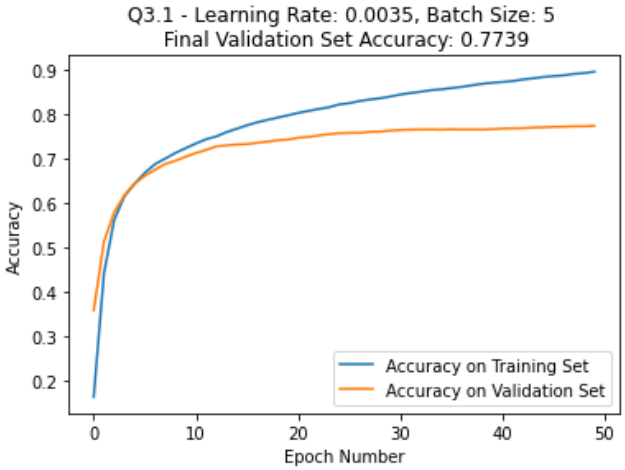
Q2.1.3 Theory

- We initialize with random numbers to avoid following the same gradient descent trajectory in every run and, thus, decrease the odds of getting stuck in the same local minima.
- We scale the initialization depending on layer size using Xavier's "normalized initialization" to ensure that variance of the neuron states ($Var[x]$) tracks with the weights ($Var[W]$) (and so to keep it from exploding in either direction). Specifically, setting $Var[W] = \frac{2}{n_{in}+n_{out}}$ is an attempt to compromise between the forward and backward requirements given in equations (11) and (12) of Xavier that $n_{in}Var[W] = 1$ and $n_{out}Var[W] = 1$. Figure 6 in Xavier demonstrates this by showing that using the proposed normalized initialization keeps the activation value distributions roughly constant throughout the network layers, as opposed to using standard initialization where the magnitude of the activation value distribution "explodes" (towards 0) with increasing layer depth.

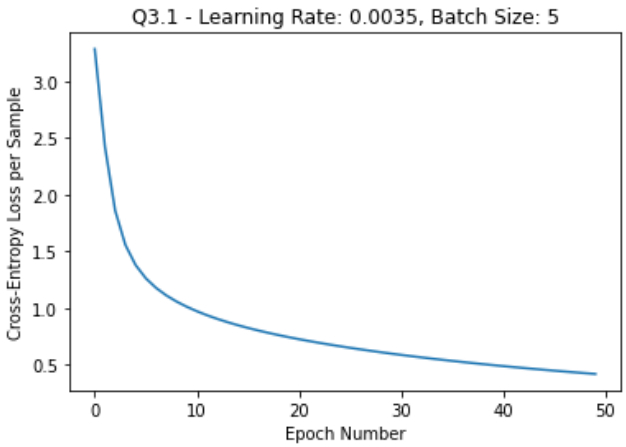
3 Training Models

Q3.1 Code

Results as requested are shown below. **Note:** Per the HW5 FAQ (@562) and @590 on Piazza, losses are shown as total loss divided by number of samples in the dataset.



(a) Accuracy of network across training epochs.



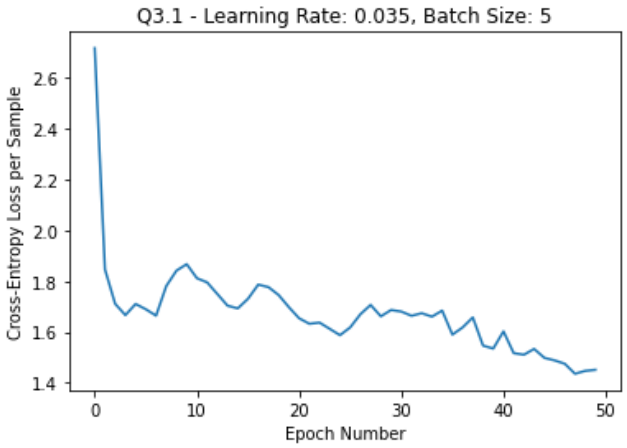
(b) Loss of network across training epochs.

Figure 4

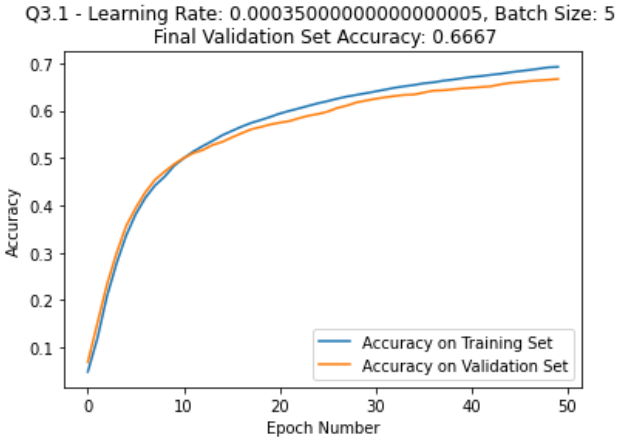
Accuracy of the tuned network (`batch_size=5`, `learning_rate=3.5e-3`) on the validation set: **77.06 %**

Q3.2 Writeup

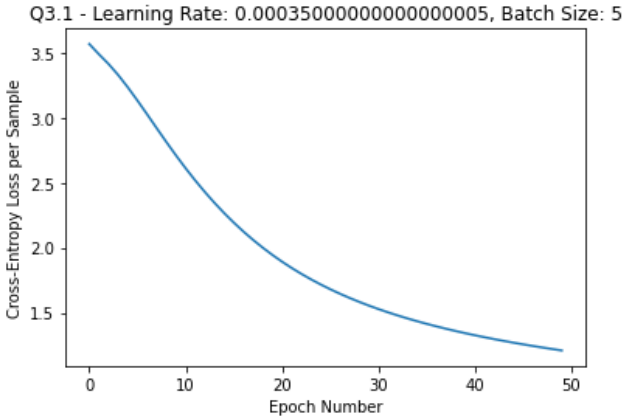
(a) Accuracy of network across training epochs for 10× best learning rate.



(b) Loss of network across training epochs for 10× best learning rate.



(c) Accuracy of network across training epochs for 0.1× best learning rate.



(d) Loss of network across training epochs for 0.1× best learning rate.

Figure 5

From comparing these plots to those generated for Q3.1 in Figure 4, it's evident that significantly increasing the learning rate above the ideal learning rate will cause a rapid decrease in initial loss followed by a long period of slow net reduction in loss characterized by excessive overshooting of the optimum, which eventually levels out at a much higher loss level than exhibited with an appropriate learning rate (as exhibited in Figure 5(b)). Similarly, as a result, the accuracy initially rises up sharply but ultimately varies quite rapidly and levels off at a low value (as exhibited in Figure 5(a)).

Likewise, from comparing these plots to those generated for Q3.1 in Figure 4, it's evident that significantly decreasing the learning rate below the ideal learning rate will cause much slower learning (as exhibited in Figure 5(d)). Also of note is that Figure 5(c) would seem to show that the validation accuracy more closely tracks the training accuracy in this case than in the ideal case; however, it should be dually noted that the accuracies shown in this plot are significantly lower than those in Figure 4(a) and if one compares the separation between the training and validation accuracies in Figure 4(a) at the accuracy levels shown in Figure 5(c), it would be seen that they both track similarly. Thus, it seems that in the case of a suboptimal learning rate, learning is mainly is just slower, requiring more epochs.

Final accuracy of the best network (`batch_size=5`, `learning_rate=3.5e-3`) on the testing set: **77.11 %**

Q3.3 Writeup

Figure 6 below shows the reshaped weight vectors as weight images for each neuron in layer 1 after training, whereas Figure 7 shows the same weight vectors before training. From these visualizations, it's evident that the training process tunes these weights from being essentially random noise to a state where the corresponding neuron in layer 1 activates in response to a particular salient low-level feature of the input image, namely it seems prominent edges, lines, and arcs that are common to the dataset (for example, there are a number that correspond to variations of vertical lines such as those found in a lot of alphanumerics like 'L', 'D', 'E', 'T', '1', '7', etc.)

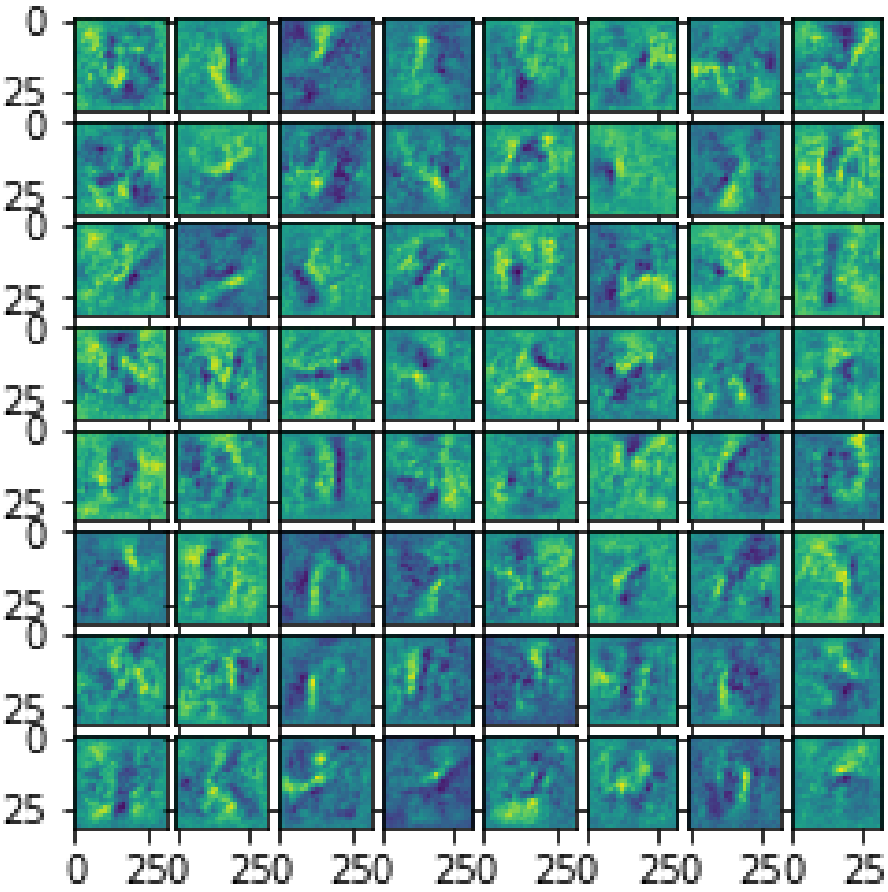


Figure 6: Reshaped weight vectors as weight images for each neuron in layer 1 after training.

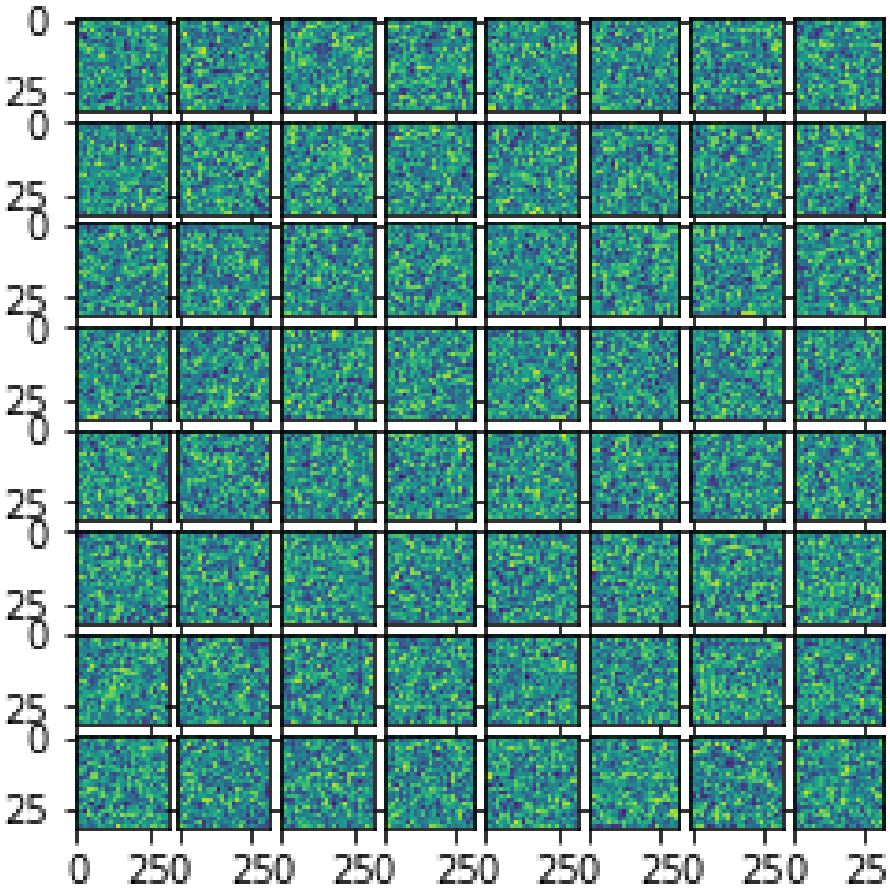


Figure 7: Reshaped weight vectors as weight images for each neuron in layer 1 before training.

Q3.4 Writeup

Figure 8 below shows the confusion matrix of the best model from above for the testing set (per `@563` on Piazza). According to this matrix, a few of the most common misclassifications are:

- misclassifying O as 0 (and to a lesser extent 0 as O).
- misclassifying S as 5 and 5 as S.
- misclassifying U as V (and to a slightly lesser extent V as U).

The reason behind all of these is likely that, since the dataset contains handwritten symbols, a number of them are ambiguous, for example having their sharp corners rounded or rounded corners truncated (very small radius, making them look sharp), which would blur the distinguishing feature between V and U and S and 5. Of note is the fact that the most common misclassification is O for 0. This is not surprising since, in many people's handwritting, they're effectively the same symbol, perhaps with 0 being narrower. What's salient is that the number of misclassifications of an O as a 0 is significantly higher than the number of 0s misclassified as Os. The latter number is likely lesser because some people write their 0s with an extra distinguishing dash through it, which makes it more recognizable and far less likely to be misclassified. No such mark exists for an O which would guarantee it to be an O, and so a larger proportion of Os will be considered 0s than 0s considered Os.

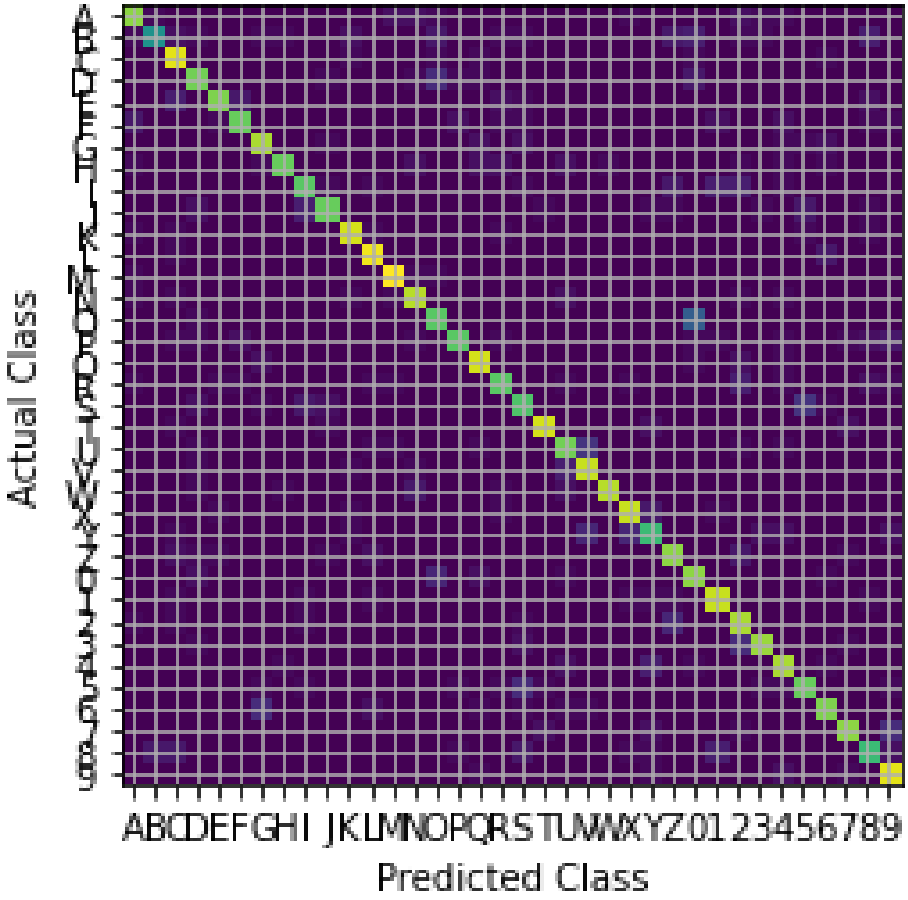


Figure 8: Confusion Matrix on Test Set

4 Extract Text from Images

Q4.1 Theory

Two big assumptions in the method outlined in the assignment are that:

1. No letters will be overlapping (or close enough that they will be made overlapping by image processing) and thus fed into the network as one character.
2. That characters will fall on a neatly identifiable line (and not some sort of curve where if you were to draw a "best-fit" line through one part of one line of text, it would intersect characters from another line of text).

The following two samples are examples of text which would break the two assumptions listed above.

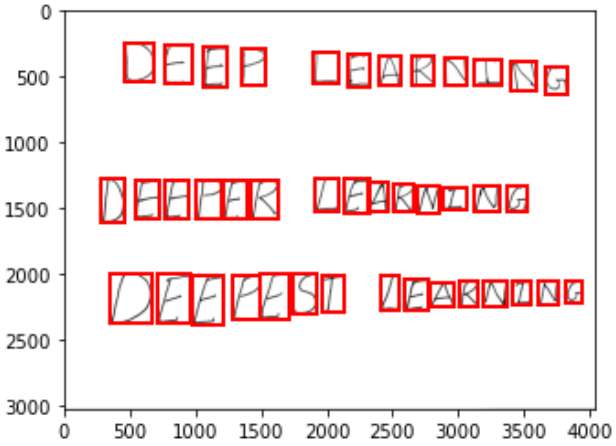
(a) Breaks assumption 1.

(b) Breaks assumption 2.

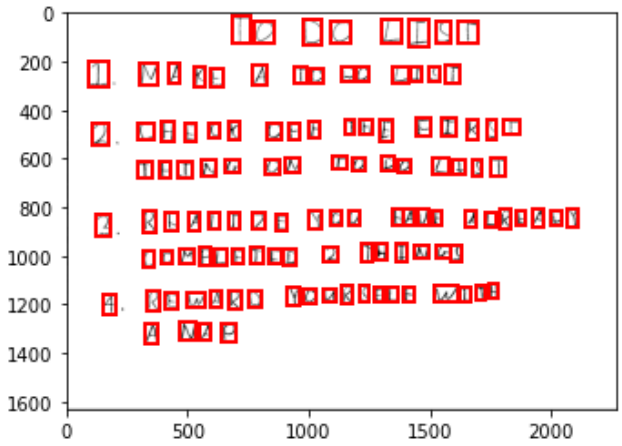
Figure 9

Q4.3 Writeup

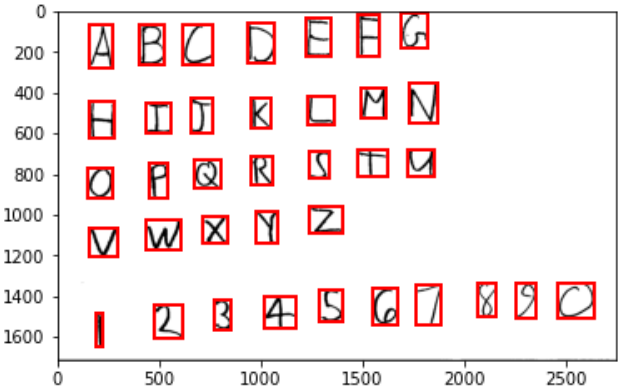
The following images display the result of running `findLetters(...)` on the supplied images. The images are the output binary `bw` image and the red boxes represent the bounding boxes of the detected letters.



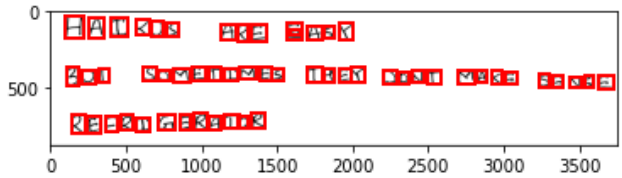
(a) Output for `04_deep.jpg`.



(b) Output for `01_list.jpg`.



(c) Output for `02_letters.jpg`.



(d) Output for `03_haiku.jpg`.

Figure 10

Q4.4 Code/Writeup

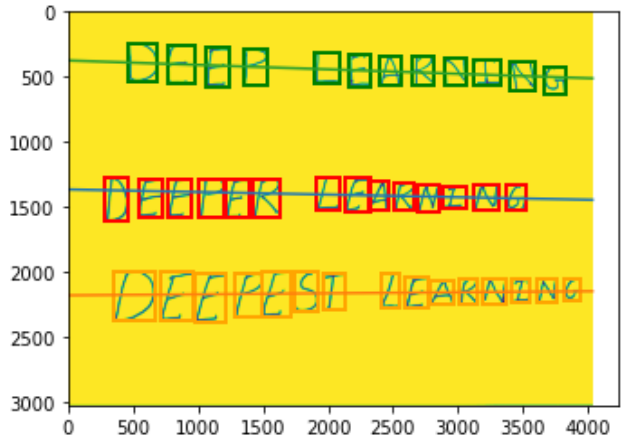
Line Finding Algorithm

To find the lines of text, I used an algorithm I refer to as NIRDE (N-Iteration RANSAC for Dominant Examples) in the code. The principle is to use RANSAC iteratively to find the line which includes the most letter box centroids as inliers, then removing the points which are inliers to the that line, and repeating n-times to find the n most prominent lines. The algorithm is as follows:

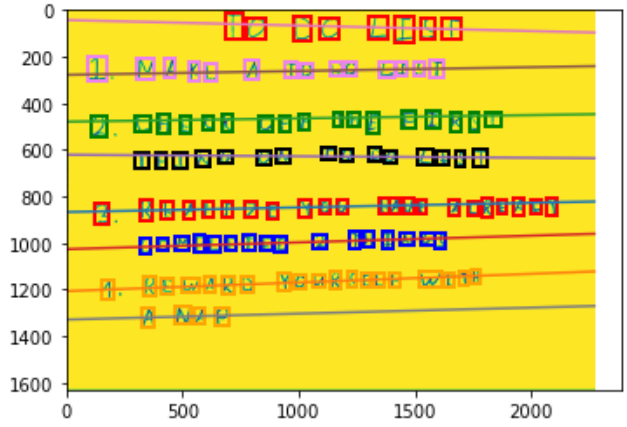
1. Run RANSAC to get a best line coefficients and indices of inliers.
2. Discard all points which are inliers to any previously solved models.
3. Repeat 1 and 2, n times or until all centroids have been used as an inlier to a line.
4. Return a matrix containing all RANSAC-ed line coefficient vectors and a boolean matrix indicating which box indices were inliers to each line found.

Figure 11 below shows the solutions NIRDE found for each line by drawing all dominant lines found on top of the images and then, for each line, drawing the bounding boxes of the letters whose centroids were inliers to the line with different colors for each line. If a letter were not found to be a inlier to any line, it would not show up in these plots. Fortunately, all the letters do. To reproduce this visualization, you can run `show_lines_on_image(im, Cs, inliers, bboxes)` after `NIRDE_lines(...)` in the code.

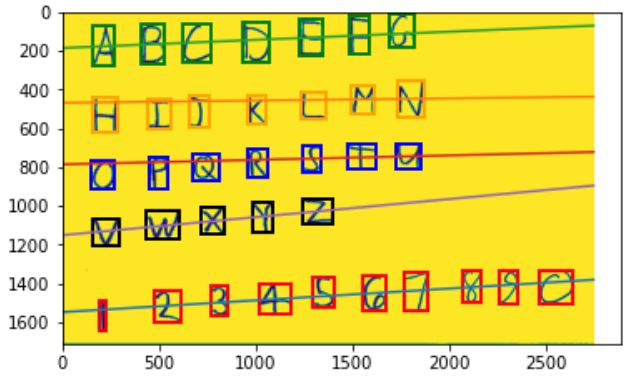
With the results of this algorithm, one which letters correspond to each line and must simply sort the lines by increasing y-intercept (top-down) to get the lines in the right order.



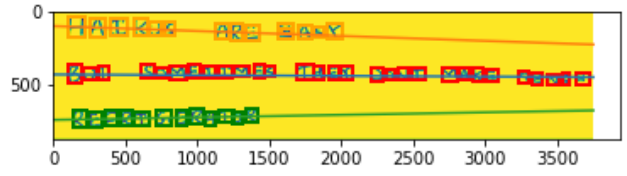
(a) Output for `04_deep.jpg`.



(b) Output for `01_list.jpg`.



(c) Output for `02_letters.jpg`.

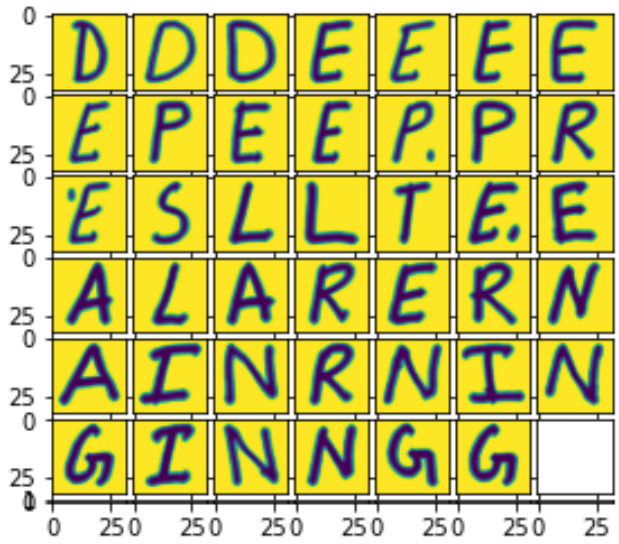


(d) Output for `03_haiku.jpg`.

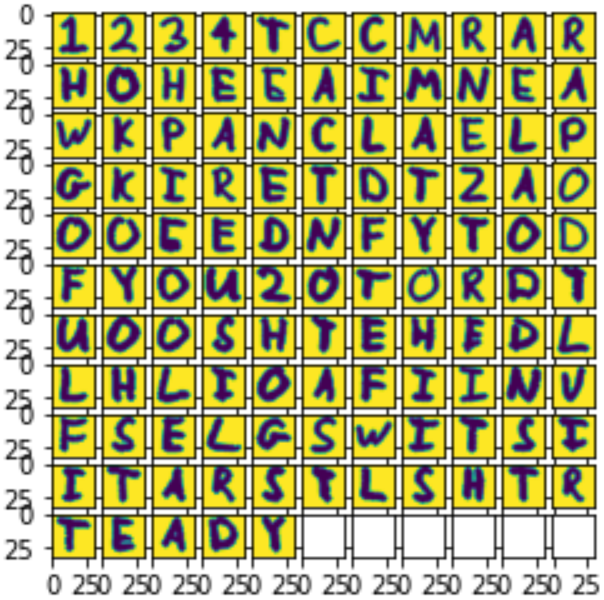
Figure 11

Letters Grids

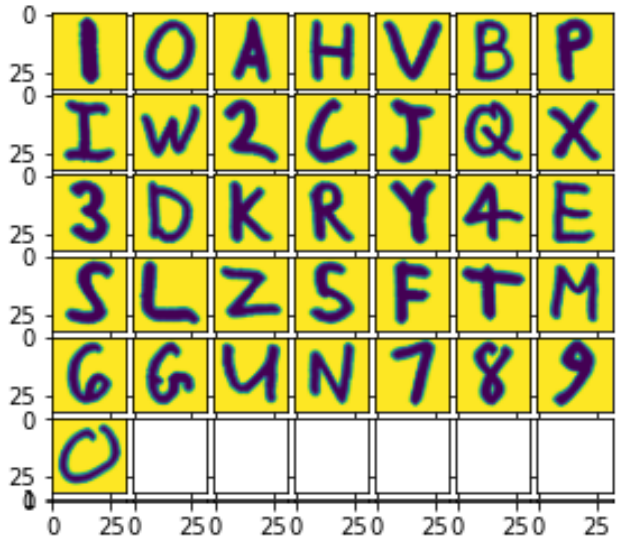
The following letters grids were created for internal evaluation purposes and show the post-processed letters that were fed into the neural network. Note: the ordering of the letters is not relevant in the grids (they are sorted by centroid x-position).



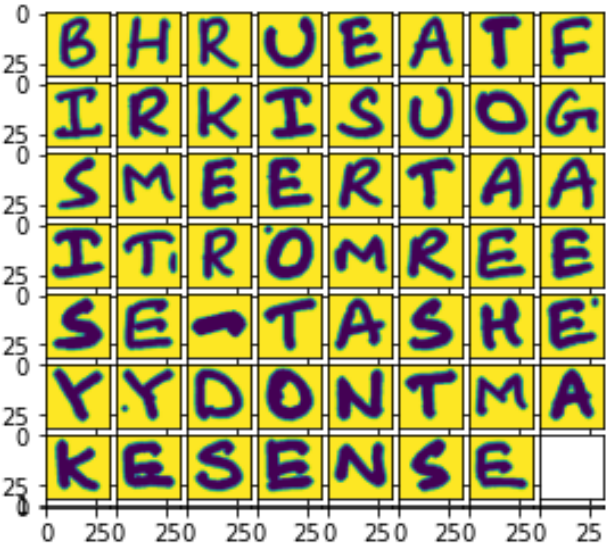
(a) Output for `04_deep.jpg`.



(b) Output for `01_list.jpg`.



(c) Output for `02_letters.jpg`.



(d) Output for `03_haiku.jpg`.

Figure 12

Output

Raw Output The following is the raw output from `run_q4.py`.

```
==== 04_deep.jpg Text: ====
DEEP LEARMING
DE8PER LEARNING
DEEREST LEARNING

==== 01_list.jpg Text: ====
PO DO LIST
I NA EE A TO QO LIS T
2 CH E C K OF F 7 HE F IRS T
YH ING QN TO DQ LIS T
3 RE AL I ZE YOU HAVE A LRE ADY
CO MPLET2D 2 YH INGS
4 RE WARD YOURSELF WIIR
A NA P

==== 02_letters.jpg Text: ====
2 B 6 D E F G
H II K L M N
0 P Q R S TU
V WX Y Z
1 2 3 4S 6 7 8 9 O

==== 03_haiku.jpg Text: ====
HAIKUS ARE EM AGY
BUI SQMETIMEG TAEY DDNT MAK6 SGNQ E
REFRIGERA7QR
```

Cleaned up Output The following is the same output with manually cleaned up spacing and punctuation to ease grading (as requested on Piazza).

```
==== 04_deep.jpg Text: ====
DEEP LEARMING
DE8PER LEARNING
DEEREST LEARNING

==== 01_list.jpg Text: ====
PO DO LIST
I. NAEE A TOQO LIST
2. CHECK OFF 7HE FIRST
   YHING QN TO DQ LIST
3. REALIZE YOU HAVE ALREADY
   COMPLET2D 2 YHINGS
4. REWARD YOURSELF WIIR
   A NAP

==== 02_letters.jpg Text: ====
2 B 6 D E F G
H I I K L M N
0 P Q R S T U
V W X Y Z
1 2 3 4 S 6 7 8 9 O

==== 03_haiku.jpg Text: ====
HAIKUS ARE EMAGY
BUI SQMETIMEG TAEY DDNT MAK6 SGNQE
REFRIGERA7QR
```

5 Image Compression with Autoencoders

Training the Autoencoder

Q5.2 Writeup/Code

Figure 13 below shows the training loss curve as training progresses. The loss decreased precipitously in the first few epochs, then almost leveled out until around epoch 65 when there was a brief and quick decrease in loss, after which point the loss remained nearly constant at 18.7. Essentially, it levels off fairly quickly at what's likely a nonideally high loss level. This high loss value is likely due to the fact that it's very difficult if not impossible to very closely reconstruct the original image when it can come from such a diverse set of classes as handwritten alphanumeric characters.

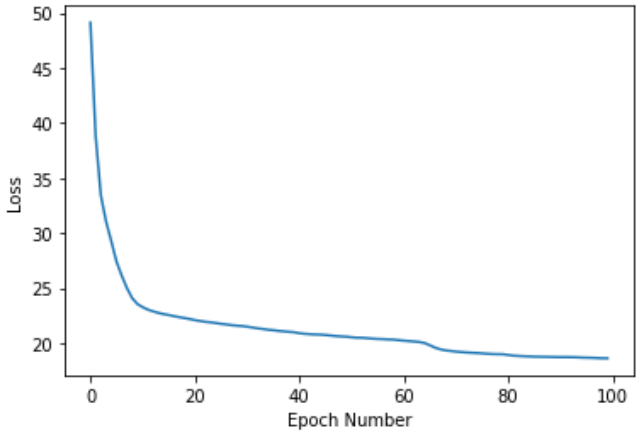


Figure 13: Network Loss over Training Epochs

Evaluating the Autoencoder

Q5.3.1 Writeup/Code

Figures 14-18 below show the comparisons between the original and reconstructed images for 5 classes. The constructed images seem to, broadly, have two classes of faults: 1. "ghostly" shadows of features not present in the original image, such as the arcs around "S" or the box around "T" and 2. smoothing over of the existing features (generalizing them) such as the arcs in the first "T" and first "W" and the rectilinearizing of the lines in "4" (they've become more straight and right angled). This is sensible considering that each of (relatively) small number of the neurons in the first two layers of autoencoder is tasked activating in response to general features — or ensembles of them — and the neurons in the latter layers are effectively responsible for learning to rebuild the image based which earlier neurons are active and which image features they typically activate in the presence of. As such, it makes sense that a wide variety of features such as box-like and arc-like features could activate the earlier neurons in the same way, making it intractable for the later neurons to precisely determine the root cause in the source image.

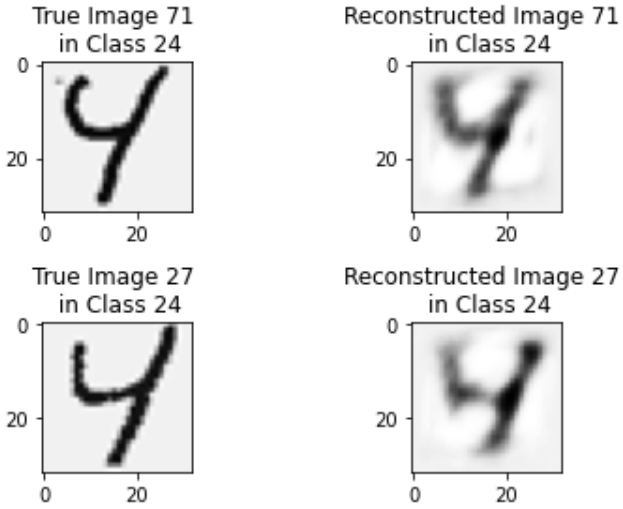


Figure 14

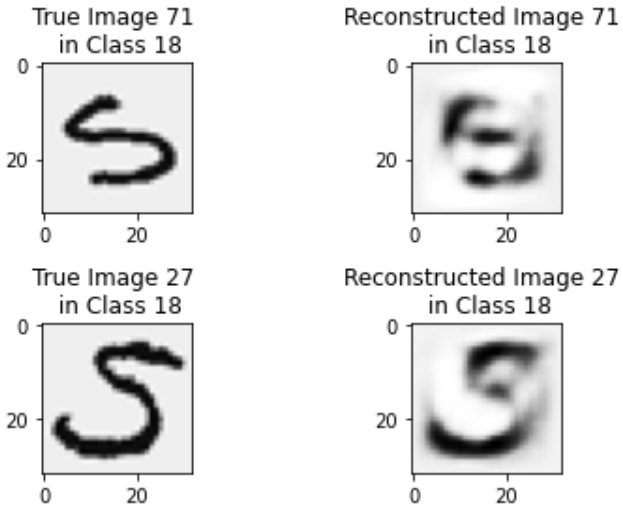


Figure 15

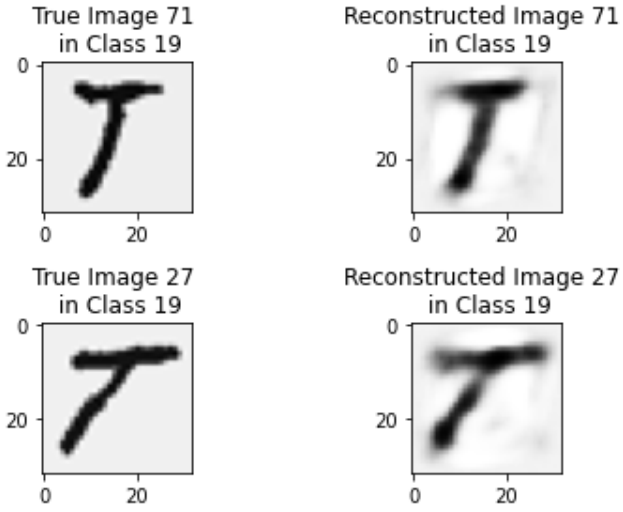


Figure 16

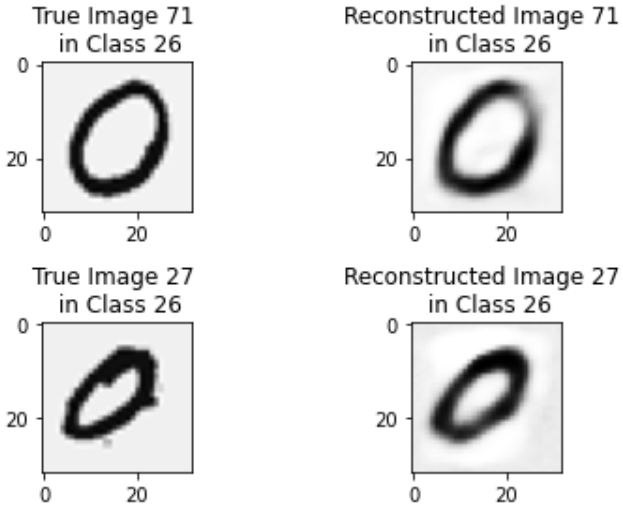


Figure 17

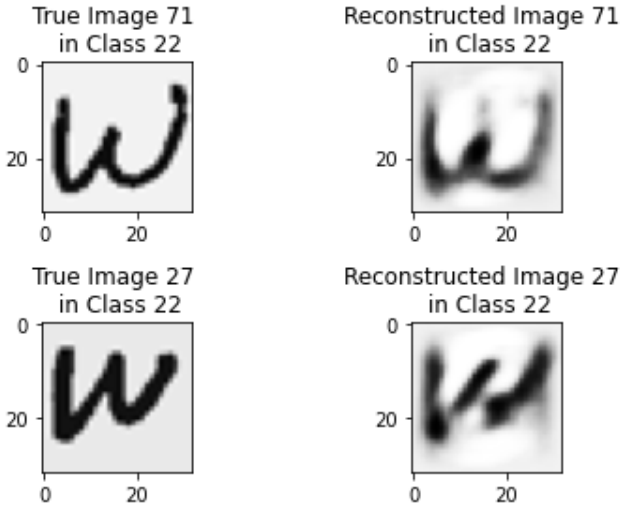


Figure 18

Q5.3.2 Writeup/Code

Average PSNR for Trained Autoencoder across Validation Set: `15.942`

6 Pytorch Extra Credit

6.1 Q6.1.1 Code/Writeup

Model trained much more slowly and didn't really converge then kernel crashed on final epoch... Must have set something up incorrectly. Terminal output below. Code in `run_q6_1.py`.

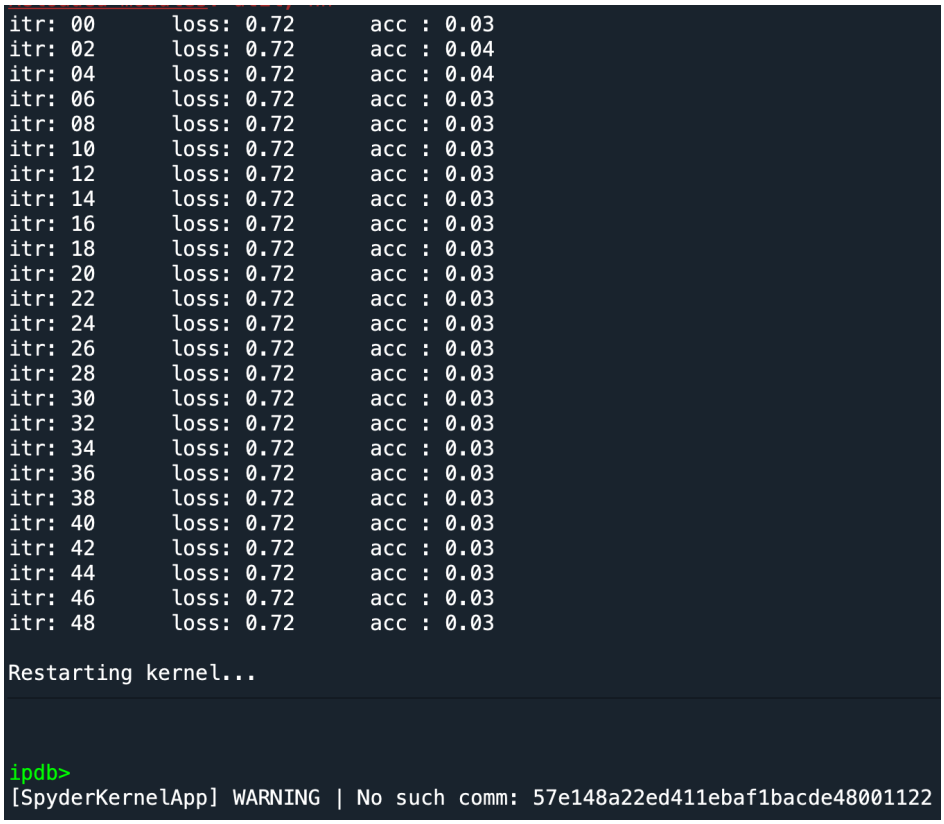


Figure 19