Andrew ID: haomingc

16720 – Computer Vision
Assignment 5

Question 1.1.1:

ReLU (Rectified Linear Unit) has the form of $g(a) = \max(0, a)$, which is simply truncating the negative-value part and replace it by a horizontal line. This mechanism guarantees its gradient to be a constant value (1 if a > 0) which leads to faster learning. However, for sigmoid function sig(a), we have the gradient of the form sig(a) * [1 - sig(a)] whose value becomes smaller and closer to 0 as the value of a increases, this is the so-called "vanishing gradient" problem. Moreover, ReLU has significantly greater chance of generating zero weights/values than sigmoid due to its rectified nature, which leads to a sparse representation that is beneficial to the training phase.

Question 1.1.2:

If the activation function of the hidden layers is linear, denoted by $g(\boldsymbol{a}^{(i)})$, then given the input **x**, weights **W** and biases **b** for every hidden layer, softmax function **o** for the output layer, we have the output f(**x**):

$$\begin{aligned} \mathrm{f}(\mathbf{x}) &= \boldsymbol{o}\left(\boldsymbol{a}^{(T)}\right) = \boldsymbol{o}\left(\boldsymbol{W}^{(T)} g\left(\boldsymbol{a}^{(T-1)}\right) + \boldsymbol{b}^{(T)}\right) \\ &= \boldsymbol{o}\left(\boldsymbol{W}^{(T)} g\left(\boldsymbol{W}^{(T-1)} g\left(\boldsymbol{a}^{(T-2)}\right) + \boldsymbol{b}^{(T-1)}\right) + \boldsymbol{b}^{(T)}\right) \\ &= \boldsymbol{o}\left(\boldsymbol{W}^{(T)} g\left(\boldsymbol{W}^{(T-1)} g\left(\boldsymbol{W}^{(i)} g\left(\ldots \boldsymbol{g}(\boldsymbol{W}^{(1)}\mathbf{x} + \boldsymbol{b}^{(1)})\right) + \boldsymbol{b}^{(i)}\right) + \boldsymbol{b}^{(T-1)}\right) + \boldsymbol{b}^{(T)}\right) \end{aligned}$$

Because the activation function is linear, then the first intermediate output of the first hidden layer is $\boldsymbol{h}^{(1)} = \boldsymbol{g}(\boldsymbol{W}^{(1)}\mathbf{x} + \boldsymbol{b}^{(1)})$, the second hidden output $\boldsymbol{h}^{(2)} = \boldsymbol{g}(\boldsymbol{W}^{(2)}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)})$ is also just a linear transformation of the first linear function, which can sum up to become a simple linear transformation instead of two. Likewise, for T layers, all T linear transformation can sum up to become just one or $T' \neq T$ transformations, thus the original neural network can be simply replaced by a $T' \neq T$ layer network or just a one-layer perceptron.

Question 2.1.1:

For all zeros initialization, the activations $\boldsymbol{a}^{(i)} = \boldsymbol{W}^{(i)}\boldsymbol{h}^{(i-1)}(\boldsymbol{x}) + \boldsymbol{b}^{(i)}$ are zeros, which means that the gradients for all weights are always going to be zeroes and prevent the weights

from changing to learn from the loss function. Thus, it's a bad idea to initialize a network with all zeros. For all ones or all some specific constant value, we face the problem of getting the exact same signal value $\boldsymbol{a}^{(i)}$ for every unit in a hidden layer and same updates for every units during training, this also prevents the neural network from learning the actual representation because all ones/constant value initialization only covers a tiny fraction of the hypothesis space. Therefore, it's also not a good idea to set all weights and biases to be the same value.

Question 2.1.3:

For weights, we apply a heuristic and initialize each weight vector as w = 0.01 * rand(n) / sqrt(n), where n is the number of its inputs (dimension of previous layer for a fully connected network). This heuristic produces a weight vector for each unit within the interval [0.0, 1.0] and normalize the variance of output to 1 by dividing the sqrt(n), then we multiply each weight by 0.01 to make them close to 0. Such mechanism ensures that all units in the network initially have approximately the same output distribution and empirically improves the rate of convergence. For biases, we simply initialize them to 0. It's mathematically correct to do so and the asymmetry breaking is associated with weights but not biases.

- reference: http://cs231n.github.io/neural-networks-2

Question 2.4.1:

Stochastic gradient descent (SGD) only use a random single training data sample to update the parameter in each iteration, thus it has the benefit of improving the model from the very first sample and thus faster to converge. Moreover, SGD is adaptive to update the model "online" while batch GD cannot do so as it must use all training data. However, SGD's nature of "stochastic" leads to the fact that the result is an approximation and the loss may not be minimized well as batch GD.

Batch gradient descent (batch GD) uses all the complete training set to compute the gradient. Thus, for a gigantic training set, this process can be very costly and extremely slow which leads to a longer converge time. However, batch GD also has its strength. It's using the whole dataset so the result is deterministic and guarantee to reach the same optimum with specific iterations.

SGD is faster in terms of iterations; batch GD is faster to train in terms of number of epochs (One pass through the entire dataset).

Question 3.1.2:

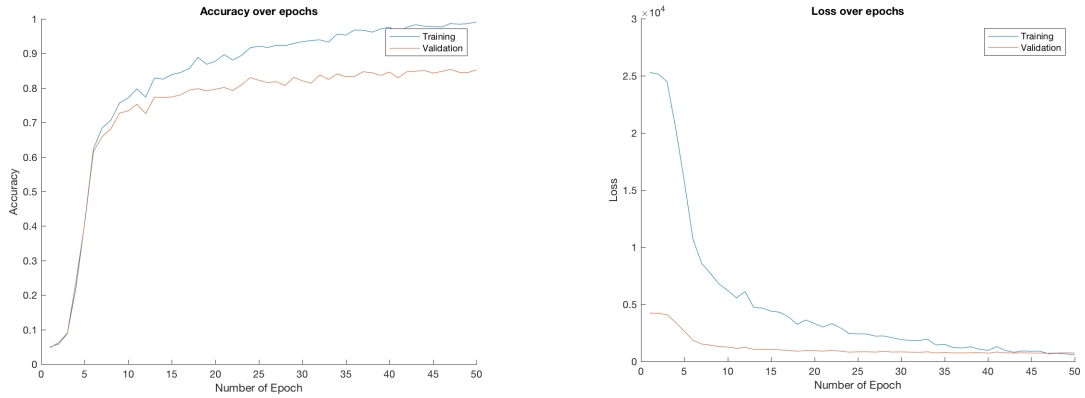


Figure 1. learning rate = 0.01

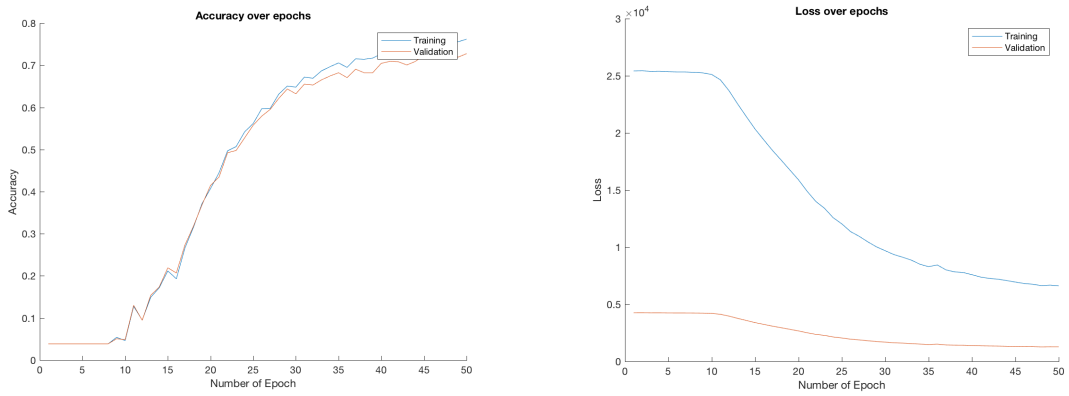


Figure 2. learning rate = 0.001

Learning rate is crucial in training a neural net. As we can see from the plots above, learning rate with 0.01 typically converges faster than that of 0.001, whose accuracy's improvement of the first 10 epochs are almost negligible. With the same number of epochs (50), learning rate 0.01 reaches 99% accuracy for training set and 85% for validation set. While learning rate 0.01 only has 76.22% accuracy for training set, which means it's still far away from the optimal minimum, let alone to mention its accuracy for validation (72.7%).

A relatively large learning rate means a bigger step in updating the weights, thus it requires less iterations to reach the optimal solution. On the contrary, a super small learning rate means

each iteration will only move towards the optimal solution with a tiny step, which leads to slower convergence. However, we should also keep in mind that the learning rate cannot be too large, otherwise it will skip the global minimum and jump back and forth around it but can never reach it.

Final accuracy of test set using learning rate=0.01 is 85.00%.

Question 3.1.3:

- Accuracy: 85.00%
- Cross Entropy Loss: 1372.20934498128

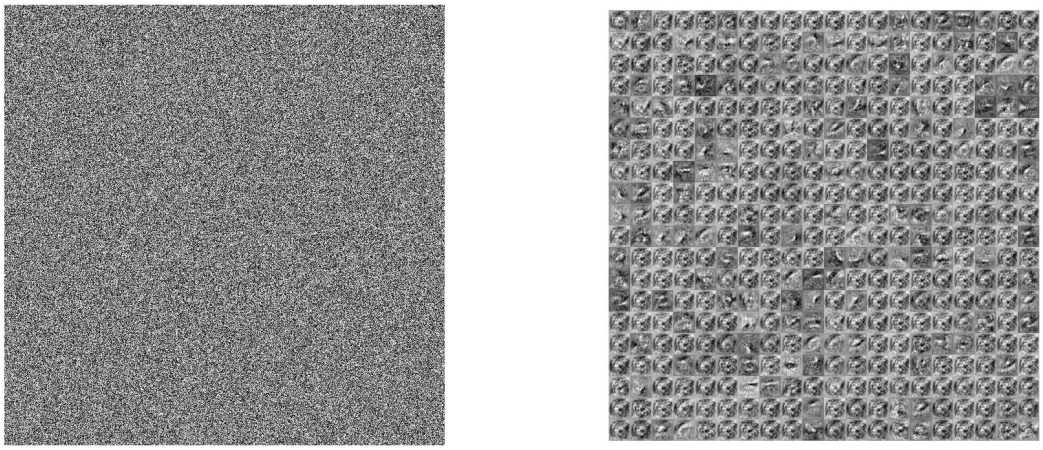


Figure 3. Initial weights (left) and Learned weights (right)

As shown above, the initial weights of the first layer are just some random "noise" which have no specific patterns, while the learned weights clearly present some rules and patterns that are high-level features to classify images. It seems that some weights of the first layer are representing some edge filters while others are similar to each other and may represent the somewhat similar features.

Question 3.1.4:

We generate the confusion matrix and scale up for a more explicit view. As shown in Figure 4, the performance is pretty good as our neural net classify most part of the data correctly. The

top two pairs that are most commonly confused are (“B”, “O”) and (“R”, “K”). This is reasonable as each pair of letters share similar structures like edges, curves, etc. Even human being may mistake them for each other sometimes, especially when we are dealing with hand-written letters and digits where sloppy writing often happens. For instance, the middle horizontal line may be fuzzy to make a “B” like “O”. The top curve of “R” is not learned well as a feature in the neural net or just bad-written to be easily mistaken to be “K”. Under which circumstance each pair of letters are extremely similar to each other.

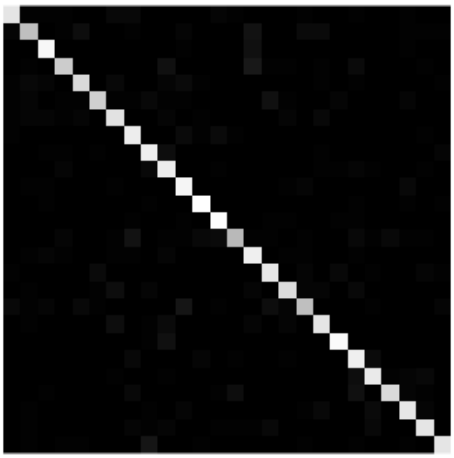


Figure 4. Confusion Matrix

Question 3.2.1:

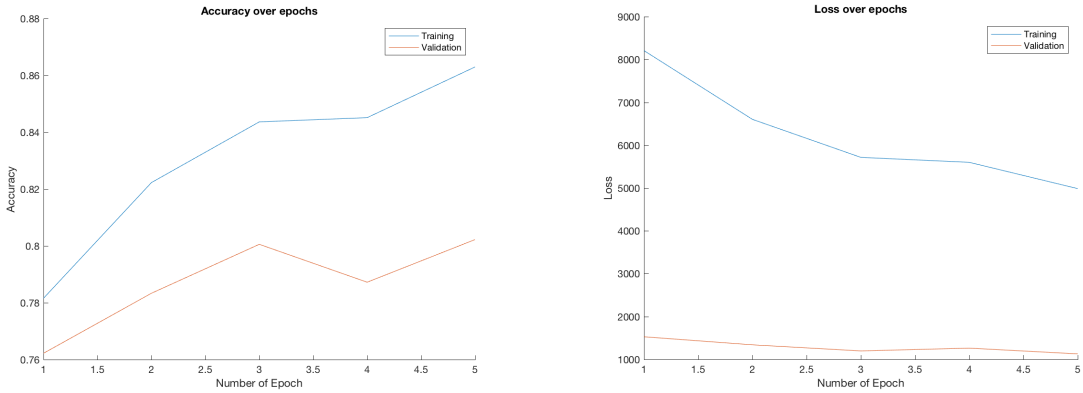


Figure 5. Fine-tuned network

Question 3.2.2:

- Accuracy: 81.08%
- Cross Entropy Loss: 2397.93607758436

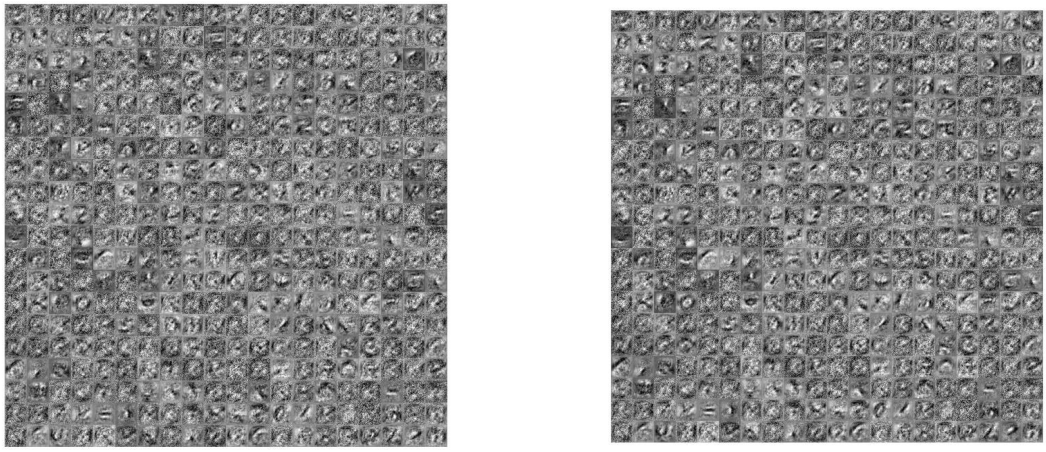


Figure 6. Initial weights (left) and Learned weights (right)

The difference between the initial weights and the learned weights are relatively small this time comparing to Q3.1.3. This is because the initial weights are already learned by a well-trained network to represent most of the classes (26 letters out of 36 classes). However, if we take a closer look, we can still spot some updates and difference in blocks as now the network should predict 36 classes, thus the weights will be updated and modified to be adaptive to represent the extra 10 digits.

Question 3.2.3:

As shown in Figure 7, the top two pairs that are most commonly confused are ("0", "O") and ("2", "Z"). For the digits "0", our neural net only recognizes 47% of them and classify 48% of them to letter "O". For digits "2", our neural net recognizes 59% of them and classify 19% of them to letter "Z". This is reasonable as each pair of letters/digits share similar structures like edges, curves, etc. Even human being may mistake them for each other sometimes, especially when we are dealing with hand-written letters and digits where sloppy writing often happens, under which circumstance each pair of letters/digits are extremely similar to each other. By introducing more classes to the network apparently lower its capability and accuracy to recognize hand-written objects as it now must distinguish an object from more classes. Moreover, some of the newly added classes of digits resemble the existed letters a lot as

demonstrated above, which also makes it harder for the network to achieve the former performance when number of classes are small.

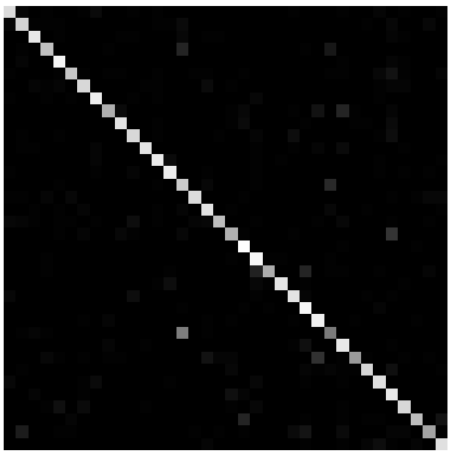
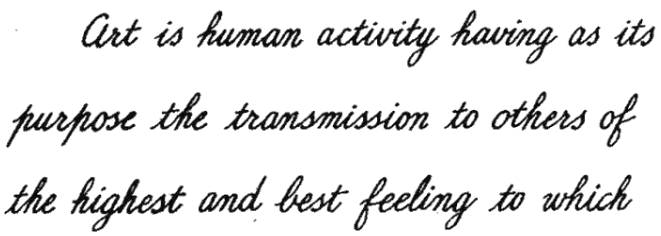


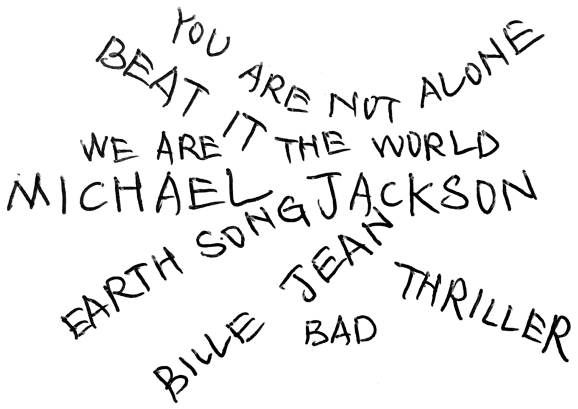
Figure 7. Confusion Matrix

Question 4.1:

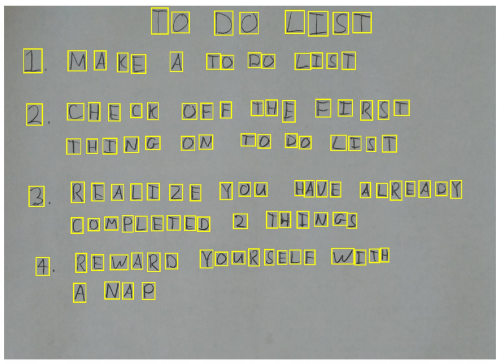
1. The method assumes that a character is a fully connected component which is not true in the real world. For instance, the letter 'G' is usually written in two parts (a half-left circle and the rest), or the middle horizontal line of letter 'E' is often separated from the vertical left-edge (such case also shows up in the sample image "03_haiku.jpg"). Moreover, hand-written letters are usually connected to their neighbors, so a connected component may consist of multiple letters.

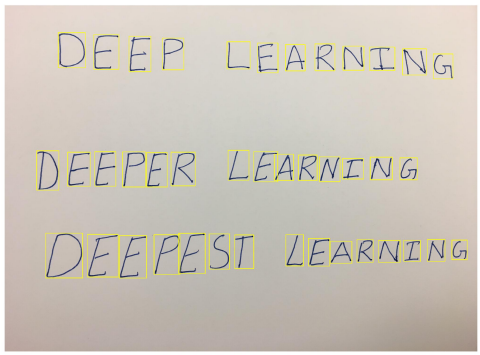


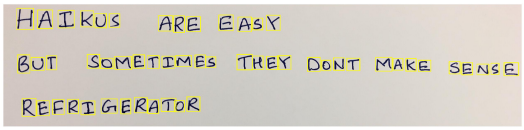
2. Another assumption is that the character is mostly parallel to the horizontal axis. If rotation with relatively large angle appears, then there's no proper "lines" in the image, which leads to an unexpected input for our neural net, and the extracted text will no longer be reasonable or interpretable for human being (e.g. a word cloud).

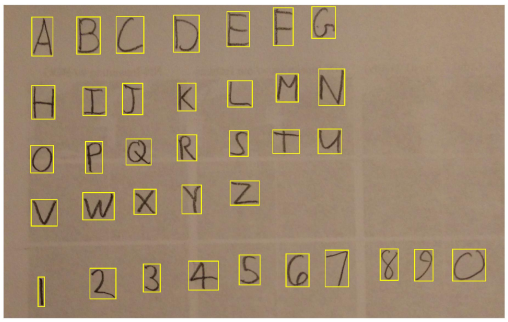


Question 4.3:









Question 4.5:

Space addition is included in the code to automatically insert space between words when necessary. The following shows the extracted texts. (Note that the result highly relies on the way we extract the character image and the initialized parameters of neural net.)

- 01_list.jpg

T0 U0 LI6T

I MAKE A T0 DQ LIST

2 CHGCK QFF THE FIRST

THING 0N T0 DQ LIST

3 RFALIZE Y1U HAUE ALREADY

C0MPLETED 2 THINGS

4 REWARD Y1URSELF WITH

A NAP

- 02_letters.jpg

  A B C U F F G

  M I I K L M N

  Q P Q K S T U

  V W X Y Z

  1 Z 4 G S G J X Y 6

- 03_haiku.jpg

  HAIWU6 ARE GASY

  BUT SQMETIMES THEY D0WT MAKG SENGG

  REGQIGERATOR

- 04_deep.jpg

  CFFP LCARMING

  DFFKFR LEAKNING

  6FFPF6T LEARNING