

CMU Fall24 16820 Homework 4

Patrick Chen

October 27, 2024

Q1.1 at page 3

Ans:

We have the following equation:

$$\text{softmax}(\mathbf{x} + c) = \frac{e^{x_i+c}}{\sum_j e^{x_j+c}} \quad (1)$$

$$= \frac{e^c e^{x_i}}{e^c \sum_j e^{x_j}} \quad (2)$$

$$= \frac{e^{x_i}}{\sum_j e^{x_j}} \quad (3)$$

$$= \text{softmax}(\mathbf{x}) \quad (4)$$

The idea to use $c = -\max_{x_i}$ is beneficial because we can have more numerical stability as the value of $e^{x_i - \max_{x_i}}$ in softmax would be in the range from 0 to 1, avoiding the risk of overflow.

Q1.2 at page 3

Ans:

The range of each element, $\frac{e^{x_i}}{\sum e^{x_i}}$ is from 0 to 1, the sum over all elements would be 1, because $\sum \frac{e^{x_i}}{\sum e^{x_i}} = \sum \frac{e^{x_i}}{e^{x_1} + e^{x_2} + \dots + e^{x_n}} = 1$.

One could say that “softmax takes an arbitrary real-valued vector \mathbf{x} and turns it into a probability distribution”.

The first step: $s_i = e^{x_i}$ transforms x_i to a positive value, representing the frequency of x_i . The second step: $S = \sum s_i$ can be deemed as a normalization to ensure the sum of output probability equals to 1, and it can also be deemed as the total frequency of each dimension x_i in \mathbf{x} . The third step: $\text{softmax}(x_i) = \frac{1}{S} s_i$ calculates the ratio of frequency of each element x_i among all the dimensions in \mathbf{x} , indicating the probability, where each element is between 0 and 1, and the summation across all of each element is 1.

Q1.3 at page 3

Ans:

When using multi-layer neural networks without a non-linear activation function, we would have (Assuming n layers fully connected neural network):

$$y = W_n x_n + b_n \quad (5)$$

$$= W_n(W_{n-1}x_{n-1} + b_{n-1}) + b_n \quad (6)$$

$$= W_n W_{n-1}x_{n-1} + W_n b_{n-1} + b_n \quad (7)$$

$$= W_n W_{n-1}(W_{n-2}(\dots(W_1 x_1 + b_1))) + W_n b_{n-1} + b_n \quad (8)$$

$$= W' x' + b', \text{ where } W' = W_n W_{n-1} \dots W_1, x' = x_1, b' = W_n W_{n-1} \dots W_2 b_1 + \dots + W_n b_{n-1} + b_n \quad (9)$$

As we can see in equation (9) above, it is still a linear regression.

Q1.4 at page 3

Ans:

$$\frac{d\sigma(x)}{dx} = \frac{d}{dx} \frac{1}{1+e^{-x}} \quad (10)$$

$$= \frac{e^{-x}}{(1+e^{-x})^2} \quad (11)$$

$$= \frac{e^{-x}}{(1+e^{-x})(1+e^{-x})} \quad (12)$$

$$= \frac{e^{-x}}{(1+e^{-x})} \frac{1}{(1+e^{-x})} \quad (13)$$

$$= (1 - \sigma(x))\sigma(x) \quad (14)$$

Q1.5 at page 3

Ans:

$$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial y} \frac{\partial y}{\partial W} \quad (15)$$

$$= \delta x^T \in \mathbb{R}^{k \times d} \quad (16)$$

$$\frac{\partial J}{\partial x} = \frac{\partial J}{\partial y} \frac{\partial y}{\partial x} \quad (17)$$

$$= W^T \delta \in \mathbb{R}^{d \times 1} \quad (18)$$

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y} \frac{\partial y}{\partial b} \quad (19)$$

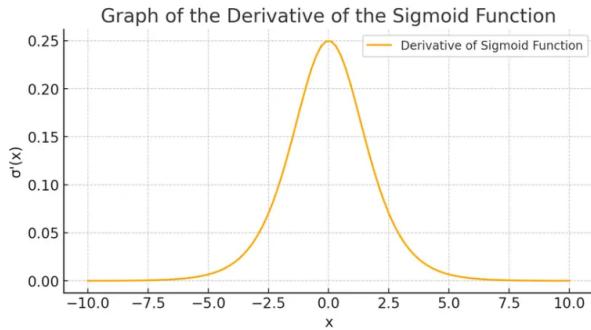
$$= \delta \circ \mathbf{1}_k, \text{ where } \circ \text{ is the element-wised multiplication, and } \mathbf{1}_k \text{ is a kx1 vector with all elements set to 1.} \quad (20)$$

$$= \delta \in \mathbb{R}^{k \times 1} \quad (21)$$

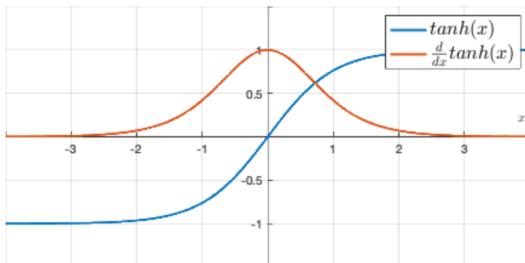
Q1.6 at page 3

Ans:

- As we can see from the below plot of the derivative of sigmoid function, the value of $(1 - \sigma(x))\sigma'(x)$ is in range [0, 0.25]. If we have multiple layers neural network applying sigmoid activation function, then during the backpropagation, we would need to multiply $(1 - \sigma(x))\sigma'(x)$ many times from the output layer to the input layer to get the gradient of weight, and the maximum of the value would be 0.25^n , assuming we have n hidden layers. This value would be very small, nearly zero, resulting in the vanishing of gradient.



- The output range of $\tanh(x)$ is $[-1, 1]$, and the output range of sigmoid function is $[0, 1]$. We prefer \tanh because it has wider range around zero, and it can map input x to either positive or negative values, reducing the bias introduced by sigmoid only outputting positive activation values.
- As we can see from the below figure, the $\tanh'(x)$ has value nearly 1 for inputs near zero, which is much more larger than the value of 0.25 of $\sigma'(x)$ (from above figure in problem 1) when inputs near zero. This means that $\tanh(x)$ has less of a vanishing gradient problem.



- The derivation of $\tanh(x)$ in terms of $\sigma(x)$ is shown below:

$$\tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \quad (22)$$

$$= \frac{1}{1 + e^{-2x}} + \left(-\frac{e^{-2x}}{1 + e^{-2x}} \right) \quad (23)$$

$$= \sigma(2x) - (1 - \sigma(2x)) \quad (24)$$

$$= 2\sigma(2x) - 1 \quad (25)$$

Q2.1.1 at page 4

Ans:

When weights are initialized to zero, then in forward propagation, all the output of layers would have the same zero value. Then during the back propagation, all the weights would have same gradients, and it means that each neuron in each layer is trying to learn the same feature, failing to capture the variation and complexity in each dimension of data. The result would be the model fails to converge to a effective solution.

Q2.1.2 at page 4

Ans:

The following Figure 1 shows the code snippet of initialize_weights() in the nn.py:

```
39 ~ def xavier_uniform(in_size,out_size):
40     #np.random.seed(42) #for test
41     a = np.sqrt(6/(in_size+out_size))
42     return np.random.uniform(-a, a, (in_size, out_size))
43
44 ~ def initialize_weights(in_size, out_size, params, name=""):
45     W, b = None, None
46
47     ##### your code here #####
48
49     W = xavier_uniform(in_size, out_size)
50     b = np.zeros(out_size)
51
52     params["W" + name] = W
53     params["b" + name] = b
54
55
```

Figure 1: Code Snippet

Q2.1.3 at page 4

Ans:

Initializing the weight with random numbers helps the neural network learn different features and pattern in data and facilitate the searching for an optimal point that minimize the loss. Scaling the initialization depending on the layer size could help the output of forward and backward propagation of each layer have the same variance as the input, helping stabilizing the output values and avoid gradient vanishing and explosion.

Q2.2.1 at page 4

Ans:

The following Figure 2 shows the code snippet of sigmoid() and forward() in the nn.py:

```
57 ##### Q 2.2.1 #####
58 # x is a matrix
59 # a sigmoid activation function
60 def sigmoid(x):
61     res = None
62
63     ##### your code here #####
64     #####
65
66     res = 1/(1+np.exp(-x))
67
68     return res
69
70
71 ##### Q 2.2.1 #####
72 def forward(X, params, name="", activation=sigmoid):
73     """
74     Do a forward pass
75
76     Keyword arguments:
77     X -- input vector [Examples x D]
78     params -- a dictionary containing parameters
79     name -- name of the layer
80     activation -- the activation function (default is sigmoid)
81     """
82     pre_act, post_act = None, None
83     # get the layer parameters
84     W = params["W" + name]
85     b = params["b" + name]
86
87     ##### your code here #####
88     #####
89
90     pre_act = X@W + b
91     post_act = activation(pre_act)
92
93     # store the pre-activation and post-activation values
94     # these will be important in backprop
95     params["cache_" + name] = (X, pre_act, post_act)
96
97     return post_act
98
```

Figure 2: Code Snippet of sigmoid() and forward()

Q2.2.2 at page 4

Ans:

The following Figure 3 shows the code snippet of softmax() in the nn.py:

```
100 ##### Q 2.2.2 #####
101 # x is [examples,classes]
102 # softmax should be done for each row
103 def softmax(x):
104     res = None
105
106     #####
107     #### your code here #####
108     #####
109     x_prone = x - np.max(x, axis=-1).reshape(-1, 1)
110     exp_x = np.exp(x_prone)
111     res = exp_x/np.sum(exp_x, axis=-1).reshape(-1, 1)
112
113
```

Figure 3: Code Snippet of softmax()

Q2.2.3 at page 4

Ans:

The following Figure 4 shows the code snippet of compute_loss_and_acc() in the nn.py:

```
115 ##### Q 2.2.3 #####
116 # compute total loss and accuracy
117 # y is size [examples,classes]
118 # probs is size [examples,classes]
119 def compute_loss_and_acc(y, probs):
120     loss, acc = None, None
121
122     ##### your code here #####
123
124
125     loss = -np.sum(np.log(probs[np.arange(probs.shape[0]), np.argmax(y, axis=1)]))
126     predictions = np.argmax(probs, axis=-1)
127     true_classes = np.argmax(y, axis=-1)
128     acc = np.mean(predictions == true_classes)
129
130     return loss, acc
131
```

Figure 4: Code Snippet of compute_loss_and_acc()

Q2.3 at page 5

Ans:

The following Figure 5 shows the code snippet of backwards() in the nn.py:

```
133 ##### Q 2.3 #####
134 # we give this to you
135 # because you proved it
136 # it's a function of post_act
137 def sigmoid_deriv(post_act):
138     res = post_act * (1.0 - post_act)
139     return res
140
141
142 def backwards(delta, params, name="", activation_deriv=sigmoid_deriv):
143     """
144     Do a backwards pass
145
146     Keyword arguments:
147     delta -- errors to backprop
148     params -- a dictionary containing parameters
149     name -- name of the layer
150     activation_deriv -- the derivative of the activation_func
151     """
152     grad_X, grad_W, grad_b = None, None, None
153     # everything you may need for this layer
154     W = params["W" + name]
155     b = params["b" + name]
156     X, pre_act, post_act = params["cache_" + name]
157
158     # do the derivative through activation first
159     # (don't forget activation_deriv is a function of post_act)
160     # then compute the derivative W, b, and X
161     #### your code here #####
162     #####
163     grad_Z = delta * activation_deriv(post_act)
164     grad_X = grad_Z @ W.T
165     grad_W = X.T @ grad_Z
166     grad_b = np.sum(grad_Z, axis=0)
167
168     # store the gradients
169     params["grad_W" + name] = grad_W
170     params["grad_b" + name] = grad_b
171
172     return grad_X
173
174
```

Figure 5: Code Snippet of backwards()

Q2.4 at page 5

Ans:

The following Figure 6 and Figure 7 shows the code snippet of get_random_batch() in nn.py and training loop procedure in run_q2.py:

```

175 ##### Q 2.4 #####
176 # split x and y into random batches
177 # return a list of [(batch1_x,batch1_y)...]
178 def shuffle(X, y):
179     """
180     :param X: The original input data in the order of the file.
181     :param y: The original labels in the order of the file.
182     :param epoch: The epoch number (0-indexed).
183     :return: Permuted X and y training data for the epoch.
184     """
185     N = len(y)
186     ordering = np.random.permutation(N)
187     return X[ordering], y[ordering]
188
189 def get_random_batches(x, y, batch_size):
190     batches = []
191     ##### your code here #####
192     ##### your code here #####
193     x, y = shuffle(x, y)
194     for start_idx in range(0, len(y), batch_size):
195         batch_x = x[start_idx:start_idx+batch_size, :]
196         batch_y = y[start_idx:start_idx+batch_size, :]
197         batches.append((batch_x, batch_y))
198
199     return batches

```

Figure 6: Code Snippet of Random Batch

```

83 ##### Q 2.4 #####
84 batches = get_random_batches(x, y, 5)
85 # print batch sizes
86 print([_[0].shape[0] for _ in batches])
87 batch_num = len(batches)
88
89 # WRITE A TRAINING LOOP HERE
90 max_iters = 500
91 learning_rate = 1e-3
92 # with default settings, you should get loss < 35 and accuracy > 75%
93 for itr in range(max_iters):
94     total_loss = 0
95     avg_acc = 0
96     for xb, yb in batches:
97         ##### your code here #####
98         ##### your code here #####
99         ##### your code here #####
100         # forward
101         h1_out = forward(xb, params, 'layer1', sigmoid)
102         y_hat = forward(h1_out, params, 'output', softmax)
103
104         # loss
105         # be sure to add loss and accuracy to epoch totals
106         loss, acc = compute_loss_and_acc(yb, y_hat)
107         total_loss += loss
108         avg_acc += acc
109
110         # backward
111         grad_y_hat = y_hat - yb
112         delta2 = backwards(grad_y_hat, params, "output", linear_deriv)
113         _ = backwards(delta2, params, "layer1", sigmoid_deriv)
114
115         # apply gradient
116         # gradients should be summed over batch samples
117         params['Wlayer1'] -= learning_rate*params['grad_Wlayer1']
118         params['blayer1'] -= learning_rate*params['grad_blayer1']
119         params['Woutput'] -= learning_rate*params['grad_Woutput']
120         params['boutput'] -= learning_rate*params['grad_boutput']
121
122         avg_acc /= batch_num
123
124     if itr % 100 == 0:
125         print(
126             "itr: {:02d} \t loss: {:.2f} \t acc : {:.2f}".format(
127                 itr, total_loss, avg_acc
128             )
129         )
130
131

```

Figure 7: Code Snippet of Training Loop

Q2.5 at page 5

Ans:

The following Figure 8 shows the code snippet of finite difference in run_q2.py:

```
133 # Q 2.5 should be implemented in this file
134 # you can do this before or after training the network.
135
136 def runForwardNN(x, y, params):
137     h1 = forward(x, params, "layer1")
138     probs = forward(h1, params, "output", softmax)
139     loss, acc = compute_loss_and_acc(y, probs)
140
141     return loss, acc
142
143 # compute gradients using forward and backward
144 h1 = forward(x, params, "layer1")
145 probs = forward(h1, params, "output", softmax)
146 loss, acc = compute_loss_and_acc(y, probs)
147 delta1 = probs - y
148 delta2 = backwards(delta1, params, "output", linear_deriv)
149 backwards(delta2, params, "layer1", sigmoid_deriv)
150
151 # save the old params
152 import copy
153
154 params_orig = copy.deepcopy(params)
155
156 # compute gradients using finite difference
157 eps = 1e-6
158 for k, v in params.items():
159     if "_" in k:
160         continue
161     # for each value inside the parameter
162     # add epsilon
163     # run the network
164     # get the loss
165     # subtract 2*epsilon
166     # run the network
167     # get the loss
168     # restore the original parameter value
169     # compute derivative with central diffs
170     if 'W' in k:
171         for i in range(v.shape[0]):
172             for j in range(v.shape[1]):
173                 orig_v = v[i, j]
174                 v[i, j] += eps
175                 loss_pos, acc_pos = runForwardNN(x, y, params)
176                 v[i, j] -= 2*eps
177                 loss_neg, acc_neg = runForwardNN(x, y, params)
178                 params['grad_'+k][i, j] = (loss_pos - loss_neg)/(2*eps)
179                 v[i, j] = orig_v
180
181     if 'b' in k:
182         for i in range(v.shape[0]):
183             orig_v = v[i]
184             v[i] += eps
185             loss_pos, acc_pos = runForwardNN(x, y, params)
186             v[i] -= 2*eps
187             loss_neg, acc_neg = runForwardNN(x, y, params)
188             params['grad_'+k][i] = (loss_pos - loss_neg)/(2*eps)
189             v[i] = orig_v
190
191 ##### your code here #####
192 #####
193 #####
```

Figure 8: Code Snippet of Finite Difference Gradient Check

Q3.1 at page 6

Ans:

The following Figure 9 and Figure 10 shows the result after 100 epochs training with batch size set to 64, and learning rate set to $2e^{-3}$. The validation accuracy is 77%.'

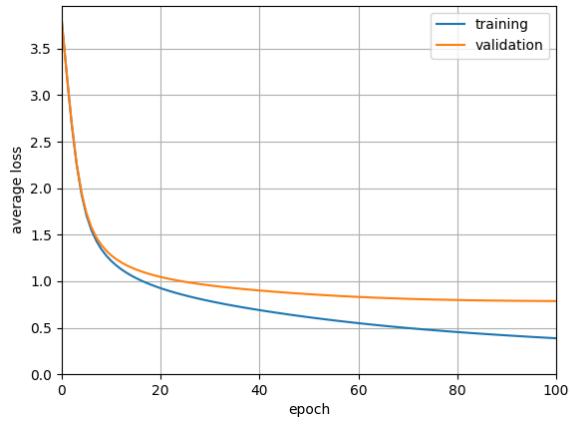


Figure 9: Average Loss vs Epoch

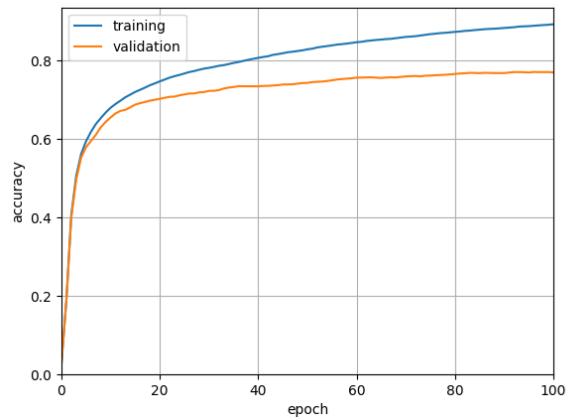


Figure 10: Accuracy vs Epoch

Q3.2 at page 6

Ans:

All of the following results are generated with batch size = 64.

The following Figure 11 and Figure 12 shows the result of learning rate set to the ten times of tuned one in Q3.1, which is $2e^{-2}$. The test accuracy is 72.22%, lower than the case of tuned learning rate in Q3.1. The plots show sawtooth-like pattern, indicating the learning rate is set too high, making the gradient update and the loss change drastic.

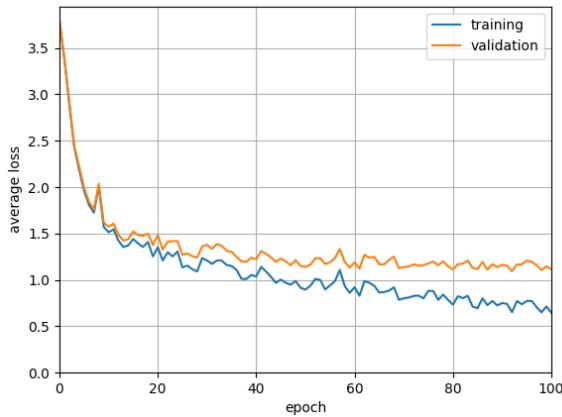


Figure 11: Average Loss vs Epoch

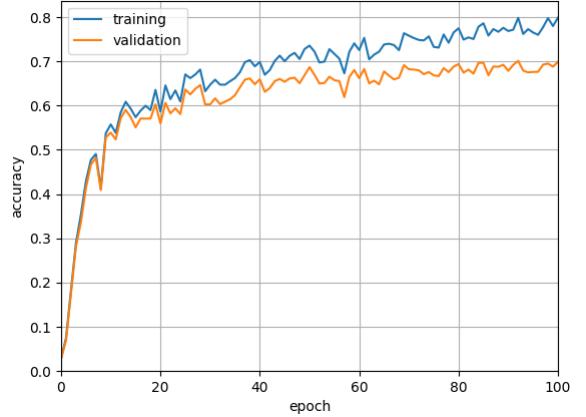


Figure 12: Accuracy vs Epoch

The following Figure 13 and Figure 14 show the result of the tuned learning rate set to the same as Q3.1, which is $2e^{-3}$. The result is the best, having test accuracy 77.83%.

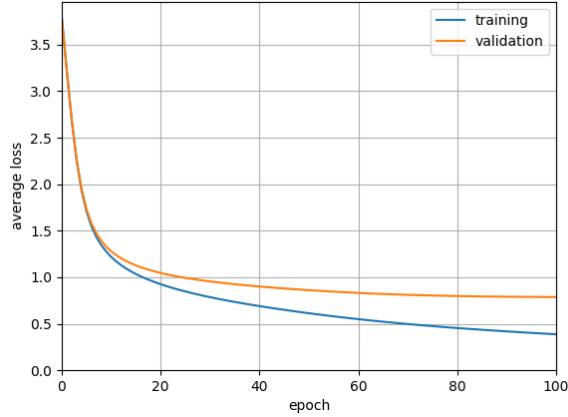


Figure 13: Average Loss vs Epoch

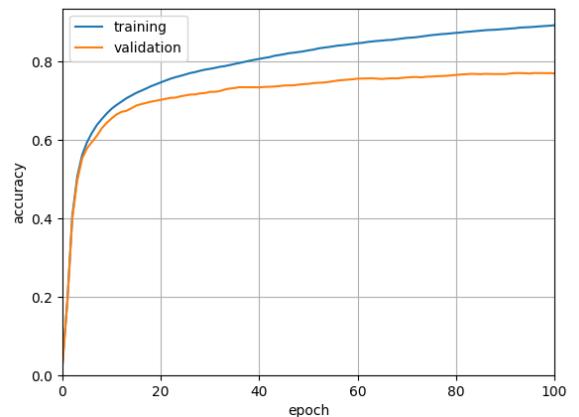


Figure 14: Accuracy vs Epoch

The following Figure 15 and Figure 16 show the result of learning rate set to one tenth of the one set in Q3.1, which is $2e^{-4}$. The test accuracy is 69%, lower than the tuned case in Q3.1. The plots show that the model has not yet reached the convergence point because of the small learning rate. There is still room for improvement, and we need more epochs to optimize this model.

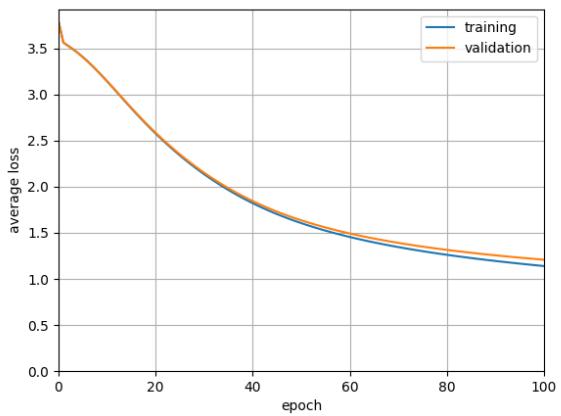


Figure 15: Average Loss vs Epoch

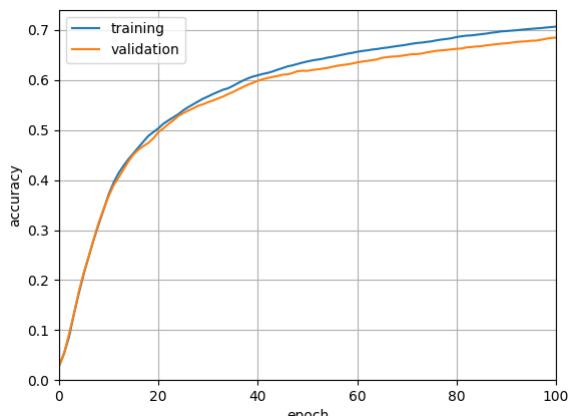


Figure 16: Accuracy vs Epoch

Q3.3 at page 6

Ans:

The following Figure 17 shows the initialized weight that looks like noise, which is expected because we initialized the weighted as random uniform distribution. The Figure 18 shows the optimized weight after 100 epochs, demonstrating more clear patterns.

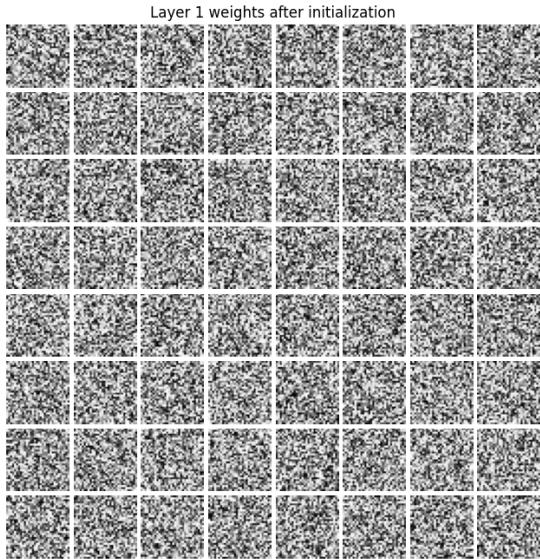


Figure 17: Layer1 Weights Initialization

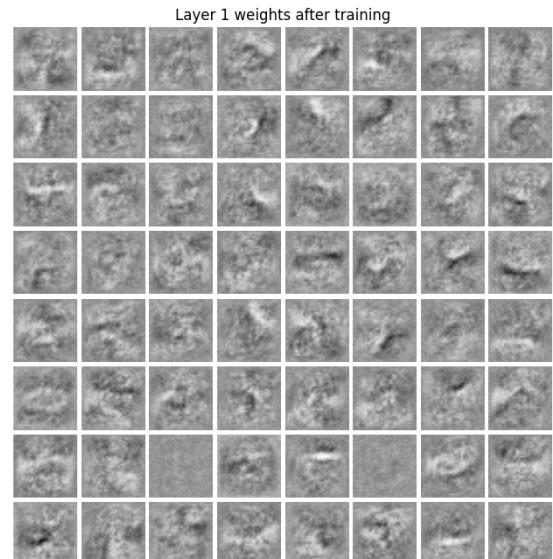


Figure 18: Layer1 Weights After Training

Q3.4 at page 6

Ans:

The Figure 19 shows the confusion matrix of my best model: learning rate = $2e^{-3}$, batch size = 64, epochs = 100. Some top pairs of classes that are most commonly confused are: 'O' and '0', '5' and 'S', '2' and 'Z', 'I' and '1'.

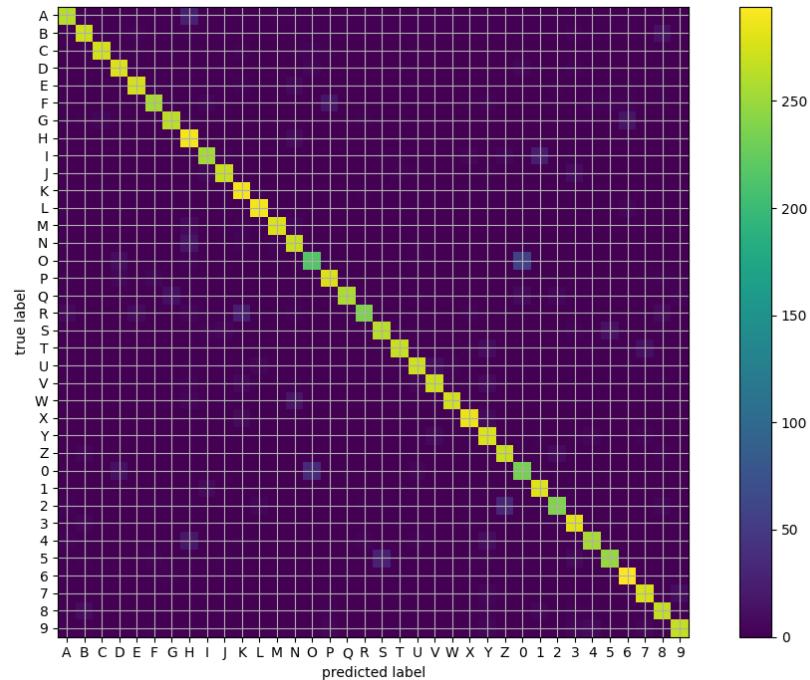


Figure 19: Confusion Matrix of Best Model

Q4.1 at page 8

Ans:

Two assumptions are made as the following:

- Each individual letter is a connected component. In the step2 of the method outlined at page 7 in the hw4.pdf, the connected component is used, indicating that each letter should be a connected component, otherwise, the method would fail. For example, Figure 20 may fail to extract each individual letter because of not being connected component:

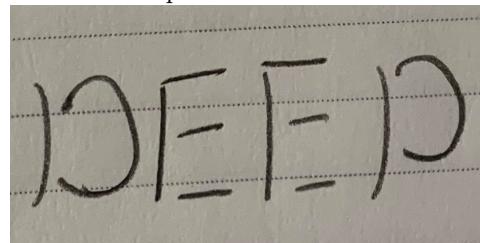


Figure 20: Example1 of Fail

- There should not be any connection between two individual letters. As in the step2 to grep the connected component as a letter, two individual letters should not be connected, otherwise, there may be failure on separating them. For example the Figure 21 may fail to extract each letter due to some connection between individual letters, such as 'A' and 'C', 'C' and 'H', 'E' and 'A', ..., and so on:

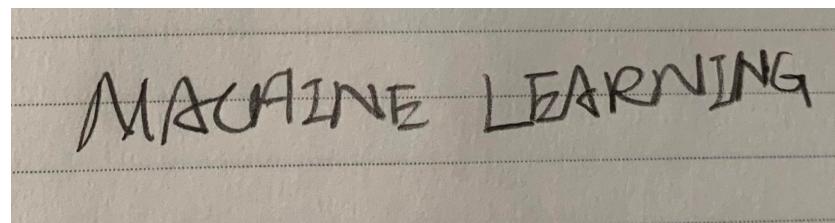


Figure 21: Example2 of Fail

Q4.2 at page 8

Ans:

The following Figure 22 shows the code snippet:

```
13 # takes a color image
14 # returns a list of bounding boxes and black_and_white image
15 def findletters(image):
16     bboxes = []
17     bw = None
18     # insert processing in here
19     # one idea estimate noise -> denoise -> greyscale -> threshold -> morphology -> label -> skip small boxes
20     # this can be 10 to 15 lines of code using skimage functions
21
22     ##### your code here #####
23     #####
24     #####
25     # To float
26     image_float = skimage.img_as_float(image)
27
28     # Blur/Denoise
29     image_gau = skimage.filters.gaussian(image_float, sigma=2.3, channel_axis=2)
30
31     # Grayscale
32     image_gray = skimage.color.rgb2gray(image_gau)
33
34     # Threshold
35     thresh_val = skimage.filters.threshold_otsu(image_gray)
36     binary_image = image_gray < thresh_val # Number is white, background is black, for skimage.segmentation.clear_border()
37
38     # Morphological Closing: Dilation + Erosion
39     bw_cl = skimage.morphology.closing(binary_image, skimage.morphology.octagon(2, 4))
40
41     # Clearing: As in tutorial, clear white component touching border
42     bw = skimage.segmentation.clear_border(bw_cl)
43     bw = skimage.img_as_float(bw)
44
45     # Invert to have number in black and background in white as required
46     bw = 1.0 - bw
47
48     # Label image regions
49     labels = skimage.measure.label(bw, connectivity=2, background=1)
50
51     for region in skimage.measure.regionprops(labels):
52         # take regions with large enough areas
53         if region.area > 600:
54             # draw rectangle around segmented coins
55             bboxes.append(region.bbox)
56
57     ...
58     fig, axs = plt.subplots(1, 1, figsize=(10, 5))
59     axs.imshow(bw, cmap='gray')
60     axs.set_title("binary_image Image")
61     axs.axis('off')
62     plt.show()
63     a = input()
64     ...
65
66     return bboxes, bw
```

Figure 22: Code Snippet of findLetters()

Q4.3 at page 8

Ans:

Results are show as the following Figure 23 to Figure 26.

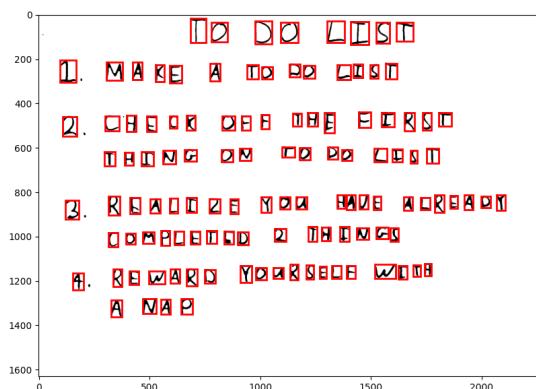


Figure 23: Result of Located Boxes in 01_list.jpg

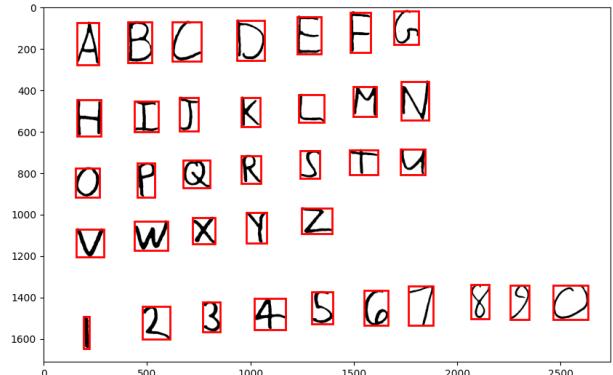


Figure 24: Result of Located Boxes in 02_letters.jpg

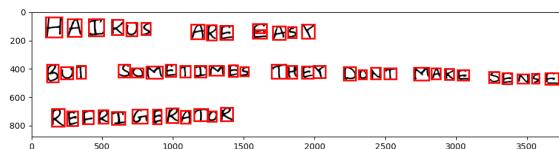


Figure 25: Result of Located Boxes in 03_haiku.jpg



Figure 26: Result of Located Boxes in 04_deep.jpg

Q4.4 at page 8

Ans:

Results are shown as the following Figure 27 - Figure 38. The correction rate of 01.list.jpg is 80.8696%, the correction rate of 02_letters.jpg is 83.333%, the correction rate of 03_haiku.jpg is 88.889%, and the 04_deep.jpg is 85.366%.

```
TO DO LIST
I H A 8 E A T O D D 6 I S T
2 C H E E K U F 8 7 H 8 F I R S T
T H I N G 0 N T O D O L I S T
3 R F A L I Z E Y 0 U H A V E A L R E A D X
C O M P L E T E D 2 Y H I N G S
4 R E W A R D Y 0 U R S E L F W I T H
A N A P
correction rate = 80.8695652173913%
```

Figure 27: Output result of the model.

```
TO DO LIST
I H A 8 E A T O D D 6 I S T
2 C H E E K U F 8 7 H 8 F I R S T
T H I N G 0 N T O D O L I S T
3 R F A L I Z E Y 0 U H A V E A L R E A D X
C O M P L E T E D 2 Y H I N G S
4 R E W A R D Y 0 U R S E L F W I T H
A N A P
```

Figure 28: Add space back to the result.

TO DO LIST
 1. MAKE A TO DO LIST
 2. CHECK OFF THE FIRST THING ON TO DO LIST
 3. REALIZE YOU HAVE ALREADY COMPLETED 2 THINGS
 4. REWARD YOURSELF WITH A NAP

Figure 29: Ground Truth: 01.list.jpg

```
A B C D E F G
H I J K L M N
O P Q 8 S T V
V W X Y Z
1 Z 3 4 S 6 7 X 9 0
correction rate = 83.333333333334%
```

Figure 30: Output result of the model.

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

Figure 31: Add space back to the result.

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 2 3 4 5 6 7 8 9 0

Figure 32: Ground Truth: 02.letters.jpg

```
HAIKUS ARE EASY
BUT SOMETIMES THEY DONT MAKE SENSE
REFRIGERATOR
correction rate = 88.888888888889%
```

Figure 33: Output result of the model.

```
HAIKUS ARE EASY
BUT SOMETIMES THEY DONT MAKE SENSE
REFRIGERATOR
```

Figure 34: Add space back to the result.

HAIKUS ARE EASY
 BUT SOMETIMES THEY DONT MAKE SENSE
 REFRIGERATOR

Figure 35: Ground Truth: 03.haiku.jpg

```
DEEPE2RHINH
DEEPERLEARHIN6
DEEPESTLEARNIHG
correction rate = 85.36585365853658%
```

Figure 36: Output result of the model.

```
DEEP LE2RHINH
DEEPER LEARHIN6
DEEPEST LEARNIHG
```

Figure 37: Add space back to the result.

DEEP LEARNING
 DEEPER LEARNING
 DEEPEST LEARNING

Figure 38: Ground Truth: 04.deep.jpg

Q5 at page 8

Ans:

Two

Collaborations

Ans:

Though I do not have collaborators, I found the following websites helpful on understanding the concepts in this homework.

1. https://www.ri.cmu.edu/pub_files/pub3/baker_simon_2003_3/baker_simon_2003_3.pdf.
2. <http://16385.courses.cs.cmu.edu/spring2024/lecture/track>
3. <https://stats.stackexchange.com/questions/559575/how-does-addition-of-a-regularization-term-ensure>
4. https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html
5. https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.imsave.html
6. https://docs.opencv.org/4.x/d4/d61/tutorial_warp_affine.html