CMU Fall24 16820 Homework 4

Patrick Chen

October 27, 2024

Q1.1 at page 3

Ans:

We have the following equation:

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

$$= \frac{e^c e^{x_i}}{e^c \sum_j e^{x_j}} \tag{2}$$

$$= \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{3}$$

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

The idea to use c = -max_{x_i} is beneficial because we can have more numerical stablility 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 $\sum \frac{e^{x_i}}{\sum e^{x_i}} = \frac{\sum e^{x_i}}{\sum e^{x_i}} = 1$.

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: $softmax(x_i) = \frac{1}{S} S_i$ 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):

$$y = W_n x_n + b_n \tag{5}$$
$$= W_n(W_{n-1}x_{n-1} + b_{n-1}) + b_n \tag{6}$$
$$= W_n W_{n-1} x_{n-1} + W_n b_{n-1} + b_n \tag{7}$$
$$= W_n W_{n-1}(W_{n-2}(.....(W_1 x_1 + b_1))) + W_n b_{n-1} + b_n \tag{8}$$
$$= W^{'} x^{'} + b^{'}, where\ W^{'} = W_n W_{n-1}...W_1,\ x^{'} = x_1,\ b^{'} = W_n W_{n-1}...W_2 b_1 + ... + W_n b_{n-1} + b_n \tag{9}$$

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

Q1.4 at page 3

Ans:

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

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

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

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

$$= (1-\sigma(x))\sigma(x) \tag{14}$$

Q1.5 at page 3

Ans:

$$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial W} \tag{15}$$

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

$$\frac{\partial J}{\partial x} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial x} \tag{17}$$

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

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial b} \tag{19}$$

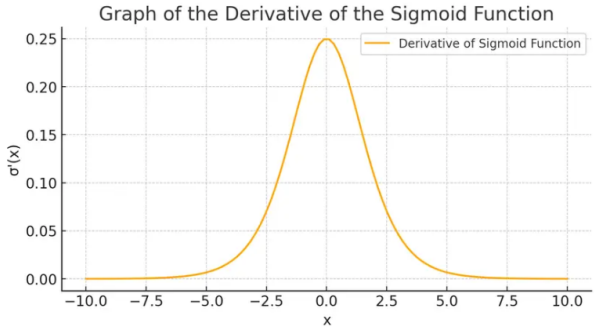
$$= \delta \circ \mathbf{1}_k, \text{where } \circ \text{ is the element-wised multiplication, and } \mathbf{1}_k \text{ is a kx1 vector with all elements set to 1.} \tag{20}$$

$$= \delta \in \mathbb{R}^{k \times 1} \tag{21}$$

Q1.6 at page 3

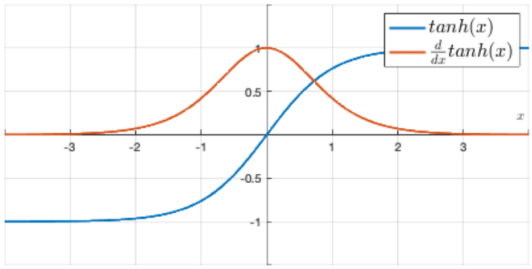
Ans:

1. 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.



2. 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.

3. 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.



4. The derivation of tanh(x) in terms of σ(x) is shown below:

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

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

$$= \sigma(2x) - (1 - \sigma(2x)) \tag{24}$$

$$= 2\sigma(2x) - 1 \tag{25}$$

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:

```python
def xavier_uniform(in_size,out_size):
    #np.random.seed(42) #for test
    a = np.sqrt(6/(in_size+out_size))
    return np.random.uniform(-a, a, (in_size, out_size))

def initialize_weights(in_size, out_size, params, name=""):
    W, b = None, None

    ##########################
    ##### your code here #####
    ##########################
    W = xavier_uniform(in_size, out_size)
    b = np.zeros(out_size)

    params["W" + name] = W
    params["b" + name] = b

```

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:

```python
############################## Q 2.2.1 ##############################
# x is a matrix
# a sigmoid activation function
def sigmoid(x):
    res = None

    #########################
    ##### your code here #####
    #########################
    res = 1/(1+np.exp(-x))

    return res


############################## Q 2.2.1 ##############################
def forward(X, params, name="", activation=sigmoid):
    """
    Do a forward pass

    Keyword arguments:
    X -- input vector [Examples x D]
    params -- a dictionary containing parameters
    name -- name of the layer
    activation -- the activation function (default is sigmoid)
    """
    pre_act, post_act = None, None
    # get the layer parameters
    W = params["W" + name]
    b = params["b" + name]

    #########################
    ##### your code here #####
    #########################
    pre_act = X@W + b
    post_act = activation(pre_act)

    # store the pre-activation and post-activation values
    # these will be important in backprop
    params["cache_" + name] = (X, pre_act, post_act)

    return post_act
```

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_pron = x - np.max(x, axis=-1).reshape(-1, 1)
110      exp_x = np.exp(x_pron)
111      res = exp_x/np.sum(exp_x, axis=-1).reshape(-1, 1)
112      return res
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     ##########################
123     ##### your code here #####
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      ##########################
162      ##### your code here #####
163      ##########################
164      grad_Z = delta * activation_deriv(post_act)
165      grad_X = grad_Z @ W.T
166      grad_W = X.T @ grad_Z
167      grad_b = np.sum(grad_Z, axis=0)
168
169      # store the gradients
170      params["grad_W" + name] = grad_W
171      params["grad_b" + name] = grad_b
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:

```python
############################## Q 2.4 ##############################
# split x and y into random batches
# return a list of [(batch1_x,batch1_y)...]
def shuffle(X, y):
    """
    :param X: The original input data in the order of the file.
    :param y: The original labels in the order of the file.
    :param epoch: The epoch number (0-indexed).
    :return: Permuted X and y training data for the epoch.
    """
    N = len(y)
    ordering = np.random.permutation(N)
    return X[ordering], y[ordering]

def get_random_batches(x, y, batch_size):
    batches = []
    ########################
    ##### your code here #####
    ########################
    x, y = shuffle(x, y)
    for start_idx in range(0, len(y), batch_size):
        batch_x = x[start_idx:start_idx+batch_size, :]
        batch_y = y[start_idx:start_idx+batch_size, :]
        batches.append((batch_x, batch_y))

    return batches
```

Figure 6: Code Snippet of Random Batch

```python
# Q 2.4
batches = get_random_batches(x, y, 5)
# print batch sizes
print([_[0].shape[0] for _ in batches])
batch_num = len(batches)

# WRITE A TRAINING LOOP HERE
max_iters = 500
learning_rate = 1e-3
# with default settings, you should get loss < 35 and accuracy > 75%
for itr in range(max_iters):
    total_loss = 0
    avg_acc = 0
    for xb, yb in batches:
        ########################
        ##### your code here #####
        ########################
        # forward
        h1_out = forward(xb, params, 'layer1', sigmoid)
        y_hat  = forward(h1_out, params, 'output', softmax)


        # loss
        # be sure to add loss and accuracy to epoch totals
        loss, acc = compute_loss_and_acc(yb, y_hat)
        total_loss += loss
        avg_acc += acc

        # backward
        grad_y_hat = y_hat - yb
        delta2 = backwards(grad_y_hat, params, "output", linear_deriv)
        _ = backwards(delta2, params, "layer1", sigmoid_deriv)

        # apply gradient
        # gradients should be summed over batch samples
        params['Wlayer1'] -= learning_rate*params['grad_Wlayer1']
        params['blayer1'] -= learning_rate*params['grad_blayer1']
        params['Woutput'] -= learning_rate*params['grad_Woutput']
        params['boutput'] -= learning_rate*params['grad_boutput']

    avg_acc /= batch_num

    if itr % 100 == 0:
        print(
            "itr: {:02d} \t loss: {:.2f} \t acc : {:.2f}".format(
                itr, total_loss, avg_acc
            )
        )
```

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:

```python
# Q 2.5 should be implemented in this file
# you can do this before or after training the network.

def runForwardNN(x, y, params):
    h1 = forward(x, params, "layer1")
    probs = forward(h1, params, "output", softmax)
    loss, acc = compute_loss_and_acc(y, probs)

    return loss, acc

# compute gradients using forward and backward
h1 = forward(x, params, "layer1")
probs = forward(h1, params, "output", softmax)
loss, acc = compute_loss_and_acc(y, probs)
delta1 = probs - y
delta2 = backwards(delta1, params, "output", linear_deriv)
backwards(delta2, params, "layer1", sigmoid_deriv)

# save the old params
import copy

params_orig = copy.deepcopy(params)

# compute gradients using finite difference
eps = 1e-6
for k, v in params.items():
    if "_" in k:
        continue
    # for each value inside the parameter
    #   add epsilon
    #   run the network
    #   get the loss
    #   subtract 2*epsilon
    #   run the network
    #   get the loss
    #   restore the original parameter value
    #   compute derivative with central diffs
    if 'W' in k:
        for i in range(v.shape[0]):
            for j in range(v.shape[1]):
                orig_v = v[i, j]
                v[i, j] += eps
                loss_pos, acc_pos = runForwardNN(x, y, params)
                v[i, j] -= 2*eps
                loss_neg, acc_neg = runForwardNN(x, y, params)
                params['grad_'+k][i, j] = (loss_pos - loss_neg)/(2*eps)
                v[i, j] = orig_v

    if 'b' in k:
        for i in range(v.shape[0]):
            orig_v = v[i]
            v[i] += eps
            loss_pos, acc_pos = runForwardNN(x, y, params)
            v[i] -= 2*eps
            loss_neg, acc_neg = runForwardNN(x, y, params)
            params['grad_'+k][i] = (loss_pos - loss_neg)/(2*eps)
            v[i] = orig_v

    ##########################
    ##### your code here #####
    ##########################
```

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%.'

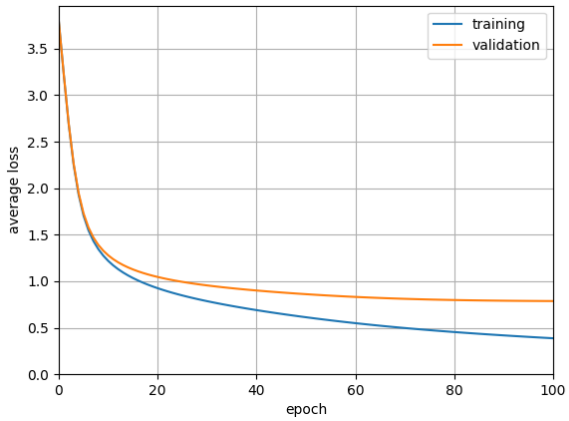


Figure 9: Average Loss vs Epoch

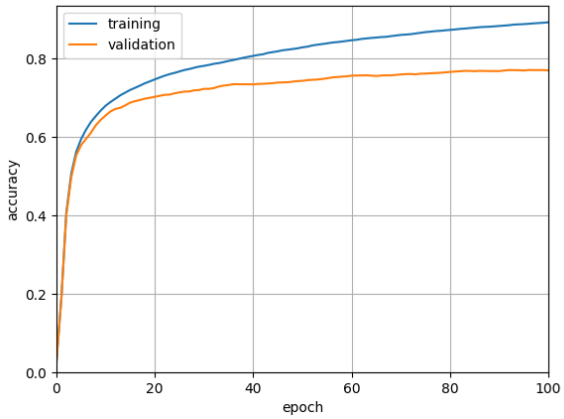


Figure 10: Accuracy vs Epoch

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.

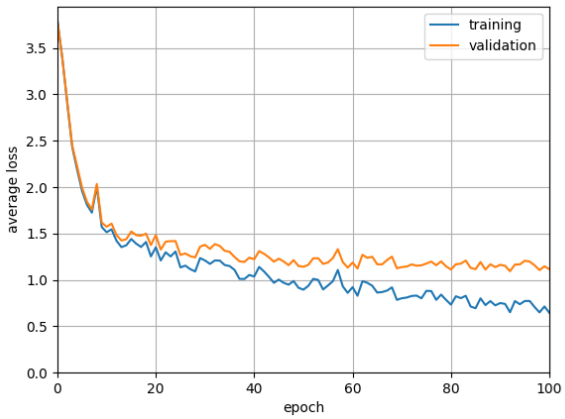


Figure 11: Average Loss vs Epoch

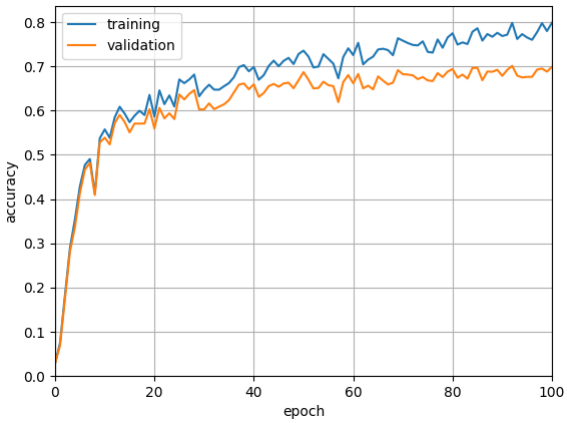


Figure 12: Accuracy vs Epoch

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%.

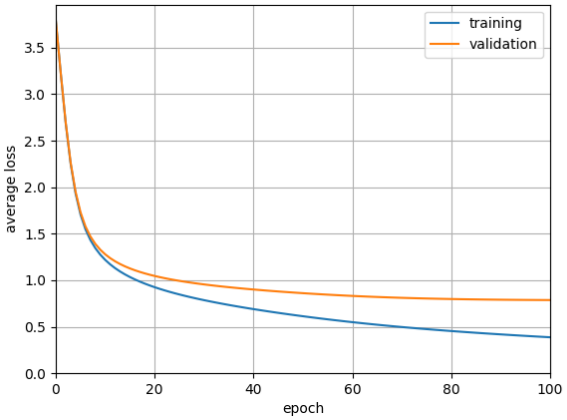


Figure 13: Average Loss vs Epoch

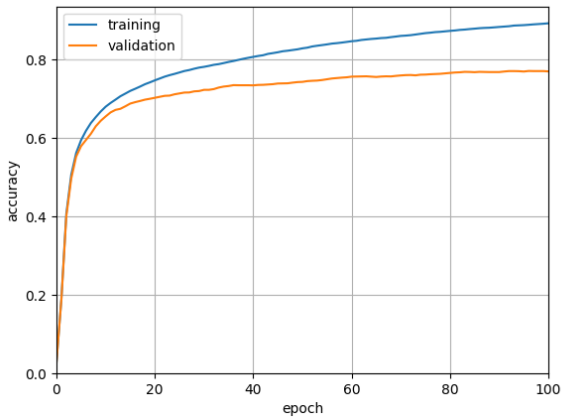


Figure 14: Accuracy vs Epoch

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.

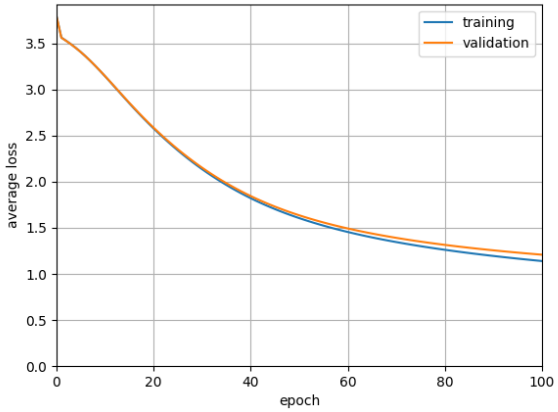


Figure 15: Average Loss vs Epoch

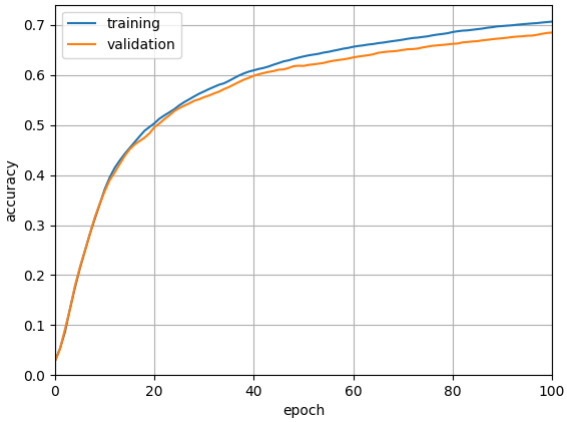


Figure 16: Accuracy vs Epoch

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.

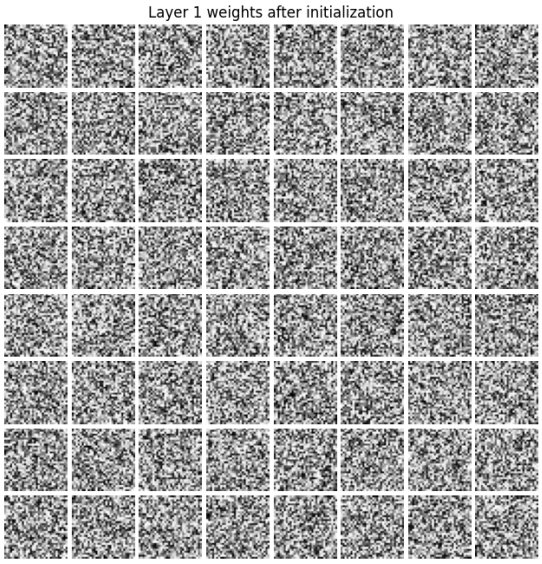


Figure 17: Layer1 Weights Initialization

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'.

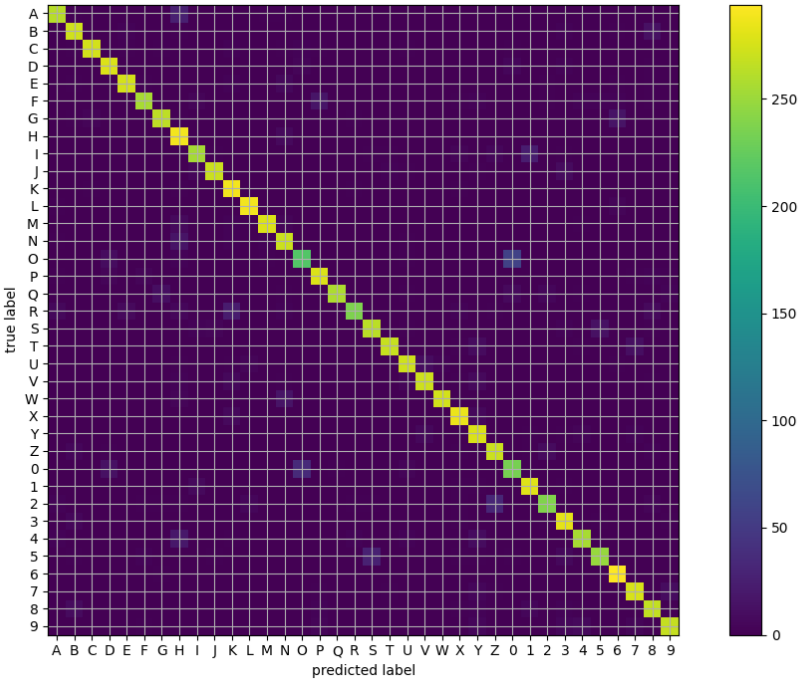


Figure 19: Confusion Matrix of Best Model

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:

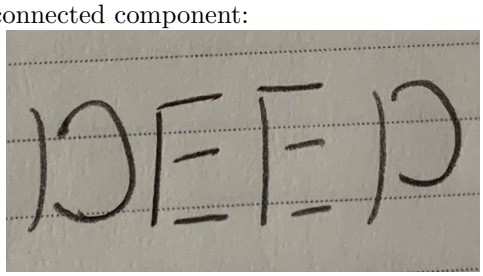


Figure 20: Example1 of Fail

- 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:

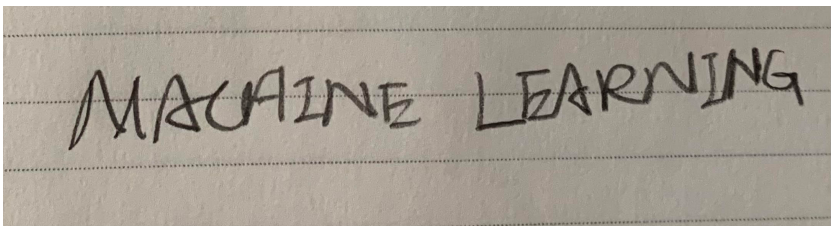


Figure 21: Example2 of Fail

Q4.2 at page 8

Ans:

The following Figure 22 shows the code snippet:

```python
# takes a color image
# returns a list of bounding boxes and black_and_white image
def findLetters(image):
    bboxes = []
    bw = None
    # insert processing in here
    # one idea estimate noise -> denoise -> greyscale -> threshold -> morphology -> label -> skip small boxes
    # this can be 10 to 15 lines of code using skimage functions

    ##########################
    ##### your code here #####
    ##########################
    # To float
    image_float = skimage.img_as_float(image)

    # Blur/Denoise
    image_gau = skimage.filters.gaussian(image_float, sigma=2.3, channel_axis=2)

    # Grayscale
    image_gray = skimage.color.rgb2gray(image_gau)

    # Threshold
    thresh_val = skimage.filters.threshold_otsu(image_gray)
    binary_image = image_gray < thresh_val # Number is white, background is black, for skimage.segmentation.clear_border()

    # Morphological Closing: Dilation + Erosion
    bw_cl = skimage.morphology.closing(binary_image, skimage.morphology.octagon(2, 4))

    # Clearing: As in tutorial, clear white component toudhing border
    bw = skimage.segmentation.clear_border(bw_cl)
    bw = skimage.img_as_float(bw)

    # Invert to have number in black and background in white as required
    bw = 1.0 - bw

    # Label image regions
    labels = skimage.measure.label(bw, connectivity=2, background=1)

    for region in skimage.measure.regionprops(labels):
        # take regions with large enough areas
        if region.area >= 600:
            # draw rectangle around segmented coins
            bboxes.append(region.bbox)

    '''
    fig, axs = plt.subplots(1, 1, figsize=(10, 5))
    axs.imshow(bw, cmap='gray')
    axs.set_title(f"binary_image Image")
    axs.axis('off')
    plt.show()
    a = input()
    '''
    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.

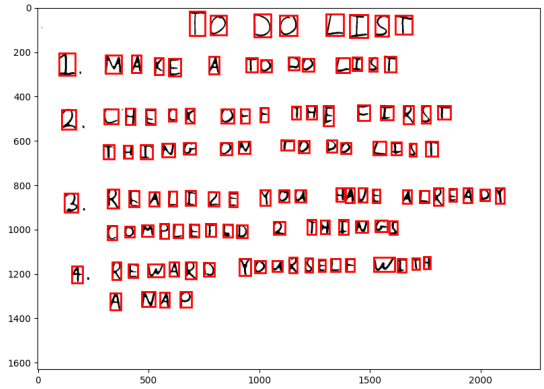


Figure 23: Result of Located Boxes in 01_list.jpg

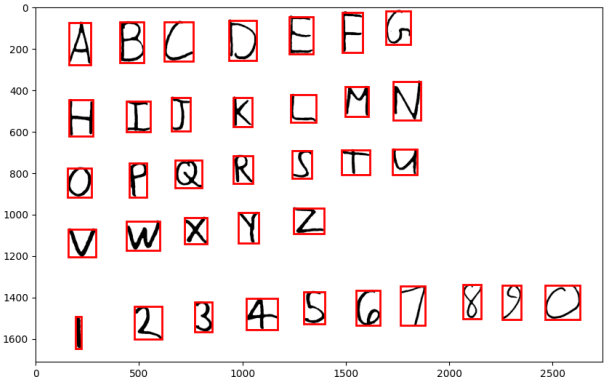


Figure 24: Result of Located Boxes in 02_letters.jpg

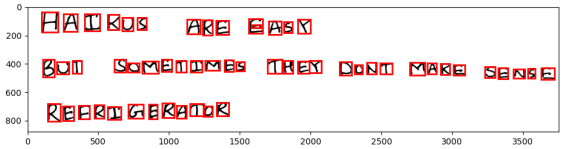


Figure 25: Result of Located Boxes in 03_haiku.jpg

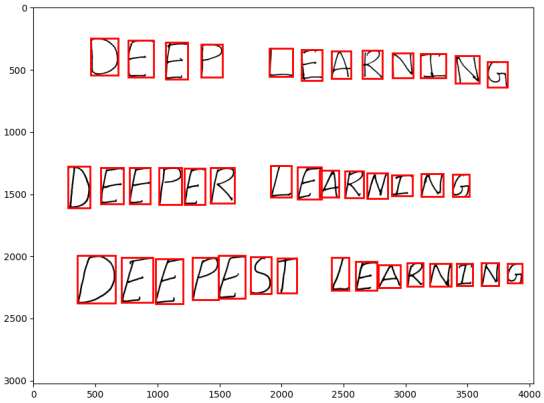


Figure 26: Result of Located Boxes in 04_deep.jpg

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%.

```
T0D0LIST
IHA8EAT0DD6IST
2CHEEKUF87H8FIRST
THING0NT0D0LIST
3RFALIZEY0UHAVEALREADX
C0MPLETED2YHINGS
4REWARDY0URSELFWITH
ANAP
correction rate = 80.8695652173913%
```

Figure 27: Output result of the model.

```
TO DO LIST
I HA8E A TO DD 6IST
2 CHEEK UF8 7H8 FIRST
THING ON TO DO LIST
3 RFALIZE YOU HAVE ALREADX
COMPLETED 2 YHINGS
4 REWARD YOURSELF WITH
A NAP
```

Figure 28: Add space back to the result.

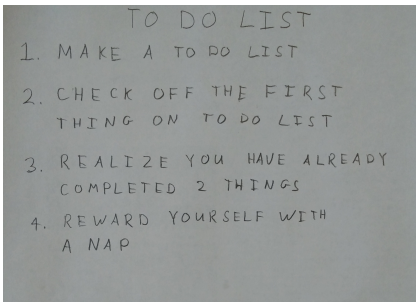


Figure 29: Ground Truth: 01_list.jpg

```
ABCDEFG
HIJKLMN
0PQ8STV
VWXYZ
1Z34S67X90
correction rate = 83.33333333333334%
```

Figure 30: Output result of the model.

```
A B C D E F G
H I J K L M N
0 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.

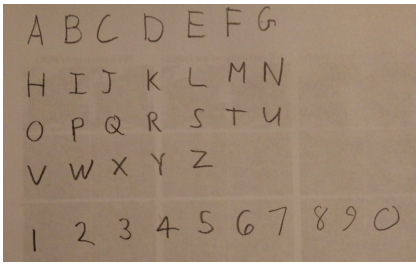


Figure 32: Ground Truth: 02_letters.jpg

```
HAIKUSAREEASY
BUTSOMETIMESTHEXD0NTMAKRSEMGE
REFRI6ERATOR
correction rate = 88.88888888888889%
```

Figure 33: Output result of the model.

```
HAIKUS ARE EASY
BUT SOMETIMES THEX D0NT MAKR SEMGE
REFRI6ERATOR
```

Figure 34: Add space back to the result.

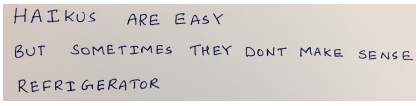


Figure 35: Ground Truth: 03_haiku.jpg

```
DEEPLE2RHINH
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.

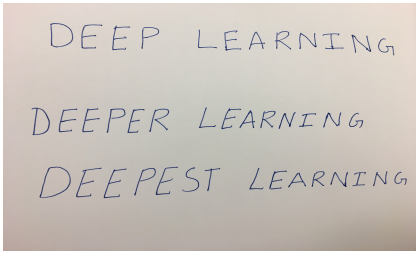


Figure 38: Ground Truth: 04_deep.jpg

Q5.1.1 at page 8

Ans:

The following Figure 39 shows the code snippet of weight initializaing (lines 38-51), forward pass (lines 74-77), loss function calculation (line 81), and the backward pass (lines 84-88).

```
35  # Initialize weights
36  n_inst, x_dimension = train_x.shape
37  n_inst_val, x_dimension_val = valid_x.shape
38  initialize_weights(x_dimension, hidden_size, params, "layer1")
39  initialize_weights(hidden_size, hidden_size, params, "layer2")
40  initialize_weights(hidden_size, hidden_size, params, "layer3")
41  initialize_weights(hidden_size, x_dimension, params, "output")
42  assert x_dimension == x_dimension_val
43
44  # Initialize momentum terms
45  upd_parameters_name = []
46  grad_parameters_name = []
47  momentum_params = Counter()
48  for key in params.keys():
49      momentum_params[f'm_{key}'] = np.zeros_like(params[key])
50      upd_parameters_name.append(key)
51      grad_parameters_name.append('grad_'+key)
52
53
54  # should look like your previous training loops
55  losses = []
56  valid_losses = []
57  for itr in range(max_iters):
58      total_loss = 0
59      for xb,_ in batches:
60          # training loop can be exactly the same as q2!
61          # your loss is now squared error
62          # delta is the d/dx of (x-y)^2
63          # to implement momentum
64          #   use 'm_'+name variables in initialize_weights from nn.py
65          #   to keep a saved value
66          #   params is a Counter(), which returns a 0 if an element is missing
67          #   so you should be able to write your loop without any special conditions
68
69          ##########################
70          ##### your code here #####
71          ##########################
72
73          # forward pass
74          h1_out = forward(xb    , params, 'layer1', relu)
75          h2_out = forward(h1_out, params, 'layer2', relu)
76          h3_out = forward(h2_out, params, 'layer3', relu)
77          y_hat  = forward(h3_out, params, 'output', sigmoid)
78
79
80          # loss
81          total_loss += np.sum(np.power((y_hat - xb), 2))
82
83          # backward
84          grad_y_hat = 2*(y_hat -xb)
85          delta3 = backwards(grad_y_hat, params, "output", sigmoid_deriv)
86          delta2 = backwards(delta3    , params, "layer3", relu_deriv)
87          delta1 = backwards(delta2    , params, "layer2", relu_deriv)
88          _      = backwards(delta1    , params, "layer1", relu_deriv)
```

Figure 39: Code Snippet

Q5.1.2 at page 9

Ans:

The following Figure 40 shows the code snippet of momentum gradient updating of the weights (lines 91-94).

```
89
90          # apply gradient, remember to update momentum as well
91          for param_name, grad in zip(upd_parameters_name, [params[grad_parameters_name[i]] for i in range(len(grad_parameters_name))]):
92              # Momentum update rule
93              momentum_params[f'm_{param_name}'] = 0.9 * momentum_params[f'm_{param_name}'] - learning_rate * grad
94              params[param_name] += momentum_params[f'm_{param_name}']
95
96      '''
97      h1_out_val = forward(valid_x   , params, 'layer1', relu)
98      h2_out_val = forward(h1_out_val, params, 'layer2', relu)
99      h3_out_val = forward(h2_out_val, params, 'layer3', relu)
100     y_hat_val  = forward(h3_out_val, params, 'output', sigmoid)
101     valid_loss = np.sum(np.power((y_hat_val - valid_x), 2))
102     valid_losses.append(valid_loss / n_inst_val)
103     '''
104
105     losses.append(total_loss/train_x.shape[0])
106     if itr % 2 == 0:
107         print("itr: {:02d} \t loss: {:.2f}".format(itr,total_loss))
108     if itr % lr_rate == lr_rate-1:
109         learning_rate *= 0.9
110
111 # plot loss curve
112 plt.plot(range(len(losses)), losses, 'r', label='Training Loss')  # Training loss in red
113 #plt.plot(range(len(valid_losses)), valid_losses, 'y', label='Validation Loss')  # Validation loss in blue
114 plt.xlabel("epoch")
115 plt.ylabel("average loss")
116 #plt.xlim(0, len(losses)-1)
117 plt.xlim(0, max(len(losses), len(valid_losses)) - 1)
118 plt.ylim(0, None)
119 plt.legend()  # Display legend to show color meaning
120 plt.grid()
121 plt.show()
122
```

Figure 40: Code Snippet

Q5.2 at page 9

Ans:

The following Figure 41 shows the result after 100 epochs with learning rate set to $3e^{-5}$, batch size set to 36, hidden size set to 32. I observe that at the beginning 20 epochs, the loss drops drastically and quickly. However, as the epoch number increases, the loss drop becomes smoother and more flattened. Finally, at the last epoch number from 90 to 100, the loss tends to converge to some values range from 20 to 30.

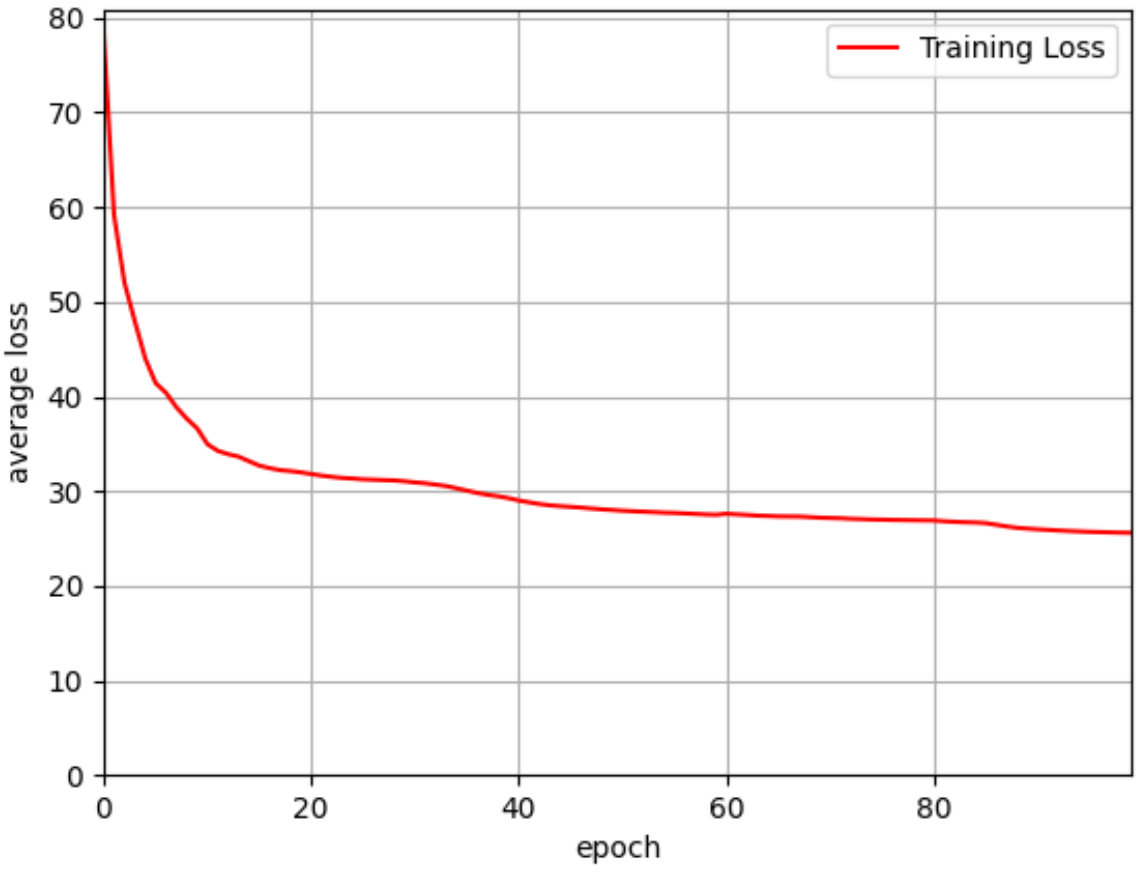


Figure 41: Average Loss vs Epoch

Q5.3.1 at page 9

Ans:

The following Figure 42 shows the original validation images and their corresponding reconstruction images. The reconstructed images show similar shape and structure compared with their original images, but they are more blurred. I think it is because the Autoencoder with limited hidden units can only preserve some latent features of the original images, so as to reach the goal of compressed images, but would inevitably lose some information.

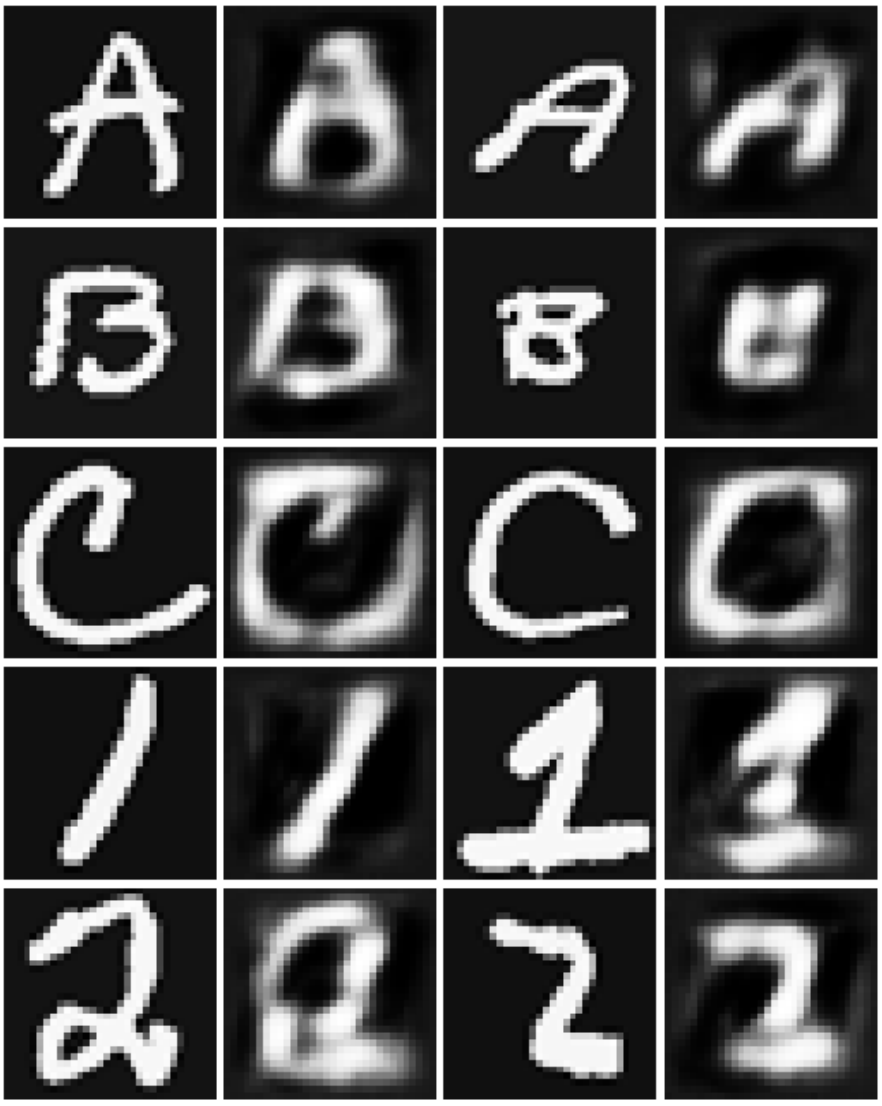


Figure 42: Validation Original Images and the Reconstruction Images

Q5.3.2 at page 9

Ans:

The following Figure 43 shows the average PSNR result of the validation dataset, which is 15.2388.

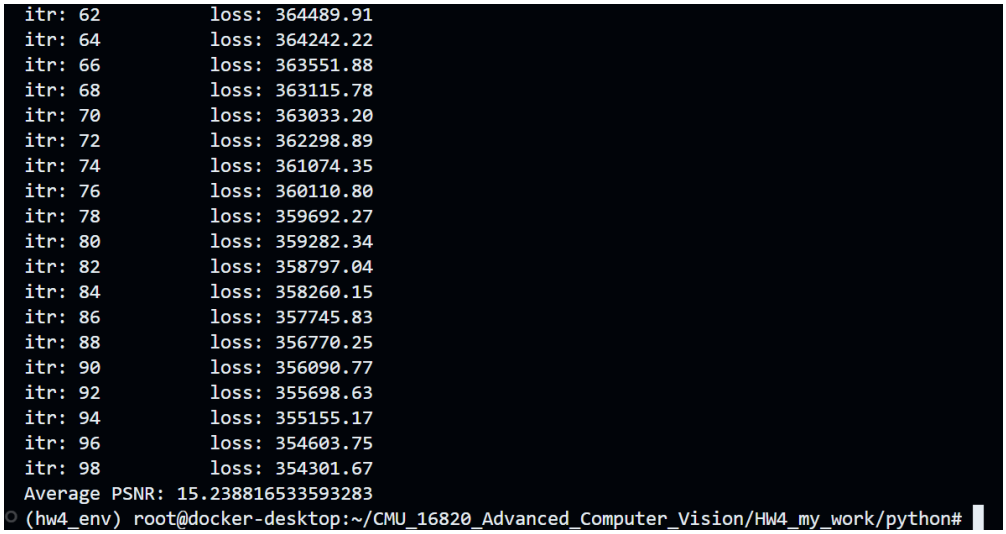


Figure 43: Average PSNR Result of Validation Dataset

Q6.1.1 at page 10

Ans:

The following Figure 44 shows the result of the 1-hidden-layer fully connected neural network written in Pytorch. The test accuracy is 83.28% as printed on the standard output from the program: run_q6_1_1.py. The following Figure 45 - Figure 47 shows the code snippet of the implementation in run_q6_1_1.py

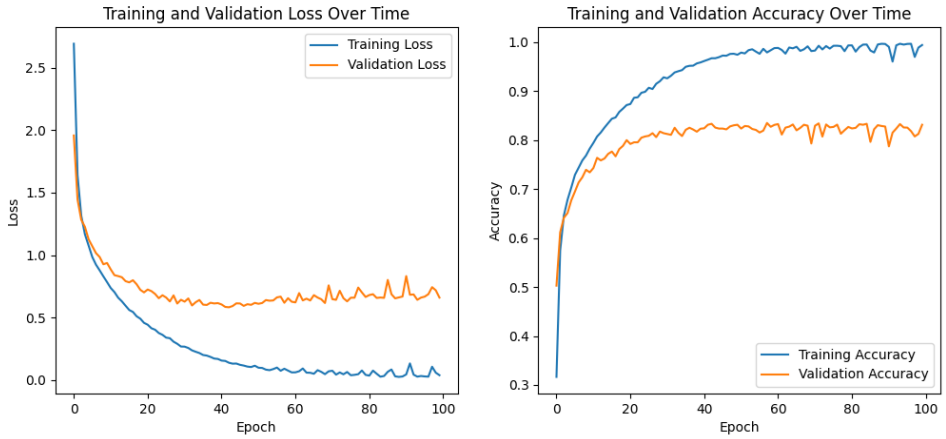


Figure 44: Loss and Accuracy Over Epoch on Training and Validation Dataset

```python
# Define the network architecture
class FullyConnectedNetwork(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(FullyConnectedNetwork, self).__init__()
        self.layer1 = nn.Linear(input_size, hidden_size)
        self.output_layer = nn.Linear(hidden_size, output_size)
        self.sigmoid = nn.Sigmoid()
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        h1_out = self.sigmoid(self.layer1(x))
        y_hat = self.output_layer(h1_out)
        #y_hat = self.softmax(self.output_layer(h1_out)) #Don't need softmax if using nn.CrossEntropyLoss() as the loss function
        return y_hat

### Sample dataloader For Q.6.1 ####
class NIST36_Data(torch.utils.data.Dataset):
    def __init__(self, type):
        self.type = type
        self.data = scipy.io.loadmat(f"../data/nist36_{type}.mat")
        self.inputs, self.one_hot_target = (
            self.data[f"{self.type}_data"],
            self.data[f"{self.type}_labels"],
        )
        self.target = np.argmax(self.one_hot_target, axis=1)

    def __len__(self):
        return self.inputs.shape[0]

    def __getitem__(self, index):
        inputs = torch.from_numpy(self.inputs[index]).type(torch.float32)
        target = torch.tensor(self.target[index]).type(torch.LongTensor)
        return inputs, target

def init_weights(m):
    if isinstance(m, nn.Linear):
        nn.init.xavier_uniform_(m.weight)
        if m.bias is not None:
            nn.init.zeros_(m.bias)
```

Figure 45: Code Snippet1 of Q6.1.1

```python
# Hyperparameters
hidden_size = 256
output_size = 36  # For NIST36 (36 classes)
#learning_rate = 2e-3
learning_rate = 1e-3
max_iters = 200
batch_size = 32

# Initialize datasets and dataloaders
train_data = NIST36_Data(type="train")
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
valid_data = NIST36_Data(type="valid")
valid_loader = DataLoader(valid_data, batch_size=batch_size, shuffle=False)

# Initialize model, loss, and optimizer
input_size = np.array(train_data[0][0]).shape[0]
model = FullyConnectedNetwork(input_size, hidden_size, output_size)
model.apply(init_weights)
criterion = nn.CrossEntropyLoss()  # Loss function
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Define a learning rate scheduler
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=100, gamma=0.5)

# Training loop
train_losses = []
valid_losses = []
train_accuracies = []
valid_accuracies = []
```

Figure 46: Code Snippet2 of Q6.1.1

```python
for itr in range(max_iters):
    ############################## Training ##########################
    model.train()  # Set the model to training mode
    total_train_loss = 0
    total_train_correct = 0

    for xb, yb in train_loader:
        optimizer.zero_grad()  # Zero the parameter gradients

        # Forward pass
        y_hat = model(xb)

        # Compute loss
        loss = criterion(y_hat, yb)
        total_train_loss += loss.item()

        # Compute accuracy
        _, predicted = torch.max(y_hat, 1)
        total_train_correct += (predicted == yb).float().sum().item()

        # Backward pass and update
        loss.backward()
        optimizer.step()

    avg_train_loss = total_train_loss / len(train_loader)
    avg_train_acc = total_train_correct / len(train_data)
    train_losses.append(avg_train_loss)
    train_accuracies.append(avg_train_acc)

    ############################## Validation ########################
    model.eval()  # Set the model to evaluation mode
    total_valid_loss = 0
    total_valid_correct = 0

    with torch.no_grad():  # Disable gradient computation for validation
        for xb, yb in valid_loader:
            # Forward pass
            y_hat = model(xb)

            # Compute loss and accuracy
            loss = criterion(y_hat, yb)
            total_valid_loss += loss.item()

            _, predicted = torch.max(y_hat, 1)
            total_valid_correct += (predicted == yb).float().sum().item()

    # Calculate average validation loss and accuracy
    avg_valid_loss = total_valid_loss / len(valid_loader)
    avg_valid_acc = total_valid_correct / len(valid_data)
    valid_losses.append(avg_valid_loss)
    valid_accuracies.append(avg_valid_acc)

    # Step the scheduler at the end of each epoch
    scheduler.step()

    # Print current learning rate for tracking
    current_lr = scheduler.get_last_lr()[0]

    print(f"Epoch {itr+1}/{max_iters}, "
          f"Train Loss: {avg_train_loss:.4f}, Train Acc: {avg_train_acc:.4f}, "
          f"Valid Loss: {avg_valid_loss:.4f}, Valid Acc: {avg_valid_acc:.4f}, "
          f"Learning rate after epoch {itr+1}: {current_lr:.6f}")

```

Figure 47: Code Snippet3 of Q6.1.1

Q6.1.2 at page 10

Ans:

The following Figure 48 shows the result of the convolutional neural network (CNN) written in Pytorch. The test accuracy is 93.56% as printed on the standard output from the program: run_q6_1_2.py. We can see from the result that CNN has higher test accuracy and less overfitting compared with the previous 1-hidden-layer fully-connected network. The following Figure 49 shows the code snippet of the implementation of CNN in run_q6_1_2.py

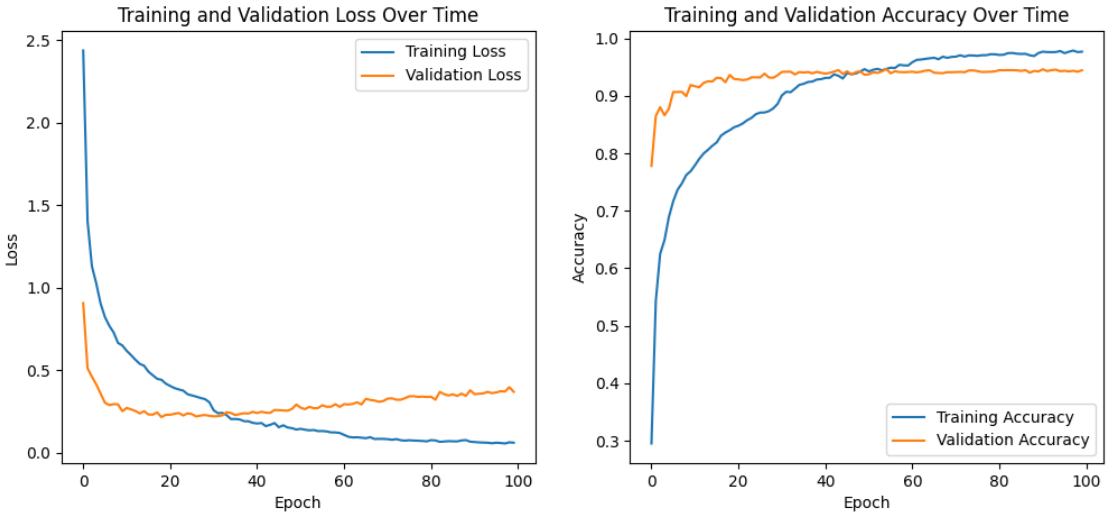


Figure 48: Loss and Accuracy Over Epoch on Training and Validation Dataset

```python
# Define the network architecture
class Net(nn.Module):
    def __init__(self, num_class, H, W):
        super(Net, self).__init__()

        # Define the convolutional layers using nn.Sequential
        # First convolutional layer
        self.conv1 = nn.Sequential(
            # Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0,
            #        dilation=1, groups=1, bias=True, padding_mode='zeros')
            nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1), # Output size: 32 x H x W
            nn.BatchNorm2d(32),
            # ReLU(inplace=False)
            nn.ReLU(),
            # MaxPool2d(kernel_size, stride=None, padding=0, dilation=1, return_indices=False, ceil_mode=False)
            nn.MaxPool2d(kernel_size=2, stride=2)  # Output size: 32 x H/2 x W/2
        )

        # Second convolutional layer
        self.conv2 = nn.Sequential(
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1), # Output size: 64 x H/2 x W/2
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)  # Output size: 64 x H/4 x W/4
        )

        # Third convolutional layer
        self.conv3 = nn.Sequential(
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1), # Output size: 128 x H/4 x W/4
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)  # Output size: 128 x H/8 x W/8
        )

        # Flatten layer
        self.flatten = nn.Flatten()

        # Fully connected layers
        # Linear(in_features, out_features, bias=True)
        self.fc1 = nn.Linear(128 * (H // 8) * (W // 8), 256)  # Adjust based on the input image size
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(256, num_class)  # Adjust the output size for the number of classes

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)

        x = self.flatten(x)  # Flatten the output for the fully connected layers
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x) # using nn.CrossEntropyLoss() as your loss func, then you do not need to specify softmax, it is already inside the loss function.

        return x
```

Figure 49: Code Snippet of CNN

Q6.1.3 at page 10

Ans:

The following Figure 50 shows the result of the same convolutional neural network (CNN) in previous problem running on CIFAR-10 dataset. The test accuracy is 79.28% as printed on the standard output from the program: run_q6_1_3.py. The result shows that under same configuration and parameter setting method, the accuracy is lower and the overfitting is more severe, indicating CIFAR-10 dataset is more complex and versatile than NIST36 dataset.

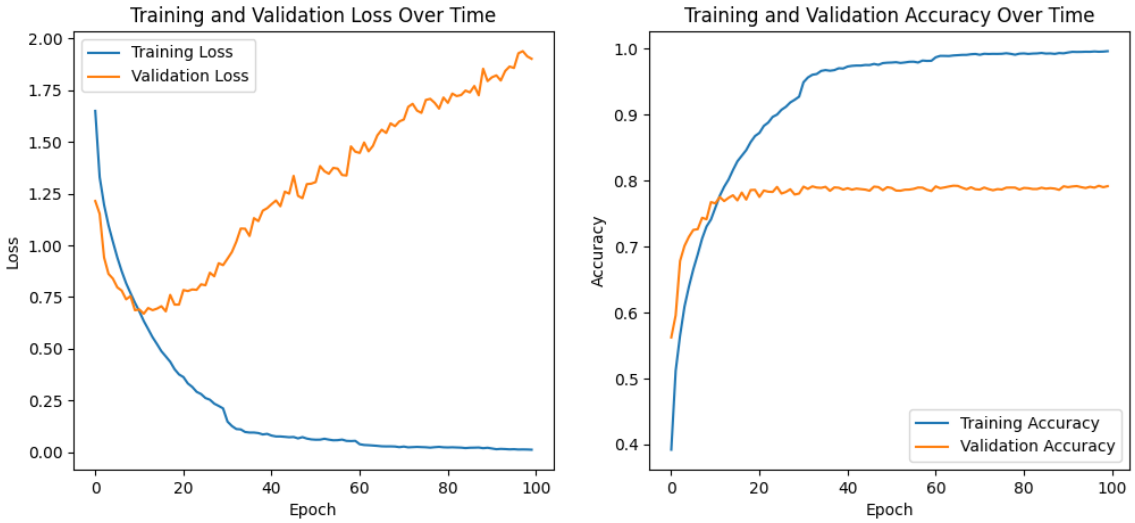


Figure 50: Loss and Accuracy Over Epoch on Training and Validation Dataset

Q6.2 at page 10

Ans:

The following Figure 51 shows the result of fine-tune of SqueezeNet1_1 with replacing last classifier layer with a Convolution Layer that has kernel size = (1, 1), stride = 1, and output channel = 17 (number of class). With learning rate $2e^{-3}$, batch size = 32, and epochs = 100, the test accuracy is 87.06% on flower17 dataset as the output of running run_q6_2.finetune.py.

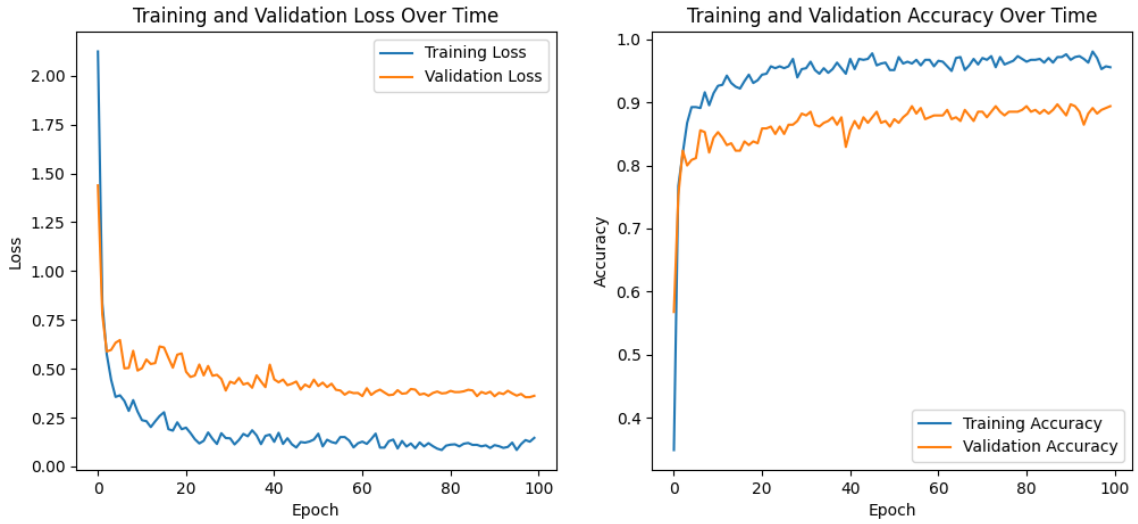


Figure 51: Loss and Accuracy Over Epoch with Fine-tune on SqueezeNet1_1

The following Figure 52 shows the result of my own CNN architecture used in Q6.1.2 and Q6.1.3. With learning rate $1e^{-5}$, batch size = 32, and epochs = 100, the test accuracy is 44.12% on flower17 dataset as the output of running run_q6_2.orig.py.

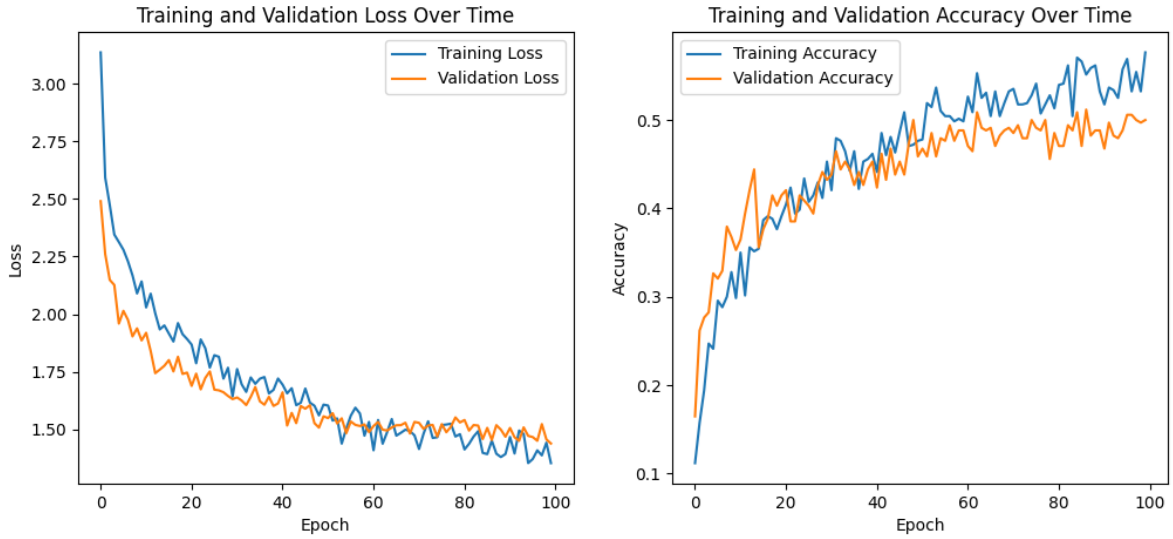


Figure 52: Loss and Accuracy Over Epoch with Fine-tune on My Own CNN

It is obvious that the SeueezeNet1_1 with fine-tuned classifier has performance much better than my own CNN architecture, indicating the my CNN model is too simple for the complex flower17 dataset.

Besides, the fine-tuned SqueezeNet1_1 converges much more faster than my model, using only 20 epochs to get the final performance. However, it seems that my own CNN model has smaller overfitting than the fine-tuned SqueezeNet1_1, as the gap between validation loss and training loss is smaller.

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. https://pytorch.org/tutorials/beginner/pytorch_with_examples.html
3. https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.imsave.html
4. https://www.geeksforgeeks.org/how-to-load-cifar10-dataset-in-pytorch/
5. https://www.geeksforgeeks.org/how-do-you-use-pytorchs-dataset-and-dataloader-classes-for-custo
6. https://github.com/facebookarchive/fb.resnet.torch/issues/180
7. https://pytorch.org/tutorials/beginner/data_loading_tutorial.html#afterword-torchvision
8. https://www.kaggle.com/code/satriasyahputra/flowers-17