Com Sci 146 Homework 4
Krish Patel StUID : 605796227
Professor: Aditya Grover

Krish Patel
605 796 227

Com Sci M146
Homework 4

Q1) a) Data point $x^{(1)} = \{5.51, 5.35\}$
projection coordinate = $v_1 \cdot x^{(1)}$
$\{5.51, 5.35\} \cdot \{0.694, 0.720\}$
$= 3.82394 + 3.9816 \approx 7.80554$ or $\underline{7.81}$

b) $y_1^{(1)} = \underline{7.81}$ from (1)a)
to reconstruct, we multiply with the ~~vector~~ of v_1
$= 7.80554 \times 0.694 + \mu_1 = 5.417 + \mu_1$
$= 7.80554 \times 0.720 + \mu_2 = 5.6199 + \mu_2$
μ_1 = mean of x_1 = 9.29375
μ_2 = mean of x_2 = 9.685
final reconstructed vector $\approx (14.7107, 15.3049)$

c) PCS → Principal component score $x_1 = (5.51, 5.35)$
Principle component score along the SPD eigenvectors. The eigenvector (perpendicular to v_1 would be ~~$\{0.694, -0.72\}$~~ $= \{0.720, -0.694\}$
Now, calculating the score, it would be
$x_1 \cdot v_2 = (5.51, 5.35) \cdot (0.72, -0.694)$
$= \underline{0.2543}$

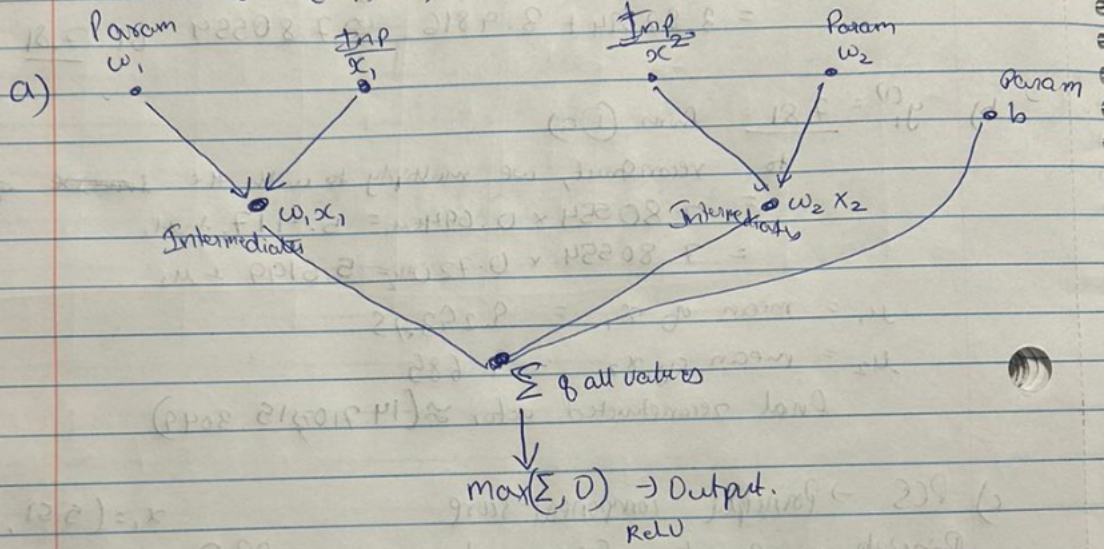
Thus the PCS for v_2 is $\underline{0.2543}$

Problem 2

a) $f_\theta(x) = \max(w_1 x_1 + w_2 x_2 + b, 0)$ $\quad x = (x_1, x_2)^T \in \mathbb{R}^2$

$\theta = (w_1, w_2, b)^T \in \mathbb{R}^3$

a)



b) $w_1 = 2 \quad w_2 = 1 \quad x_1 = 3 \quad x_2 = -2 \quad b = -1$

$$f_\theta(x) = \max(2 \times 3 + 1 \times (-2) - 1, 0)$$
$$= \max(3, 0) = 3$$

Output of forward pass is 3.

c) Computing using backward pass. Thus the values are $(3, -2, 1)$

$\frac{\partial (f_\theta(x))}{\partial w_1} = \frac{\partial}{\partial w_1}(w_1 x_1 + w_2 x_2 + b) = x_1 = 3$

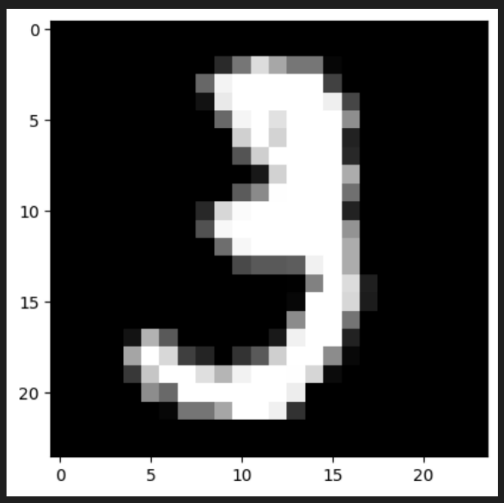
$\frac{\partial (f_\theta(x))}{\partial w_2} = \frac{\partial}{\partial w_2}(\quad\quad) = x_2 = -2$

$\frac{\partial (f_\theta(x))}{\partial b} = \frac{\partial}{\partial w_3}(\quad\quad) = 1$

Problem 3:

a)

```
    # PART (a):
~/Desktop/gitrepos/timesheet/server/src/
models/shift.ts
                                    int8').reshape([24, 24])
    fig = plt.figure()
    plt.imshow(X, cmap='gray')
    plt.show()
    if y_train[index] in set([1, 3, 5, 7, 9]):
        label = 'Odd'
    else:
        label = 'Even'
    print('Label is', label)
  ✓ 0.1s
```




```
Label is Odd
```

b)

```
# Compute the forward pass
scores = None

### ========== TODO : START ========== ###
#   Calculate the output of the neural network using forward pass.
#   The expected result should be a matrix of shape (N, C), where:
#     - N is the number of examples in the input dataset 'X'.
#     - C is the number of classes.
#   Use 'h1' as the first hidden layer output
#   Apply the ReLU activation function to 'h1' to get 'a1'. Use np.maximum for ReLU implementation.
#   The output 'scores' is the result of the second layer (before applying softmax).
#   Refer to the model architecture comments at the beginning of this class for more details.
#   Note: Do not use a for loop in your implementation.
##  Part (b): Implement the forward pass and compute scores.

### ========== TODO : END ========== ###
h1 = np.dot(X,W1)+b1
#relu functionality:
a1 = np.maximum(0,h1)
h2 = np.dot(a1,W2)+b2
scores = h2
```

Scores were calculated using the dot product and the sum of the weights and biases, and a relu functionality was added for the second layer.

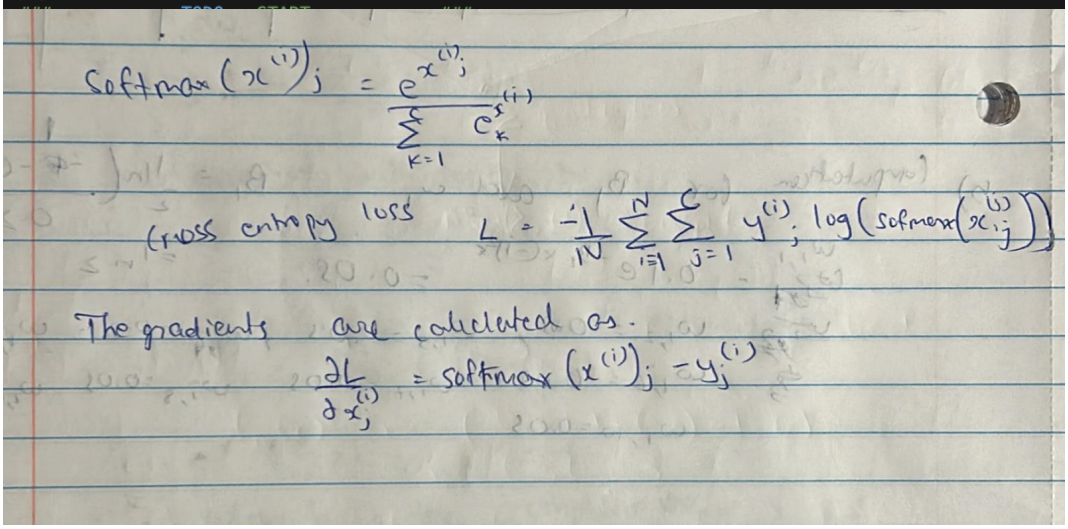
c) The formula for l2 regularization is 0.5 lambda * sum of squares of W1 and W2
In the code it looks like this, calculated using np.sum:

```
#   loss by 0.5 (in addition to t  (variable) scores: Any   'reg').
##  Part (c): Implement the regul
data_loss, dscore = softmax_loss(scores, y)  # Capture data_loss here
reg_loss = 0.5 * reg * (np.sum(W1 * W1) + np.sum(W2 * W2))
loss = data_loss + reg_loss
### ========== TODO : END ========== ###
grads = {}
```

d)
The following is the code for the calculation of the loss using cross entropy and then calculation the dx:

```
def softmax_loss(x, y):
    ### ========== TODO : START ========== ###
    #   Calculate the cross entropy loss after softmax output layer.
    #   This function should return loss and dx
    probs = np.exp(x - np.max(x, axis=1, keepdims=True)) # Other Notes: this operation is called
    probs /= np.sum(probs, axis=1, keepdims=True)
    N = x.shape[0]
    ##  Part (d): Implement the CrossEntropyLoss
    Cross_entropy_loss = -np.log(probs[range(N), y])
    loss = np.sum(Cross_entropy_loss) / N
    ##  Part (d): Implement the gradient of y wrt x
    dx = probs
    for i in range(N):
      dx[i, y[i]] -= 1
    dx /= N

    ### ========== TODO : END ========== ###
    return loss, dx
```



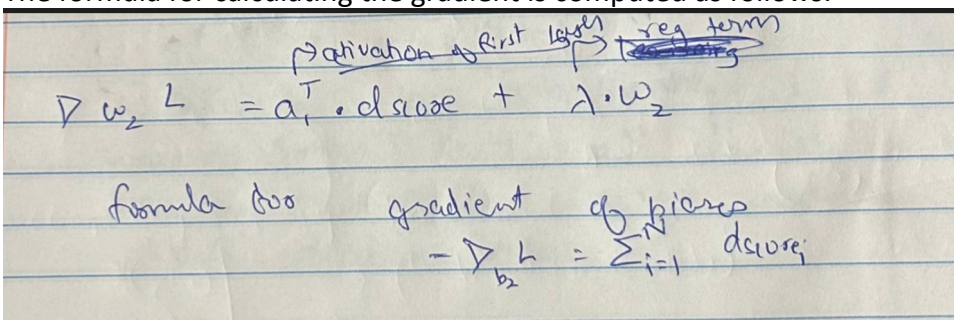
e)

```
### ========== TODO : START ========== ###
#  Compute backpropagation
#  Remember the loss contains two parts: cross-entropy and regularizatio
## Part (e): Implement the computations of gradients for W2 and b2.
grads['W2'] = np.dot(a1.T, dscore) + reg * W2
grads['b2'] = np.sum(dscore, axis=0)

dh = np.dot(dscore,W2.T)
dh[a1 <= 0] = 0

grads['W1'] = np.dot(X.T, dh) + reg * W1
grads['b1'] = np.ones(N).dot(dh)
### ========== TODO : END ========== ###
```

The formula for calculating the gradient is computed as follows:



f)

This is the prediction function:

```
### ========== TODO : START ========== ###
#   Predict the class given the input data.
##  Part (f): Implement the prediction function
h1 = np.dot(X, self.params['W1']) + self.params['b1']
a1 = np.maximum(0, h1)  # ReLU activation
h2 = np.dot(a1, self.params['W2']) + self.params['b2']
scores = h2
y_pred = np.argmax(scores, axis=1)

### ========== TODO : END ========== ###

return y_pred
```

Where we basically compute h1 and h2 using weights and biases, apply relu, and find the maximum score of the class.

These are the outputs: learning_rate: 1e-05
iteration 0 / 1000: loss 0.6931759326402716
iteration 100 / 1000: loss 0.6931502466025943
iteration 200 / 1000: loss 0.6931252794409479
iteration 300 / 1000: loss 0.6930902295940785
iteration 400 / 1000: loss 0.6930770120282669
iteration 500 / 1000: loss 0.6929992486397712
iteration 600 / 1000: loss 0.6929411480420469
iteration 700 / 1000: loss 0.6927631937195603
iteration 800 / 1000: loss 0.6926740086008922
iteration 900 / 1000: loss 0.6924921096884873
Validation accuracy: 0.8068

Test accuracy (subopt_net): 0.7989

learning_rate: 0.0001
iteration 0 / 1000: loss 0.6920428985498048
iteration 100 / 1000: loss 0.6503433220833782
iteration 200 / 1000: loss 0.4387764613212066
iteration 300 / 1000: loss 0.34755405537099954
iteration 400 / 1000: loss 0.3668200537904054
iteration 500 / 1000: loss 0.30283246741684405
iteration 600 / 1000: loss 0.2696282423492896
iteration 700 / 1000: loss 0.34346681500043
iteration 800 / 1000: loss 0.29475040321162094
iteration 900 / 1000: loss 0.32297468156263426
Validation accuracy: 0.8849
Test accuracy (subopt_net): 0.8801

learning_rate: 0.001
iteration 0 / 1000: loss 0.26273629117366826
iteration 100 / 1000: loss 0.2923635955227758
iteration 200 / 1000: loss 0.24384353228258993
iteration 300 / 1000: loss 0.14599404232328975
iteration 400 / 1000: loss 0.14306520591886046
iteration 500 / 1000: loss 0.15103641936448556
iteration 600 / 1000: loss 0.07223334785904566
iteration 700 / 1000: loss 0.1606541066317776
iteration 800 / 1000: loss 0.11532390755488447
iteration 900 / 1000: loss 0.09200337381521209
Validation accuracy: 0.9742
Test accuracy (subopt_net): 0.9709

learning_rate: 0.005
iteration 0 / 1000: loss 0.08490968526795019
iteration 100 / 1000: loss 0.6845211908534286
iteration 200 / 1000: loss 0.6323266778586386
iteration 300 / 1000: loss 0.49999738640153946
iteration 400 / 1000: loss 0.5167332950501832
iteration 500 / 1000: loss 0.45931160724845455
iteration 600 / 1000: loss 0.5690268211342785
iteration 700 / 1000: loss 0.45044230813970154
iteration 800 / 1000: loss 0.465993404400481
iteration 900 / 1000: loss 0.4140829858573113
Validation accuracy: 0.896
Test accuracy (subopt_net): 0.8921

learning_rate: 0.1
iteration 0 / 1000: loss 0.4848716772603494
iteration 100 / 1000: loss 0.7015175168757546
iteration 200 / 1000: loss 0.6942329713623974
iteration 300 / 1000: loss 0.6924889927684718
iteration 400 / 1000: loss 0.6922157424632742
iteration 500 / 1000: loss 0.6920050478969361
iteration 600 / 1000: loss 0.6935384788088882
iteration 700 / 1000: loss 0.6916083338944105
iteration 800 / 1000: loss 0.6929875313446917

iteration 900 / 1000: loss 0.6895275914682379
Validation accuracy: 0.506
Test accuracy (subopt_net): 0.5074

The best validation accuracy is for learning rate = 0.001, which gives a validation accuracy as 0.9742 and Test accuracy as 0.9709, which is the best after a thousand iterations of different learning rates. For the initial lower learning rates, the learning was too low and thus wasn't able to give the best scores for 1000 iterations, and the ones greater were overshooting the values.

Problem 4:
a)

```
## simple utility function to visualize the data
def visualize(X, ind):
    from PIL import Image
    plt.imshow(Image.fromarray(X[ind], 'RGB'))
```

✓ 0.0s

```
X, y = dataloader()
```

✓ 4.7s

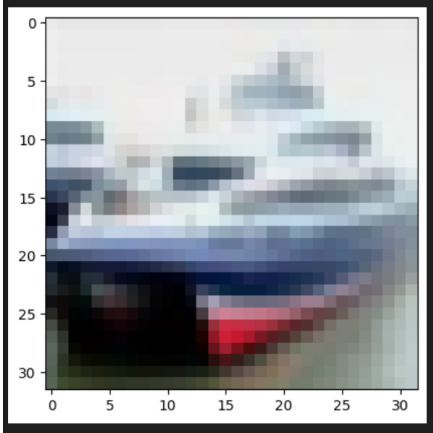
```
# 10K images of size 32 x 32 x 3
# where 32 x 32 is the height and width of the image
# 3 is the number of channels 'RGB'
X.shape, y.shape
```

✓ 0.0s

```
((10000, 32, 32, 3), (10000, 1))
```

```
visualize(X, 1)
```

✓ 0.1s




```
'''

print(X.shape)
def reshape(X):
    N = 32*32*3
    '''
    Write one line of code here
    '''
    ### ========== TODO : START ========== ###
    # part (a)
    return X.reshape(10000,N)

    ### ========== TODO : END ========== ###
```

✓ 0.0s

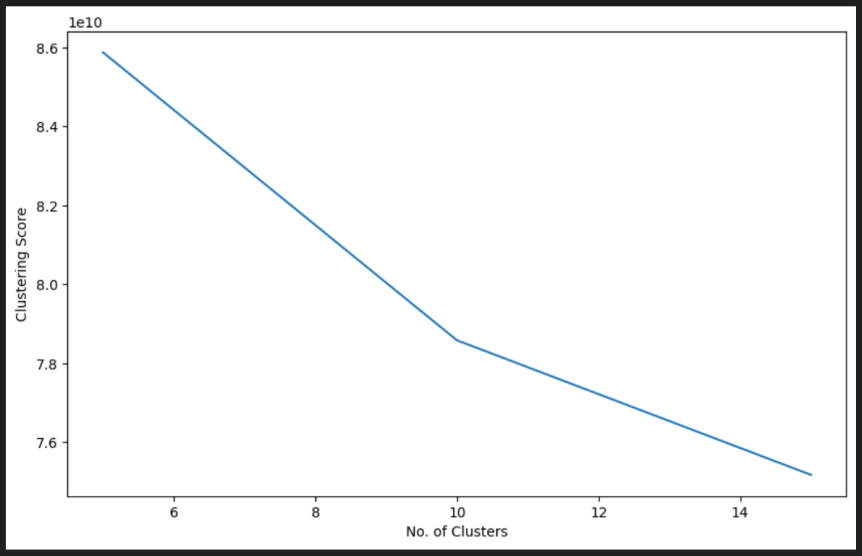
b)

```python
clustering_score = []
for i in tqdm(range(5, 20, 5)):
  score = 0
  for rs in tqdm(range(3)):
    kmeans = KMeans(n_clusters = i, init = 'random', random_state = rs)
    '''
      Write one line of code to fit the kMeans algorithm to the data
      Write another line of code to report the kMeans clustering score
      defined as sum of squared distances of samples to their closest
      cluster center, weighted by the sample weights if provided.
      Hint: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html
    '''
    ### ========== TODO : START ========== ###
    # part (b)
    kmeans.fit(X)
    score += kmeans.inertia_

    ### ========== TODO : END ========== ###
  clustering_score.append(score/3) ## divide by 3 because 3 random states
```

✓ 0.2s

```
[85874000093.78233, 78584477596.15273, 75179173390.41476]
```



c)

```
from sklearn.manifold import TSNE


# Assume X is your dataset
kmeans = KMeans(n_clusters=10, init='random', random_state=42)
tsne = TSNE(n_components=2, verbose=1, perplexity=40, n_iter=300)
X_tsne = tsne.fit_transform(X)

labels = kmeans.fit_predict(X_tsne)

u_labels = np.unique(labels)

for i in u_labels:
    plt.scatter(X_tsne[labels == i][:, 0], X_tsne[labels == i][:, 1], label=f'Cluster {i}')

plt.legend()
plt.show()
```

✓ 53.8s

here is the plot obtained from the code snippet above:

```
[t-SNE] Computing 121 nearest neighbors...
[t-SNE] Indexed 10000 samples in 0.010s...
[t-SNE] Computed neighbors for 10000 samples in 9.467s...
[t-SNE] Computed conditional probabilities for sample 1000 / 10000
[t-SNE] Computed conditional probabilities for sample 2000 / 10000
[t-SNE] Computed conditional probabilities for sample 3000 / 10000
[t-SNE] Computed conditional probabilities for sample 4000 / 10000
[t-SNE] Computed conditional probabilities for sample 5000 / 10000
[t-SNE] Computed conditional probabilities for sample 6000 / 10000
[t-SNE] Computed conditional probabilities for sample 7000 / 10000
[t-SNE] Computed conditional probabilities for sample 8000 / 10000
[t-SNE] Computed conditional probabilities for sample 9000 / 10000
[t-SNE] Computed conditional probabilities for sample 10000 / 10000
[t-SNE] Mean sigma: 751.010321
[t-SNE] KL divergence after 250 iterations with early exaggeration: 87.631859
[t-SNE] KL divergence after 300 iterations: 3.341745
/opt/homebrew/lib/python3.11/site-packages/sklearn/cluster/_kmeans.py:1416: FutureWarni
  super()._check_params_vs_input(X, default_n_init=10)
```