# CS 671 Homework 4

Yutong Shao
NetID ys392

December 9, 2022

**I consent to the following agreements.**

*This assignment represents my own work. I did not work on this assignment with others. All coding was done by myself.*

*I understand that if I struggle with this assignment that I will reevaluate whether this is the correct class for me to take. I understand that the homework only gets harder.*

## 1 VC dimension of Binary Decision Trees with Fixed Split Points

*Proof.* Proof by induction.

If $\mathcal{F} :=$ {the set of all binary decision trees whose split points are exactly equal to $\mathcal{X}$ with exactly 2 leaves}, then this class of trees can at most classify 2 points, because we can write four labeling assignments for this decision tree: $(1, 1), (1, -1), (-1, 1), (-1, -1)$.

But three points can not be shattered by this class of trees. We can prove this by contradiction. Assume we can use $F \in \mathcal{F}$ to shatter three points, then there must exist two points that are assigned the same label. However, these two data points are not necessarily in the same class, which contradicts with the assumption.

By induction, for $\mathcal{F} :=$ {the set of all binary decision trees whose split points are exactly equal to $\mathcal{X}$ with exactly $\ell$ leaves}, then this class of trees can at most classify $\ell$ points. And Similarly, we can prove by contradiction that $\ell + 1$ points cannot be perfectly classified.

Therefore, $VC(\mathcal{F}) = \ell$.

□

# 2 Topic Modeling with EM

## 2.1 Derive Log-Likelihood

$\because$ Word $n$ are drawn i.i.d. from topic $k$

$\therefore$ The total probability that word $n$ appear in document $i$ is

$$\text{likelihood}(\theta) = \prod_{q=1}^{Q} \sum_{k=1}^{K} \Pr(w_n|z_k)\Pr(z_k|d_i)$$

where Q is the total number of words, $Q = \sum_{i=1}^{M} \sum_{n=1}^{N} q(w_n; d_i)$.

$\therefore$ The log likelihood is

$$\begin{aligned}
\log \text{likelihood}(\theta) &= \sum_{i=1}^{M} \sum_{n=1}^{N} q(w_n; d_i) \log \sum_{k=1}^{K} \Pr(w_n|z_k)\Pr(z_k|d_i) \\
&= \sum_{i=1}^{M} \sum_{n=1}^{N} q(w_n; d_i) \log \sum_{k=1}^{K} \beta_{kn}\alpha_{ik}
\end{aligned}$$

## 2.2 E Step

$$\begin{aligned}
p(z_k|d_i, w_n, \alpha^{old}, \beta^{old}) &= \frac{\Pr(w_n|z_k, d_i, \alpha^{old}, \beta^{old})\Pr(z_k|d_i, \alpha^{old}, \beta^{old})}{\Pr(w_n|\alpha^{old}, \beta^{old})} \\
&= \frac{\alpha^{old}\beta^{old}}{\sum_{k=1}^{K} \alpha^{old}\beta^{old}}
\end{aligned}$$

## 2.3 Find ELBO for M-Step

$$\begin{aligned}
\log \text{likelihood}(\theta) &= \sum_{i=1}^{M} \sum_{n=1}^{N} q(w_n; d_i) \log \sum_{k=1}^{K} \Pr(w_n|z_k)\Pr(z_k|d_i) \\
&= \sum_{i=1}^{M} \sum_{n=1}^{N} q(w_n; d_i) \log \sum_{k=1}^{K} \gamma_{ink} \frac{\beta_{kn}\alpha_{ik}}{\gamma_{ink}} \\
&\geq \sum_{i=1}^{M} \sum_{n=1}^{N} \sum_{k=1}^{K} q(w_n; d_i)\gamma_{ink} \log \frac{\beta_{kn}\alpha_{ik}}{\gamma_{ink}} \qquad \leftarrow \text{Jensen's inequality} \\
&:= A(\alpha, \beta)
\end{aligned}$$

### 2.4 M Step

Define

$$A'(\alpha, \beta) := \sum_{i=1}^{M} \sum_{n=1}^{N} \sum_{k=1}^{K} q(w_n; d_i) \gamma_{ink} \log \beta_{kn} \alpha_{ik}$$

since when maximizing auxiliary function, we can only focus on numerator.

Therefore, the constrainted maximization problem is

$$\begin{aligned} &\max_{\alpha,\beta} A'(\alpha, \beta) \\ s.t. \quad &\sum_{k=1}^{K} \alpha_{ik} = 1 \\ &\sum_{n=1}^{N} \beta_{kn} = 1 \end{aligned}$$

Write Lagrangian as

$$\mathcal{L}(\alpha, \beta) = A'(\alpha, \beta) + \lambda_1 \left(1 - \sum_{k=1}^{K} \alpha_{ik}\right) + \lambda_2 \left(1 - \sum_{n=1}^{N} \beta_{kn}\right)$$

Compute derivatives and set to 0.

$$\frac{\partial \mathcal{L}}{\partial \alpha_{ik}} = 0 \tag{1}$$

$$\frac{\partial \mathcal{L}}{\partial \beta_{kn}} = 0 \tag{2}$$

For derivative (1),

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \alpha_{ik}} &= \frac{\partial A'(\alpha, \beta)}{\partial \alpha_{ik}} - \lambda_1 \\ &= \frac{\partial}{\partial \alpha_{ik}} \sum_{i=1}^{M} \sum_{n=1}^{N} \sum_{k=1}^{K} q(w_n; d_i) \gamma_{ink} \log \beta_{kn} \alpha_{ik} - \lambda_1 \\ &= \frac{\partial}{\partial \alpha_{ik}} \sum_{i=1}^{M} \sum_{n=1}^{N} \sum_{k=1}^{K} q(w_n; d_i) \gamma_{ink} \log (\beta_{kn} + \alpha_{ik}) - \lambda_1 \\ &= \underbrace{\frac{\partial}{\partial \alpha_{ik}}}_{\text{derivative of certain i and k}} \sum_{i=1}^{M} \sum_{n=1}^{N} \sum_{k=1}^{K} q(w_n; d_i) \gamma_{ink} \log \alpha_{ik} - \lambda_1 \\ &= \sum_{n=1}^{N} q(w_n; d_i) \gamma_{ink} \frac{1}{\alpha_{ik}} - \lambda_1 = 0 \\ \Longrightarrow \alpha^{new} &= \frac{\sum_{n=1}^{N} q(w_n; d_i) \gamma_{ink}}{\lambda_1} \end{aligned}$$

Note that $\sum_{k=1}^{K} \alpha_{ik} = 1$. Thus,

$$\begin{aligned}
\sum_{k=1}^{K} \alpha_{ik} &= \sum_{k=1}^{K} \frac{\sum_{n=1}^{N} q(w_n; d_i)\gamma_{ink}}{\lambda_1} = 1 \\
\implies \lambda_1 &= \sum_{k=1}^{K} \sum_{n=1}^{N} q(w_n; d_i)\gamma_{ink} \\
&= \sum_{n=1}^{N} q(w_n; d_i) \underbrace{\sum_{k=1}^{K} \gamma_{ink}}_{=1} \\
&= \sum_{n=1}^{N} q(w_n; d_i)
\end{aligned}$$

Therefore,

$$\alpha^{new} = \frac{\sum_{n=1}^{N} q(w_n; d_i)\gamma_{ink}}{\sum_{n=1}^{N} q(w_n; d_i)}$$

We can also derive $\beta^{new}$ follow the same logic. According to (2),

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \beta_{kn}} &= \underbrace{\frac{\partial}{\partial \beta_{kn}}}_{\text{derivative of certain k and n}} \sum_{i=1}^{M} \sum_{n=1}^{N} \sum_{k=1}^{K} q(w_n; d_i)\gamma_{ink} \log \beta_{kn} - \lambda_2 \\
&= \sum_{i=1}^{M} q(w_n; d_i)\gamma_{ink} \frac{1}{\beta_{kn}} - \lambda_2 = 0 \\
\implies \beta^{new} &= \frac{\sum_{i=1}^{M} q(w_n; d_i)\gamma_{ink}}{\lambda_2}
\end{aligned}$$

Note that $\sum_{n=1}^{N} \beta_{kn} = 1$. Thus,

$$\begin{aligned}
\sum_{n=1}^{N} \beta_{kn} &= \sum_{n=1}^{N} \frac{\sum_{i=1}^{M} q(w_n; d_i)\gamma_{ink}}{\lambda_2} = 1 \\
\implies \lambda_2 &= \sum_{i=1}^{M} \sum_{n=1}^{N} q(w_n; d_i)\gamma_{ink}
\end{aligned}$$

Therefore,

$$\beta^{new} = \frac{\sum_{i=1}^{M} q(w_n; d_i)\gamma_{ink}}{\sum_{i=1}^{M} \sum_{n=1}^{N} q(w_n; d_i)\gamma_{ink}}$$

# 3 Gradient computations in Neural Networks

## 3.1 Gradient Evaluation

Define the NN's structure as follows.

Input layer ($l = 0$) - hidden layer 1($l = 1$) - hidden layer 2($l = 2$) - Output Layer($l = 3$)

Before diving into derivations, I want to change some notations to make the vector and matrix representation clearer.

Let $\mathbf{X}$ be the input vector, $\mathbf{X} = (x_1, x_2, \ldots, x_d)^T_{(d\times 1)}$;

Let $\mathbf{W}^{(l)}$ be the weight matrix of layer $l$. Specifically, $\mathbf{W}^{(1)}$ should be a $d \times H$ matrix, $\mathbf{W}^{(2)}$ is $H \times H$, $\mathbf{W}^{(3)}$ should be $H \times 1$. For example,

$$\mathbf{W}^{(2)} = \begin{pmatrix} w_{11}^{(2)} & w_{11}^{(2)} & \cdots & w_{1H}^{(2)} \\ w_{21}^{(2)} & w_{21}^{(2)} & \cdots & w_{2H}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{H1}^{(2)} & w_{H1}^{(2)} & \cdots & w_{HH}^{(2)} \end{pmatrix}$$

Let $\mathbf{b}^{(l)}$ be the bias vector of layer $l$, which should be $H \times 1$ for layer 1 and 2, and a scalar for $l = 3$;

$$\mathbf{b}^{(l)} = \left(b_1^{(l)}, b_2^{(l)}, \ldots, b_H^{(l)}\right)^T_{(H\times 1)}, \quad l = 1, 2$$
$$\mathbf{b}^{(l)} = b^{(3)}, \quad l = 3$$

Let $\mathbf{Z}^{(l)}$ denote the pre-activation vector at layer $l$,

$$\mathbf{Z}^{(l)} = \mathbf{W}^{(l)T}\mathbf{X} + \mathbf{b}^{(l)}, \quad l = 1$$
$$\mathbf{Z}^{(l)} = \mathbf{W}^{(l)T}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}, \quad l > 1$$

And, $\mathbf{h}^{(l)}$ denote the activation function vector of layer $l$.

$$\begin{aligned}\mathbf{h}^{(l)} = \sigma\left(\mathbf{Z}^{(l)}\right) &= \left(h_1^{(l)}, h_2^{(l)}, \ldots, h_H^{(l)}\right)^T_{(H\times 1)} \\ &= \left(\sigma\left(z_1^{(l)}\right), \sigma\left(z_2^{(l)}\right), \ldots, \sigma\left(z_H^{(l)}\right)\right)^T_{(H\times 1)}\end{aligned}$$

Finally, the derivative of sigmoid function is

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1 - \sigma(z))$$

1. From Output Layer($l = 3$) to hidden layer 2($l = 2$)

Goal: compute $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(3)}}$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(3)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(3)}} \frac{\partial \mathbf{h}^{(3)}}{\partial \mathbf{Z}^{(3)}} \frac{\partial \mathbf{Z}^{(3)}}{\partial \mathbf{W}^{(3)}}$$

where

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(3)}} &= -\frac{y}{\mathbf{h}^{(3)}} + \frac{1-y}{1-\mathbf{h}^{(3)}} \\
\frac{\partial \mathbf{h}^{(3)}}{\partial \mathbf{Z}^{(3)}} &= \mathbf{h}^{(3)} \left(1 - \mathbf{h}^{(3)}\right) \\
\frac{\partial \mathbf{Z}^{(3)}}{\partial \mathbf{W}^{(3)}} &= \left(h_1^{(2)}, h_2^{(2)}, \dots, h_H^{(2)}\right)^T_{(H\times 1)} = \mathbf{h}^{(2)}
\end{aligned}$$

Therefore,

$$\begin{aligned}
&\delta_3 = \mathbf{h}^{(3)} - y \\
\Longrightarrow &\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(3)}} = \left(\mathbf{h}^{(3)} - y\right)\mathbf{h}^{(2)} = \delta_3 \mathbf{h}^{(2)} \qquad \leftarrow \text{a } 1 \times H \text{ vector}
\end{aligned}$$

2. From hidden layer 2 ($l = 2$) to hidden layer 1 ($l = 1$).

Goal: compute $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(2)}}$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(2)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{Z}^{(3)}} \frac{\partial \mathbf{Z}^{(3)}}{\partial \mathbf{h}^{(2)}} \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{Z}^{(2)}} \frac{\partial \mathbf{Z}^{(2)}}{\partial \mathbf{W}^{(2)}}$$

where

$$\delta_2 = \frac{\partial \mathcal{L}}{\partial \mathbf{Z}^{(2)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{Z}^{(3)}} \frac{\partial \mathbf{Z}^{(3)}}{\partial \mathbf{h}^{(2)}} \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{Z}^{(2)}} = \delta_3 \underset{H\times 1}{\mathbf{W}^{(3)}} \underbrace{\mathbf{h}^{(2)}\left(1 - \mathbf{h}^{(2)}\right)}_{1\times H}$$

Thus, $\frac{\partial \mathcal{L}}{\partial \mathbf{Z}^{(3)}} \frac{\partial \mathbf{Z}^{(3)}}{\partial \mathbf{h}^{(2)}} \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{Z}^{(2)}}$ is a $H \times H$ matrix.

Now we left one last term $\frac{\partial \mathbf{Z}^{(2)}}{\partial \mathbf{W}^{(2)}}$, which is a little bit complicated because we need to compute the derivative of a vector over a matrix. This should yield a high-dimension tensor, but note that we can write it as a matrix (See the math below 3.2 for the derivation). Also, we only care about the final product, which should also be a $H \times H$ matrix.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(2)}} = \delta_3 \underset{H\times 1}{\mathbf{W}^{(3)}} \underbrace{\mathbf{h}^{(2)}\left(1 - \mathbf{h}^{(2)}\right)}_{1\times H} \mathbf{h}^{(1)T}$$

3. From hidden layer 1 ($l = 1$) to the input layer ($l = 0$)

Goal: compute $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(1)}}$

Similarly, we can get the derivative follow the logic above. The final answer should be a $H \times H$ matrix.

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(1)}} &= \frac{\partial \mathcal{L}}{\partial \mathbf{Z}^{(1)}} \frac{\partial \mathbf{Z}^{(1)}}{\partial \mathbf{W}^{(1)}} \\ &= \delta_2 \underbrace{\mathbf{W}^{(2)}}_{H\times H} \underbrace{\mathbf{h}^{(1)} \left(1 - \mathbf{h}^{(1)}\right)}_{H\times 1} \underbrace{X^T}_{1\times H}\end{aligned}$$

### 3.2 General Rule

Since $1 \leq l \leq L-1$, we can easily get the general form from the above reasoning.

$$\delta_l = \delta_{l+1} \mathbf{W}_{l+1} \mathbf{h}_l \left(1 - \mathbf{h}_l\right)$$

and $\delta_l^i$ is the i-th element of vector $\delta_l$.

### Derivative of vector over matrix

Without the loss of generality, consider $\mathbf{y} = \mathbf{x}W$, where $\mathbf{y} = [y_1, y_2, \cdots, y_H]$ is a $1 \times H$ vector, and $\mathbf{x} = [x_1, x_2, \cdots, x_d]$ is a $1 \times d$ vector, $W$ is a $d \times H$ matrix.

We want to find $\frac{d\mathbf{y}}{dW}$. Normally, the derivative is a tensor. But we can actually write it out as a matrix.

Rewrite $\mathbf{y}$ as elements so that the j-th element can be represented as

$$y_j = x_1 W_{1,j} + x_2 W_{2,j} + \cdots + x_d W_{d,j}$$

Therefore,

$$\frac{\partial y_j}{\partial W_{i.j}} = x_i$$

Note that the derivative of $y_j$ over other elements is 0. Thus,

$$\frac{d\mathbf{y}}{dW} = \begin{pmatrix} \frac{\partial y_1}{\partial W_{1,1}} & \frac{\partial y_1}{\partial W_{2,1}} & \frac{\partial y_1}{\partial W_{3,1}} & \cdots & \frac{\partial y_1}{\partial W_{d,1}} \\ \frac{\partial y_2}{\partial W_{1,2}} & \frac{\partial y_2}{\partial W_{2,2}} & \frac{\partial y_2}{\partial W_{3,2}} & \cdots & \frac{\partial y_2}{\partial W_{d,2}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_H}{\partial W_{1,H}} & \frac{\partial y_H}{\partial W_{2,H}} & \frac{\partial y_H}{\partial W_{3,H}} & \cdots & \frac{\partial y_H}{\partial W_{d,H}} \end{pmatrix} = \begin{pmatrix} x_1 & x_2 & x_3 & \cdots & x_d \\ x_1 & x_2 & x_3 & \cdots & x_d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_1 & x_2 & x_3 & \cdots & x_d \end{pmatrix}_{H\times d}$$

# 4 Clustering

See coding appendix.

# 5 Convolutional Neural Network on CIFAR-10

See coding appendix.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random
import time

import warnings
warnings.filterwarnings('ignore')
```

# Q4 Clustering

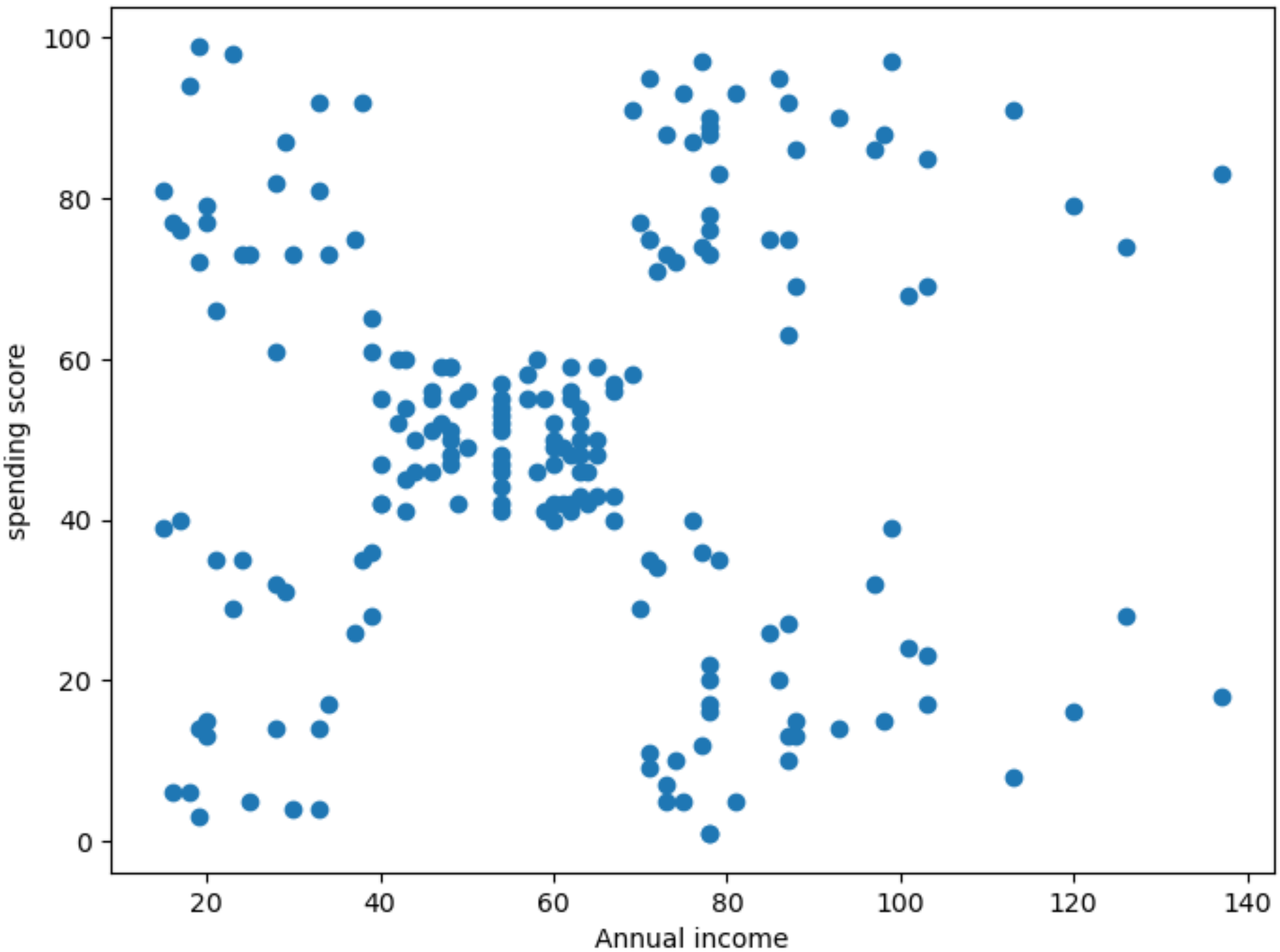
## 4.1 load dataset and draw scatter plot

```python
df5 = pd.read_csv('mall_customers.csv')
df5.head()
```

```css
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

| | Annual_Income | Spending_Score |
|---|---|---|
| **0** | 15 | 39 |
| **1** | 15 | 81 |
| **2** | 16 | 6 |
| **3** | 16 | 77 |
| **4** | 17 | 40 |

```python
fig1, ax1 = plt.subplots(figsize=(8,6))
plt.scatter(df5.iloc[:,0], df5.iloc[:,1]);
plt.xlabel('Annual income')
plt.ylabel('spending score');
```

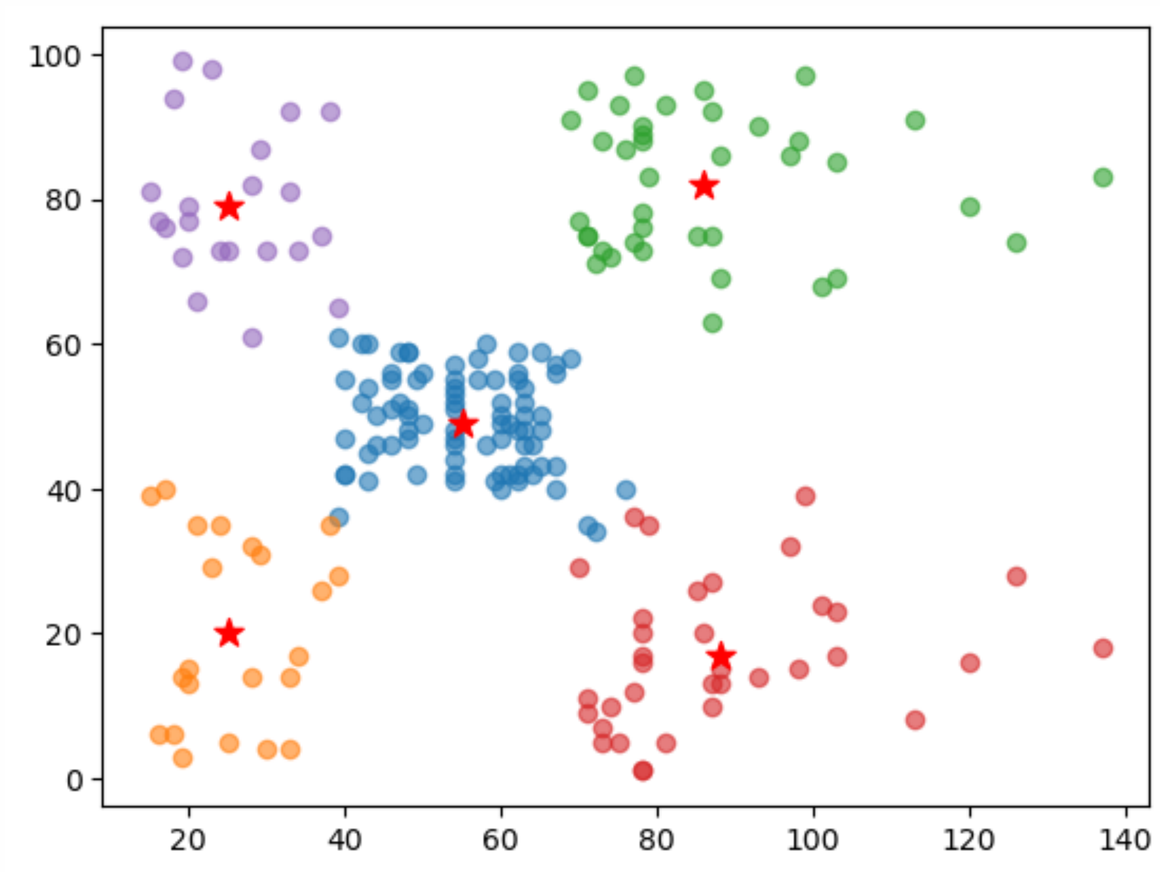



## 4.2 K-means algorithm without sklearn

```python
def EuclideanDistance(data, centers, k):
    '''''''''
    This function calculates the distance between each
    data point and each center, stores in a matrix with
    n rows (n data points) and k columns (k centers)
    '''''''''
    dis_mat = [[0]*k for i in range(data.shape[0])]
    for i in range(data.shape[0]):
```

```
        for j in range(k):
            dis_mat[i][j] = np.sqrt(np.sum((data.iloc[i,:]-centers[j])**2))

    return dis_mat


data = df5
epochs = 10
k = 5
rng = np.random.default_rng()
Cs = rng.choice(data, k)

for e in range(epochs):
    distance = EuclideanDistance(data=df5, centers=Cs, k=5)
    nearest_c = np.argmin(distance, axis=1)
    for new_c in range(k):
        Cs[new_c] = data.loc[nearest_c == new_c].mean()

for i in range(k):
    cluster = data.loc[nearest_c == i]
    X = cluster.iloc[:,0]
    y = cluster.iloc[:,1]
    plt.scatter(X, y, c=colors[i], alpha=0.6)

plt.scatter(Cs[:,0], Cs[:,1], s=100, marker='*',c='r');

print(Cs, '\n', nearest_c)
```

```
[[55 49]
 [25 20]
 [86 82]
 [88 17]
 [25 79]]
 [1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1
 4 1 4 1 4 0 0 1 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 2 3 2 0 2 3 2 3 2 0 2 3 2 3 2 3 2 3 2 0 2 3 2 3 2
 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3
 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2]
```




Each cluster represents:

the upper right cluster: people who have high income and also spend much
the lower right cluster: people who have relatively low income but spend much
the center cluster: people with medium income and medium spending score
the upper left cluster: people with high income but low spending score
the lower left cluster: people with low income and low spending score

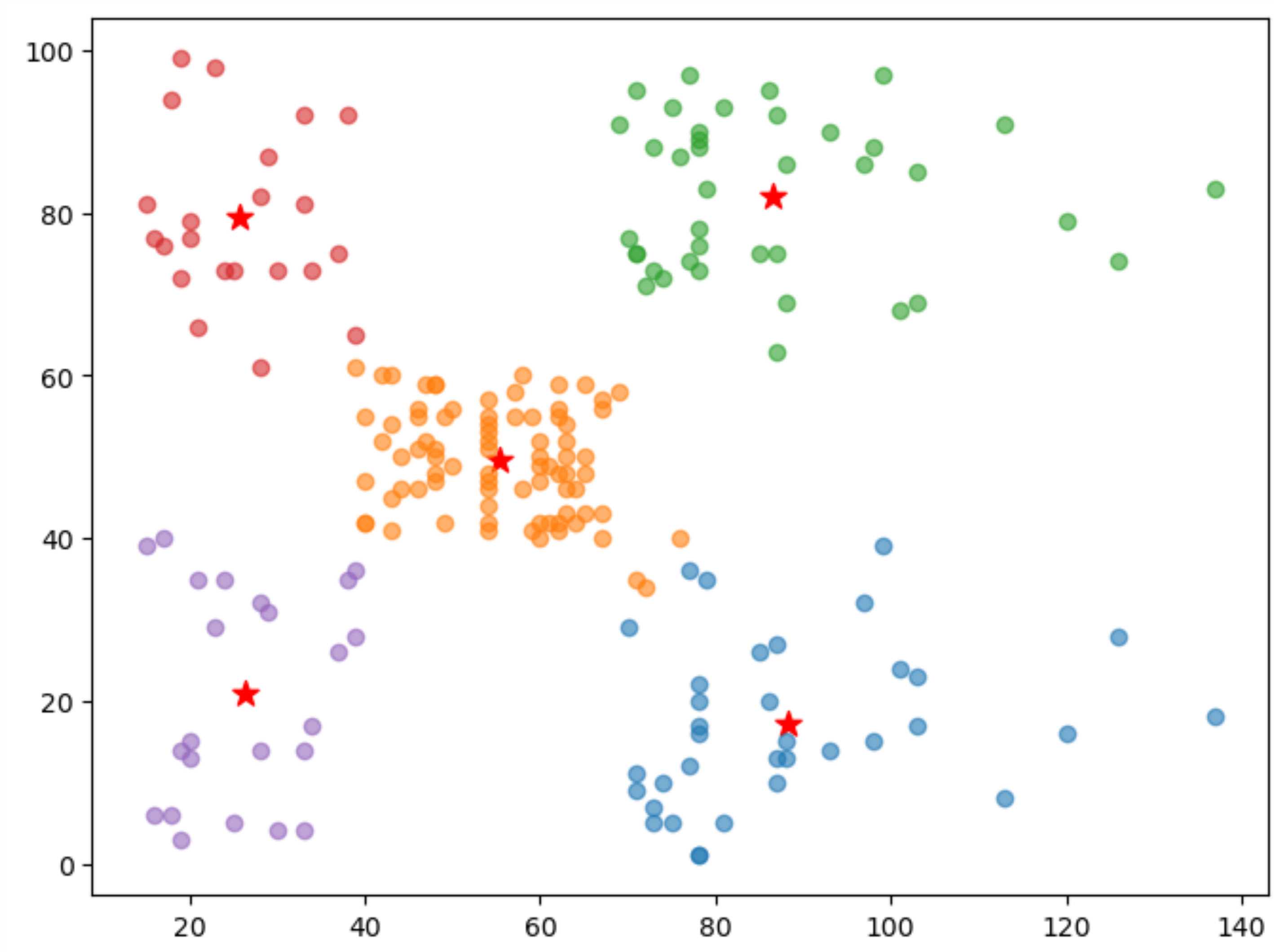
```
# print(Cs, '\n', nearest_c)
def PlotCluster(data, centers, nearest_c, k, colors):
    fig52, ax52 = plt.subplots(figsize=(8,6))
    plt.scatter(centers[:,0], centers[:,1], s=100, marker='*',c='r')
#     color = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd']

    for i in range(k):
        cluster = data.loc[nearest_c == i]
        X = cluster.iloc[:,0]
        y = cluster.iloc[:,1]
        plt.scatter(X, y, c=colors[i], alpha=0.6)

colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd']
# PlotCluster(df5, Cs, nearest_c, 5, colors)
```

## 4.3 K-means with sklearn

```python
from sklearn.cluster import KMeans
```

```python
kmeans = KMeans(n_clusters=5, random_state=0).fit(df5)
centers_sk = kmeans.cluster_centers_
nearest_c_sk = kmeans.labels_
print(centers_sk, '\n', nearest_c_sk)
PlotCluster(df5, centers_sk, nearest_c_sk, 5, colors);

```

```
[[88.2        17.11428571]
 [55.2962963  49.51851852]
 [86.53846154 82.12820513]
 [25.72727273 79.36363636]
 [26.30434783 20.91304348]]
 [4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4
 3 4 3 4 3 4 1 4 3 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 2 0 2 1 2 0 2 0 2 1 2 0 2 0 2 0 2 0 2 1 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0
 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2]
```



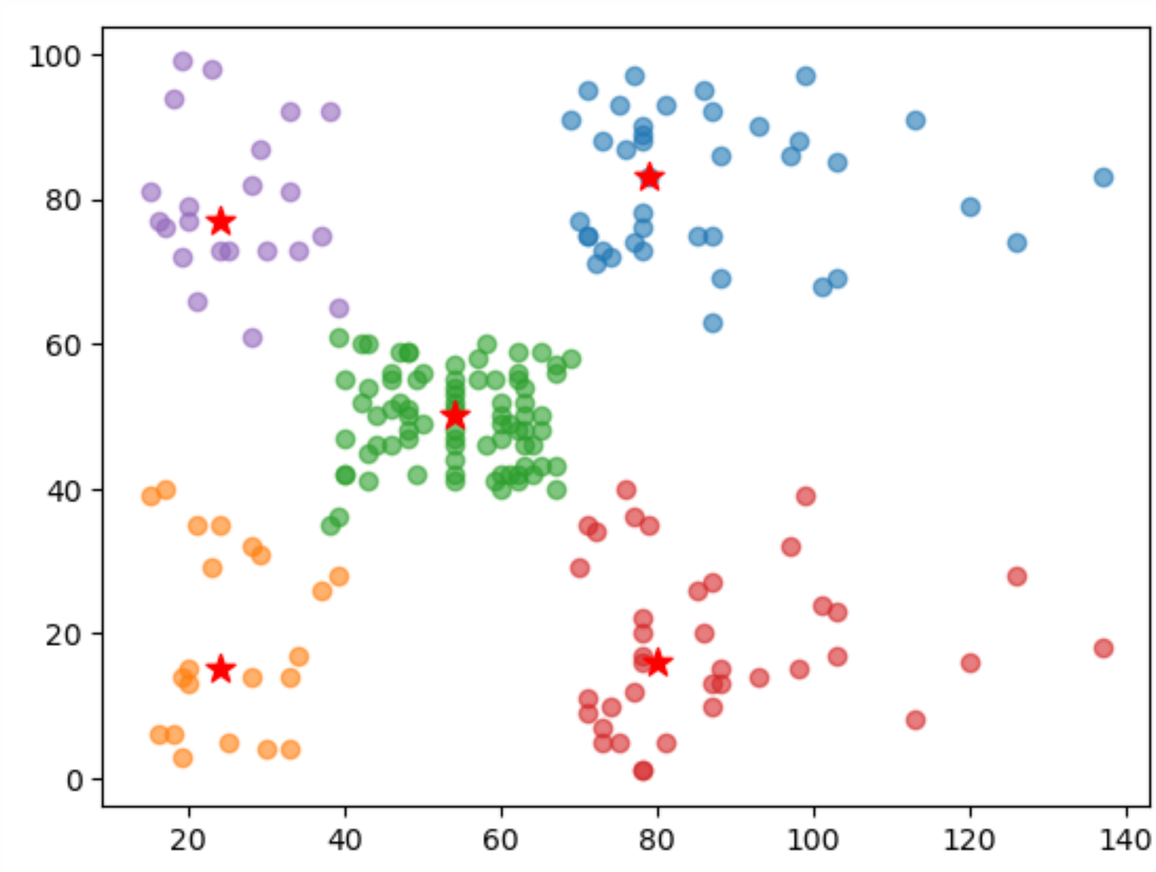

Yes, the results are the same for over 80% of the time. But Kmeans are more stable because the cluster assignment are always the same, but my algorithm has different labels for some iteration. This is probably because the initial centers of my algorithm are randomly chosen at the beginning, which may sometimes influence the final cluster result in the end.

## 4.4 K-medians

```python
def l1normDistance(data, centers, k):
    '''''''''
    This function calculates the l1-norm distance between each
    data point and each center, stores in a matrix with
    n rows (n data points) and k columns (k centers)
    '''''''''
    dis_mat = [[0]*k for i in range(data.shape[0])]

    for i in range(data.shape[0]):
        for j in range(k):
            dis_mat[i][j] = np.sum(np.abs(data.iloc[i,:] - centers[j]))

    return dis_mat
```

```python
data = df5
epochs = 10
k = 5
rng = np.random.default_rng()
Cs = rng.choice(data, k)

for e in range(epochs):
    distance = l1normDistance(data=df5, centers=Cs, k=5)
    nearest_c = np.argmin(distance, axis=1)
    for new_c in range(k):
        Cs[new_c] = data.loc[nearest_c == new_c].median()

for i in range(k):
    cluster = data.loc[nearest_c == i]
    X = cluster.iloc[:,0]
    y = cluster.iloc[:,1]
    plt.scatter(X, y, c=colors[i], alpha=0.6)

plt.scatter(Cs[:,0], Cs[:,1], s=100, marker='*',c='r');

print(Cs, '\n', nearest_c)
```

```
[[79 83]
 [24 15]
 [54 50]
 [80 16]
 [24 77]]
 [1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1 4 1
 4 1 4 2 4 2 2 1 4 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0
 3 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0 3
 0 3 0 3 0 3 0 3 0 3 0 3 0 3 0]
```




The results are not the same when using k-means and k-medians.

```
1
```

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np
from torch.utils.data.dataset import random_split
from torchvision import datasets
from sklearn.metrics import confusion_matrix
import PIL
```

# Q5

```python
##Do Not Touch This Cell

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 8, 5)
        self.conv2 = nn.Conv2d(8, 16, 3)
        self.bn1 = nn.BatchNorm2d(8)
        self.bn2 = nn.BatchNorm2d(16)
        self.fc1 = nn.Linear(16*6*6, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = F.max_pool2d(out, 2)
        out = F.relu(self.bn2(self.conv2(out)))
        out = F.max_pool2d(out, 2)
        out = out.view(out.size(0), -1)
        out = F.relu(self.fc1(out))
        out = F.relu(self.fc2(out))
        out = self.fc3(out)
        return out


```

```python
##Do Not Touch This Cell

device = 'cuda' if torch.cuda.is_available() else 'cpu'
net = Net().to(device)
optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.5)
if device =='cuda':
    print("Train on GPU...")
else:
    print("Train on CPU...")
```

```
Train on GPU...
```

```python
##Do Not Touch This Cell
max_epochs = 50

random_seed = 671
torch.manual_seed(random_seed)
```

```
<torch._C.Generator at 0x7f6528454170>
```

```python
train_transform = transforms.Compose(
     [transforms.ToTensor(),
     transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))])

test_transform = transforms.Compose(
     [transforms.ToTensor(),
     transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))])

dataset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True,
                                       transform=train_transform)
##TODO: Split the set into 80% train, 20% validation (there are 50K total images)
train_num = int(0.8 * 50000)
val_num = int(0.2 * 50000)
train_set, val_set = random_split(dataset, [train_num, val_num])

train_loader = torch.utils.data.DataLoader(train_set, batch_size=128, shuffle=True)
val_loader = torch.utils.data.DataLoader(val_set, batch_size=128, shuffle=False)

test_set = torchvision.datasets.CIFAR10(root='./data', train=False,
                                        download=True, transform=test_transform)
```

```
test_loader = torch.utils.data.DataLoader(test_set, batch_size=1, shuffle=False)

classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog',
           'frog', 'horse', 'ship', 'truck']
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
```

```
  0%|          | 0/170498071 [00:00<?, ?it/s]
```

```
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
```

```
len(train_set)
```

```
40000
```

```
loss_list, acc_list = [], []
loss_list_val, acc_list_val = [], []
criterion = nn.CrossEntropyLoss()

for epoch in range(max_epochs):
    #TODO: set the net to train mode:
    model_t = net.train()

    epoch_loss = 0.0
    correct = 0
    for batch_idx, (data, labels) in enumerate(train_loader):
        data, labels = data.to(device), labels.to(device)

        optimizer.zero_grad()
        ##TODO: pass the data into the network and store the output
        output = model_t(data)

        ##TODO: Calculate the cross entropy loss between the output and target
        loss = criterion(output, labels)
```

```

        ##TODO: Perform backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        ##TODO: Get the prediction from the output
        _, predicted = torch.max(output.data, 1)

        ##TODO: Calculate the correct number and add the number to correct
        correct += (predicted == labels).sum()

        ##TODO: Add the loss to epoch_loss.
        epoch_loss += loss

    ##TODO: calculate the average loss
    avg_loss = epoch_loss / len(train_set)

    ##TODO: calculate the average accuracy
    avg_acc = correct / len(train_set)

    ##TODO: append average epoch loss to loss list
    loss_list.append(avg_loss)

    ##TODO: append average accuracy to accuracy list
    acc_list.append(avg_acc)



    # validation
    ##TODO: set the model to eval mode
    model_val = net.eval()

    with torch.no_grad():
        loss_val = 0.0
        correct_val = 0
        for batch_idx, (data, labels) in enumerate(val_loader):
            data, labels = data.to(device), labels.to(device)
            ##TODO: pass the data into the network and store the output
            output_val = model_val(data)

            ##TODO: Calculate the cross entropy loss between the output and target
            loss_val_ = criterion(output_val, labels)

            ##TODO: Get the prediction from the output
            _, predicted_val = torch.max(output_val.data, 1)

            ##TODO: Calculate the correct number and add the number to correct_val
            correct_val += (predicted_val == labels).sum()
```

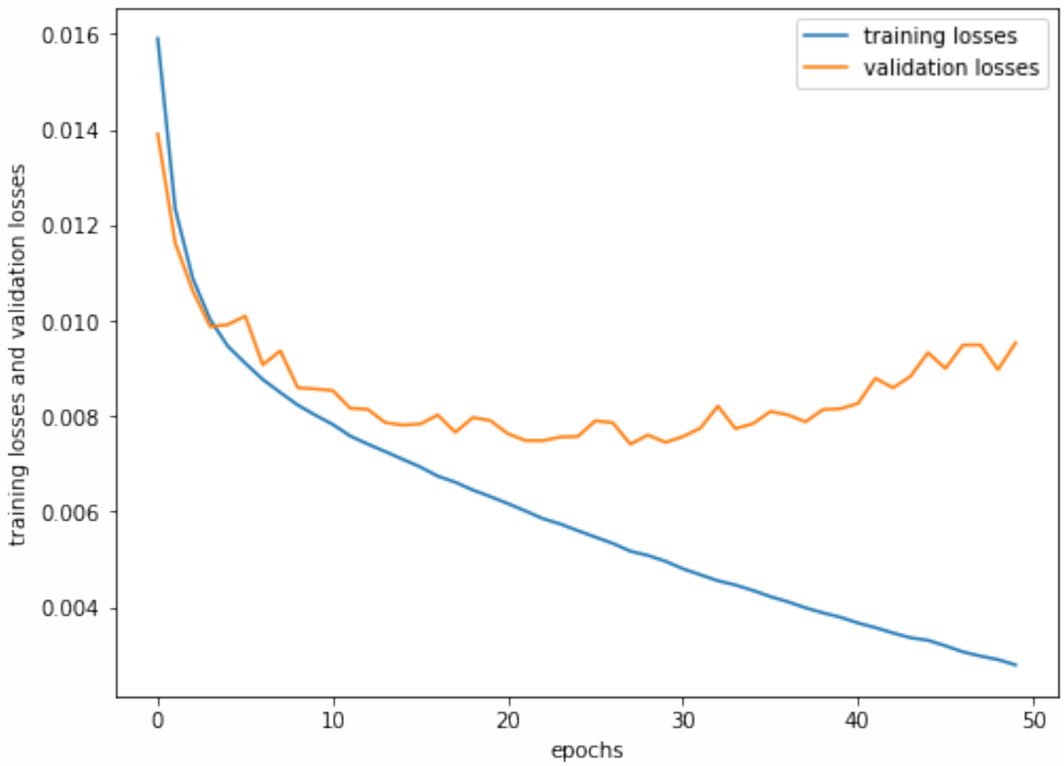
```python

            ##TODO: Add the loss to loss_val
            loss_val += loss_val_

        ##TODO: calculate the average loss of validation
        avg_loss_val = loss_val / len(test_set)

        ##TODO: calculate the average accuracy of validation
        avg_acc_val = correct_val / len(test_set)

        ##TODO: append average epoch loss to loss list of validation
        loss_list_val.append(avg_loss_val)

        ##TODO: append average accuracy to accuracy list of validation
        acc_list_val.append(avg_acc_val)

    print('[epoch %d] loss: %.5f accuracy: %.4f val loss: %.5f val accuracy: %.4f' %
(epoch + 1, avg_loss, avg_acc, avg_loss_val, avg_acc_val))
```

```
[epoch 1] loss: 0.01591 accuracy: 0.2610 val loss: 0.01390 val accuracy: 0.3579
[epoch 2] loss: 0.01234 accuracy: 0.4176 val loss: 0.01163 val accuracy: 0.4683
[epoch 3] loss: 0.01088 accuracy: 0.4937 val loss: 0.01062 val accuracy: 0.5151
[epoch 4] loss: 0.01002 accuracy: 0.5396 val loss: 0.00987 val accuracy: 0.5547
[epoch 5] loss: 0.00947 accuracy: 0.5695 val loss: 0.00991 val accuracy: 0.5523
[epoch 6] loss: 0.00911 accuracy: 0.5879 val loss: 0.01009 val accuracy: 0.5614
[epoch 7] loss: 0.00877 accuracy: 0.6035 val loss: 0.00908 val accuracy: 0.5976
[epoch 8] loss: 0.00850 accuracy: 0.6174 val loss: 0.00937 val accuracy: 0.5924
[epoch 9] loss: 0.00823 accuracy: 0.6284 val loss: 0.00859 val accuracy: 0.6161
[epoch 10] loss: 0.00802 accuracy: 0.6374 val loss: 0.00857 val accuracy: 0.6217
[epoch 11] loss: 0.00783 accuracy: 0.6459 val loss: 0.00854 val accuracy: 0.6227
[epoch 12] loss: 0.00758 accuracy: 0.6597 val loss: 0.00817 val accuracy: 0.6356
[epoch 13] loss: 0.00742 accuracy: 0.6677 val loss: 0.00814 val accuracy: 0.6411
[epoch 14] loss: 0.00726 accuracy: 0.6743 val loss: 0.00786 val accuracy: 0.6557
[epoch 15] loss: 0.00710 accuracy: 0.6805 val loss: 0.00781 val accuracy: 0.6534
[epoch 16] loss: 0.00694 accuracy: 0.6892 val loss: 0.00784 val accuracy: 0.6517
[epoch 17] loss: 0.00674 accuracy: 0.6996 val loss: 0.00803 val accuracy: 0.6459
[epoch 18] loss: 0.00662 accuracy: 0.7035 val loss: 0.00766 val accuracy: 0.6635
[epoch 19] loss: 0.00646 accuracy: 0.7102 val loss: 0.00797 val accuracy: 0.6535
[epoch 20] loss: 0.00632 accuracy: 0.7159 val loss: 0.00791 val accuracy: 0.6545
[epoch 21] loss: 0.00617 accuracy: 0.7240 val loss: 0.00763 val accuracy: 0.6642
[epoch 22] loss: 0.00602 accuracy: 0.7325 val loss: 0.00749 val accuracy: 0.6730
[epoch 23] loss: 0.00586 accuracy: 0.7392 val loss: 0.00749 val accuracy: 0.6749
[epoch 24] loss: 0.00574 accuracy: 0.7425 val loss: 0.00756 val accuracy: 0.6701
[epoch 25] loss: 0.00560 accuracy: 0.7512 val loss: 0.00757 val accuracy: 0.6788
[epoch 26] loss: 0.00547 accuracy: 0.7553 val loss: 0.00790 val accuracy: 0.6628
[epoch 27] loss: 0.00534 accuracy: 0.7616 val loss: 0.00786 val accuracy: 0.6643
[epoch 28] loss: 0.00517 accuracy: 0.7694 val loss: 0.00741 val accuracy: 0.6842
[epoch 29] loss: 0.00509 accuracy: 0.7736 val loss: 0.00761 val accuracy: 0.6747
```

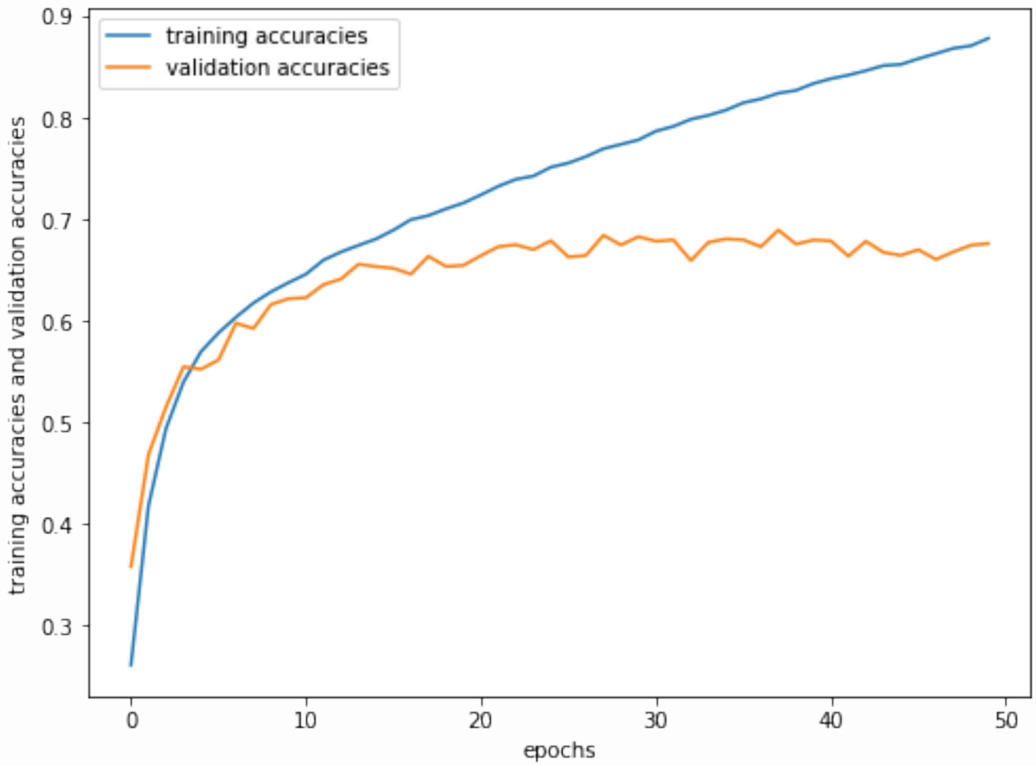
```
[epoch 30] loss: 0.00496 accuracy: 0.7781 val loss: 0.00745 val accuracy: 0.6827
[epoch 31] loss: 0.00481 accuracy: 0.7867 val loss: 0.00757 val accuracy: 0.6784
[epoch 32] loss: 0.00468 accuracy: 0.7914 val loss: 0.00775 val accuracy: 0.6796
[epoch 33] loss: 0.00456 accuracy: 0.7984 val loss: 0.00821 val accuracy: 0.6593
[epoch 34] loss: 0.00447 accuracy: 0.8022 val loss: 0.00774 val accuracy: 0.6772
[epoch 35] loss: 0.00435 accuracy: 0.8074 val loss: 0.00784 val accuracy: 0.6806
[epoch 36] loss: 0.00422 accuracy: 0.8147 val loss: 0.00810 val accuracy: 0.6797
[epoch 37] loss: 0.00412 accuracy: 0.8184 val loss: 0.00803 val accuracy: 0.6729
[epoch 38] loss: 0.00399 accuracy: 0.8241 val loss: 0.00788 val accuracy: 0.6894
[epoch 39] loss: 0.00388 accuracy: 0.8268 val loss: 0.00814 val accuracy: 0.6755
[epoch 40] loss: 0.00379 accuracy: 0.8336 val loss: 0.00815 val accuracy: 0.6795
[epoch 41] loss: 0.00367 accuracy: 0.8383 val loss: 0.00827 val accuracy: 0.6787
[epoch 42] loss: 0.00357 accuracy: 0.8418 val loss: 0.00879 val accuracy: 0.6636
[epoch 43] loss: 0.00346 accuracy: 0.8463 val loss: 0.00859 val accuracy: 0.6783
[epoch 44] loss: 0.00336 accuracy: 0.8512 val loss: 0.00883 val accuracy: 0.6674
[epoch 45] loss: 0.00331 accuracy: 0.8524 val loss: 0.00933 val accuracy: 0.6644
[epoch 46] loss: 0.00319 accuracy: 0.8580 val loss: 0.00900 val accuracy: 0.6700
[epoch 47] loss: 0.00307 accuracy: 0.8630 val loss: 0.00949 val accuracy: 0.6604
[epoch 48] loss: 0.00298 accuracy: 0.8682 val loss: 0.00949 val accuracy: 0.6679
[epoch 49] loss: 0.00291 accuracy: 0.8707 val loss: 0.00897 val accuracy: 0.6744
[epoch 50] loss: 0.00279 accuracy: 0.8781 val loss: 0.00953 val accuracy: 0.6761
```

```python
loss_list1 = [loss_list[i].item() for i in range(len(loss_list))]
loss_list_val1 = [loss_list_val[i].item() for i in range(len(loss_list_val))]
acc_list1 = [acc_list[i].item() for i in range(len(acc_list))]
acc_list_val1 = [acc_list_val[i].item() for i in range(len(acc_list_val))]

```

```python
##TODO: Plot the training losses and validation losses
X = range(max_epochs)
y1 = loss_list1
y2 = loss_list_val1
fig1, ax1 = plt.subplots(figsize=(8,6))
plt.plot(X, y1)
plt.plot(X, y2)
plt.xlabel('epochs')
plt.ylabel('training losses and validation losses')
plt.legend(['training losses', 'validation losses']);

```




```
##TODO: Plot the training accuracies and validation accuracies
X = range(max_epochs)
y3 = acc_list1
y4 = acc_list_val1
fig2, ax2 = plt.subplots(figsize=(8,6))
plt.plot(X, y3)
plt.plot(X, y4)
plt.xlabel('epochs')
plt.ylabel('training accuracies and validation accuracies')
plt.legend(['training accuracies', 'validation accuracies']);
```




This model overfits on the data. Because the training accuracy is higher than validation accuracy.

```
#Test
true_labels = []
predictions = []
correct_test = 0
model_test = net.eval()
with torch.no_grad():
    for batch_idx, (data, label) in enumerate(test_loader):
        data, label = data.to(device), label.to(device)
        ##TODO: pass the data into the network and store the output
        output_test = model_test(data)

        ##TODO: Get the prediction from the output
        _, predicted_test = torch.max(output_test.data, 1)


        ##TODO: Calculate the correct number and add the number to correct_test
        correct_test += (predicted_test == label).sum()


        ##TODO: update predictions list and true label list
        true_labels.append(label.item())
        predictions.append(predicted_test.item())

```

```
        ##We can directly append the value because here batch_size=1



print('Accuracy on the 10000 test images: %.2f %%' % (100 * correct_test / len(test_set)))
```

```
Accuracy on the 10000 test images: 67.55 %
```

```
##TODO: print the confusion matrix of test set
##You can use sklearn.metrics.confusion_matrix
confusion_matrix(true_labels, predictions)
```

```
array([[671,  28,  61,  13,  53,  11,  20,  22,  86,  35],
       [ 18, 821,   3,   6,   6,   1,  16,   7,  34,  88],
       [ 63,  11, 520,  61, 127,  39,  92,  58,  23,   6],
       [ 21,  13,  69, 441, 105, 100, 144,  64,  24,  19],
       [ 10,  10,  52,  42, 669,  14,  89,  91,  17,   6],
       [ 11,   4,  60, 198,  95, 433,  80, 101,   7,  11],
       [  4,   7,  23,  44,  41,  13, 853,   8,   6,   1],
       [ 13,  10,  23,  34,  74,  31,  15, 779,   4,  17],
       [ 57,  57,  17,  10,  18,   2,   7,   6, 796,  30],
       [ 26, 115,   9,  10,  14,   1,   6,  18,  29, 772]])
```

Cat and dog get confused most. This makes sense because cat looks very similar to dogs.

```

```