# 18662 Homework 4

**Due date: April 23rd, 5pm EST**

## Problem 1: Bayesian Learning

**[20 points]**

In class, we saw that Bayesian learning is optimal since it iteratively learns the true distribution as the number of sample data points increases. We will explore that concept in this problem. Please refer to *Section 20.1 Statistical Learning* in Russel & Norvig for review.

Recall this note from *Section 20.1*:

*"Bayesian learning simply calculates the probability of each hypothesis, given the data, and makes predictions on that basis. That is, the predictions are made by using all the hypotheses, weighted by their probabilities, rather than by using just a single "best" hypothesis. In this way, learning is reduced to probabilistic inference."*

In this problem, we have a coin for which we wish to figure out the bias (i.e. is it a fair coin that is heads for 50% of coin flips or is it biased and has some other probability that heads is flipped?). In this problem, we will consider 6 possible hypotheses (each denoting the probability that heads is flipped):

| | | |
|---|---|---|
| $h_1$: 0% | $h_2$: 20% | $h_3$: 40% |
| $h_4$: 60% | $h_5$: 80% | $h_6$: 100% |

We will also assume that each of the hypotheses is equally likely (i.e. assume a uniform prior). Perform a Bayesian Learning experiment, where you "flip" coins and determine the probability of each hypothesis with respect to the number of coin flips. The result of the coin flips will come from `bayesian_learning.mat`, where 1 indicates that a heads was flipped and a 0 indicates that a tails was flipped. Generate two plots: one that shows the posterior probability of each hypothesis with respect to the number of observations, and another that shows the probability that the next coin flip is heads with respect to the number of observations (see the plots from Russel and Norvig shown below). You may assume that observations are i.i.d. (independent and identically distributed).

**Continued on next page…**

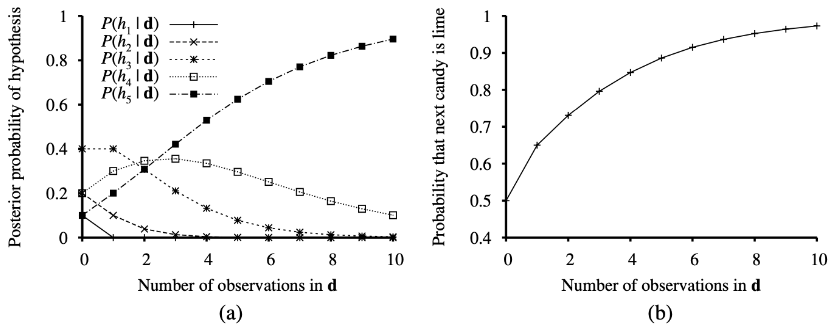


*Figure from Section 20.1 in Russel & Norvig*

In generating the first plot shown above, they multiplied the curves by some scaling factor α so that the curves approach either 0 or 100% probability (your plots don't have to exactly match those from the example). You may do this if you wish, but you can also simply normalize the probabilities at each trial of the experiment so that the probabilities sum to 1.

After generating both plots, provide a brief analysis. After all observations have been made, which is the most likely hypothesis or hypotheses? When analyzing the second plot, be sure to include how the hypothesis with the highest posterior probability influences the prediction.

You can start your code (for Problems 1-3) with the following:

```
import numpy as np
import scipy.io
import matplotlib.pyplot as plt

coin_data = scipy.io.loadmat('coin_data.mat')['coin_data_list']
```

## Problem 2: Maximum a Posteriori (MAP) Estimation

**[15 points]**

You might have noticed in the previous problem that there were many computations we had to perform to arrive at the answer. We can approximate Bayesian Learning, and we denote this approximation as maximum a posteriori (MAP) estimation. MAP estimation is very common given its simplicity and accuracy.

Regenerate the second plot (probability that the next coin flip is heads vs. the number of observations) based on this approximation. Refer to *Section 20.1* in Russel & Norvig. Try at least one other prior distribution (besides the prior uniform distribution you are already using) and compare the results. Provide a brief discussion on the differences between using MAP vs. Bayesian Learning for prediction based on the plots you have generated.

## Problem 3: Maximum Likelihood (ML) Estimation

**[15 points]**

Given the same data set as used in Problems 1 & 2, find the ML estimate of the bias of the coin. The unknown parameter here is the probability that a heads is flipped. Regenerate the second plot from before (probability that the next flip is heads vs. number of observations) using the ML estimate.

Questions to answer:

- Is the estimate equal to the estimates we found as a result of Bayesian Learning and MAP estimation? What is the major difference?
- Which is likely to be more accurate: the estimate found with ML estimation or the estimate found using the other methods?
- In what case (specifically, for what prior distribution) does MAP estimation reduce to the ML estimate? Feel free to consult *Section 20.2 Learning with Complete Data* in Russel & Norvig.
- Provide a brief discussion comparing Bayesian Learning, MAP, and ML estimation. What are the pros and cons of each?

# Problem 4: Gaussian Mixture Model with EM algorithm

## [20 points]

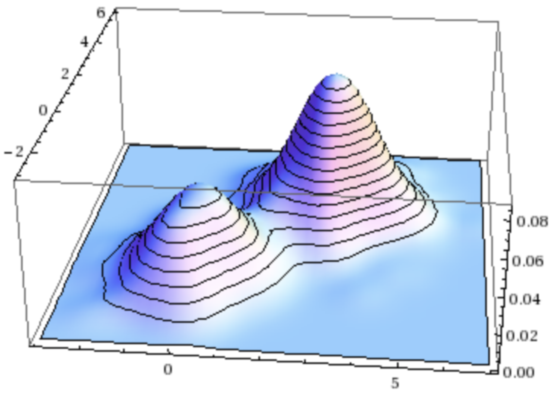
The Gaussian mixture model is simply a "mix" of Gaussian distributions. In this case, "Gaussian" means the multivariate normal distribution $\mathcal{N}(\boldsymbol{\mu}, \Sigma)$ and "mixture" means that several different gaussian distributions, all with different mean vectors $\boldsymbol{\mu}_j$ and different covariance matrices $\Sigma_j$, are combined by taking the weighted sum of the probability density functions:

$$f_{GMM}(\mathbf{x}) = \sum_{j=1}^{k} \phi_j f_{\mathcal{N}(\boldsymbol{\mu}_j, \Sigma_j)}(\mathbf{x})$$

subject to:

$$\sum_{j=1}^{k} \phi_j = 1$$

A single multivariate normal distribution has a single "hill" or "bump" located at $\boldsymbol{\mu}_i$; in contrast, a GMM is a multimodal distribution with on distinct bump per class. (Sometimes you get fewer than $k$ distinct local maxima in the p.d.f., if the bumps are sufficiently close together or if the weight of one class is zero or nearly so, but in general you get $k$ distinct bumps.) This makes it well suited to modeling data like that seen in our motivating example above, where there seems to be more than one region on high density.



We can view this is as a two-step generative process. To generate the $i$-th example:

1. Sample a random class index $C_i$ from the categorical distribution parameterized by $\boldsymbol{\phi} = (\phi_1, \ldots \phi_k)$.
2. Sample a random vector $\mathbf{X}_i$ from the multivariate distribution associated to the $C_i$-th class.

The $n$ independent samples $\mathbf{X}_i$ are the row vectors of the matrix $\mathbf{X}$.

Symbolically, we write:

$$C_i \sim \text{Categorical}(k, \boldsymbol{\phi}) \tag{3}$$
$$\mathbf{X}_i \sim \mathcal{N}(\boldsymbol{\mu}_{C_i}, \Sigma_{C_i}) \tag{4}$$

To fit a GMM model to a particular dataset, we attempt to find the maximum likelihood estimate of the parameters $\Theta$:

$$\Theta = \{\mu_1, \Sigma_1, \ldots, \mu_k, \Sigma_k\} \tag{5}$$

Because the $n \times m$ example matrix $\mathbf{X}$ is assumed to be a realization of $n$ i.i.d. samples from $f_{GMM}(\mathbf{x})$, we can write down our likelihood function as

$$\mathcal{L}(\Theta; \mathbf{X}) = P(\mathbf{X}; \Theta) = \prod_{i=1}^{n} \sum_{j=1}^{k} P(C_i = j) P(\mathbf{X}_i | C_i = j) \tag{6}$$

We know that $\mathbf{X}_i$ has a multivariate normal distribution with parameters determined by the class, so the conditional probability $P(\mathbf{X}_i | C_i = j)$ can be written down pretty much directly from the definition:

$$P(\mathbf{X}_i | C_i = j) = \frac{1}{\sqrt{(2\pi)^k |\Sigma_j|}} \exp\left( - \frac{(\mathbf{X}_i - \boldsymbol{\mu}_j)^T \Sigma_j^{-1} (\mathbf{X}_i - \boldsymbol{\mu}_j)}{2} \right) \tag{7}$$

Obtaining a formula for $P(C_i = j | \mathbf{X}_i)$ requires a little more work. We know that the unconditional probability is given by the parameter vector $\boldsymbol{\phi}$:

$$P(C_i = j) = \phi_j \tag{8}$$

So using Bayes' theorem, we can write this in terms of equation (7):

$$\begin{aligned} P(C_i = j | \mathbf{X}_i) &= \frac{P(C_i = j) P(\mathbf{X}_i | C_i = j)}{P(\mathbf{X}_i)} \\ &= \frac{\phi_j P(\mathbf{X}_i | C_i = j)}{\sum_{l=1}^{k} P(\mathbf{X}_i | C_i = l)} \end{aligned} \tag{9}$$

If we substituted equation (7) into (9) we could get a more explicit but very ugly formula, so I leave that to the reader's imagination.

Equations (6), (7), and (9), when taken together, constitute the complete likelihood function $\mathcal{L}(\Theta; \mathbf{X})$. However, these equations have a problem - they depend on the unknown random variable $C_i$. This variable tells us which class each $\mathbf{X}_i$ was drawn from and makes it much easier to reason about the distribution, but we don't actually know what $C_i$ is for any $i$. This is called a latent random variable and its presence in our model causes a kind of chicken-and-egg problem. If we knew $\boldsymbol{\mu}_j$ and $\Sigma_j$ for $j = (1, 2, \ldots, k)$ then we could make a guess about what $C_i$ is by looking at which $\boldsymbol{\mu}_j$ is closest to $\mathbf{X}_i$. If we knew $C_i$, we could estimate $\boldsymbol{\mu}_j$ and $\Sigma_j$ by simply taking the mean and covariance over all $X_i$ where $C_i = j$. But how can we estimate these two sets of parameters together, if we don't know either when we start?

## EM Algorithm for GMM

### E-step

Given the that centroid $\boldsymbol{\mu}_j$ and covariance matrix $\Sigma_j$ for class $j$ is fixed, we can update $w_{ij}$ by simply calculating the probability that $X_i$ came from each class and normalizing:

$$w_{ij} = \frac{P(X_i|K=j)}{P(K_i)} = \frac{P(X_i|K=j)}{\sum_{l=1}^{k} P(X_i|K=l)} \tag{10}$$

The conditional probablity $P(\mathbf{X}_i|K=j)$ is simply the multivariate normal distribution $\mathbf{X}_i\ \mathcal{N}(\mu_i, \Sigma_i)$ so we can use equation (4) above to calculate the probability density for each class, and then divide through by the total to normalize each row of $\mathbf{X}$ to 1. This gives us a concrete formula for the update to $w_ij$:

$$w_{ij} = \frac{f_{\mathcal{N}(\mu_i,\Sigma_i)}(\mathbf{X}_i)}{\sum_{l=1}^{k} f_{\mathcal{N}(\mu_l,\Sigma_l)}(\mathbf{X}_i)} \tag{11}$$

The probability of each class $\phi$ can then be estimated by averaging over all examples in the training set:

$$\phi_j = \sum_{i=1}^{n} w_{ij} \tag{12}$$

### M-step

Forget about the past estimates we had for $\boldsymbol{\mu}_j$ or $\Sigma$. Unlike gradient descent, the EM algorithm does not proceed by making small changes to the previous iteration's parameter estimates - instead, it makes a bold leap all the way to the *exact* estimate - but only in certain dimensions. In the M-step, we will calculate the ML estimates for $\boldsymbol{\mu}_j$ or $\Sigma$ assuming that $w_{ij}$ is held constant.

How can we make such a leap? Well, we have a matrix of $n$ observations $\mathbf{X}_i$ with weights $w_i$ which we believe came from a multivariate distribution $\mathcal{N}(\vec{\mu}, \Sigma)$. That means we can use the familiar formulas:

$$\boldsymbol{\mu}_j = \frac{1}{n}\sum_{i=1}^{n} w_{ij}\mathbf{X}_i \tag{13}$$

$$\Sigma_j = \frac{1}{n}\sum_{i=1}^{n} w_{ij}(\mathbf{X}_i - \boldsymbol{\mu}_j)(\mathbf{X}_i - \boldsymbol{\mu}_j)^T \tag{14}$$

These are in fact the ML estimate for these parameters for the multivariate normal distribution. As such, we don't need to worry about learning rate or gradients as we would with gradient descent because these estimates are already maximal! This is one of the neatest things about this algorithm.

## Task:

Let us load the iris dataset and carry out initial Exploratory Data Analysis

In [5]:

```python
## Loading the required libraries
from scipy.stats import mode
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
import seaborn as sns
import pandas as pd
iris = load_iris()
```

In [6]:

```python
## Lets see what the data looks like
data=pd.DataFrame(iris['data'],columns=['Petal length','Petal Width','Sepal Length',
data['Species']=iris['target']
data['Species']=data['Species'].apply(lambda x: iris['target_names'][x])
print(data.head())
data.describe()
```

```
   Petal length  Petal Width  Sepal Length  Sepal Width Species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa
3           4.6          3.1           1.5          0.2  setosa
4           5.0          3.6           1.4          0.2  setosa
```

Out[6]:

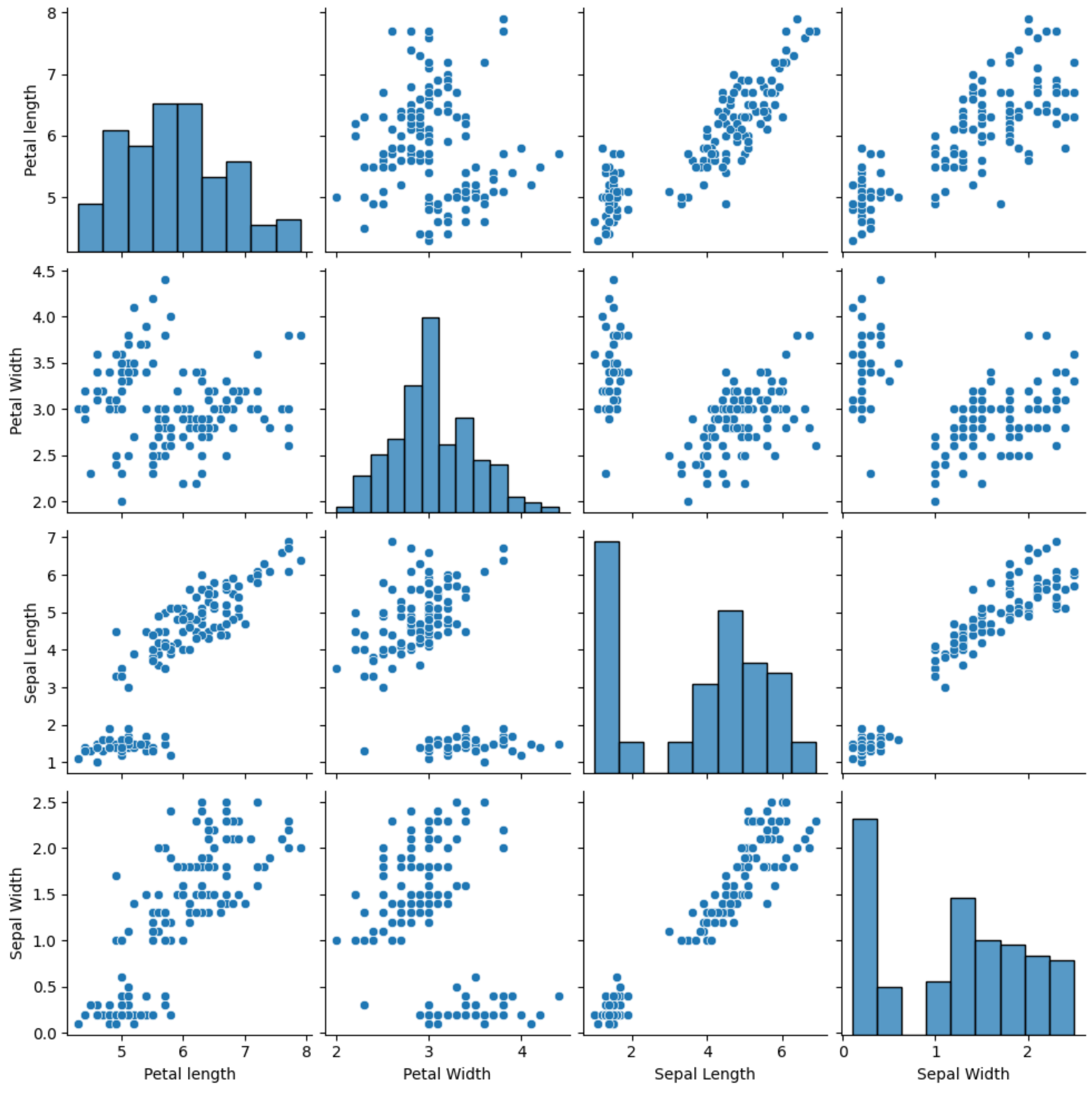
| | Petal length | Petal Width | Sepal Length | Sepal Width |
|---|---|---|---|---|
| **count** | 150.000000 | 150.000000 | 150.000000 | 150.000000 |
| **mean** | 5.843333 | 3.057333 | 3.758000 | 1.199333 |
| **std** | 0.828066 | 0.435866 | 1.765298 | 0.762238 |
| **min** | 4.300000 | 2.000000 | 1.000000 | 0.100000 |
| **25%** | 5.100000 | 2.800000 | 1.600000 | 0.300000 |
| **50%** | 5.800000 | 3.000000 | 4.350000 | 1.300000 |
| **75%** | 6.400000 | 3.300000 | 5.100000 | 1.800000 |
| **max** | 7.900000 | 4.400000 | 6.900000 | 2.500000 |

In [7]:

```
## Explore the relationship between variables
sns.pairplot(data)
```

Out[7]:

```
<seaborn.axisgrid.PairGrid at 0x7fa8bbac5a90>
```



## Let us load the data from the Dataset and use GMM to find clusters in this data

In [9]:

```
X = iris.data
```

## Complete the GMM class below by refering to theory above

In [2]:

```
import numpy as np
from scipy.stats import multivariate_normal

class GMM:
    def __init__(self, k, max_iter=5):


    def initialize(self, X):


    def e_step(self, X):
        # E-Step: update weights and phi holding mu and sigma constant


    def m_step(self, X):
        # M-Step: update mu and sigma holding phi and weights constant

    def fit(self, X):


    def predict_proba(self, X):


    def predict(self, X):
```

## Lets Evaluate the Model

In [11]:

```
np.random.seed(42)
gmm = GMM(k=3, max_iter=10)
gmm.fit(X)
```

In [12]:

```
permutation = np.array([
    mode(iris.target[gmm.predict(X) == i],keepdims=True).mode.item()
    for i in range(gmm.k)])
permuted_prediction = permutation[gmm.predict(X)]
print(np.mean(iris.target == permuted_prediction))
confusion_matrix(iris.target, permuted_prediction)
```

```
0.96
```

Out[12]:

```
array([[50,  0,  0],
       [ 0, 44,  6],
       [ 0,  0, 50]])
```

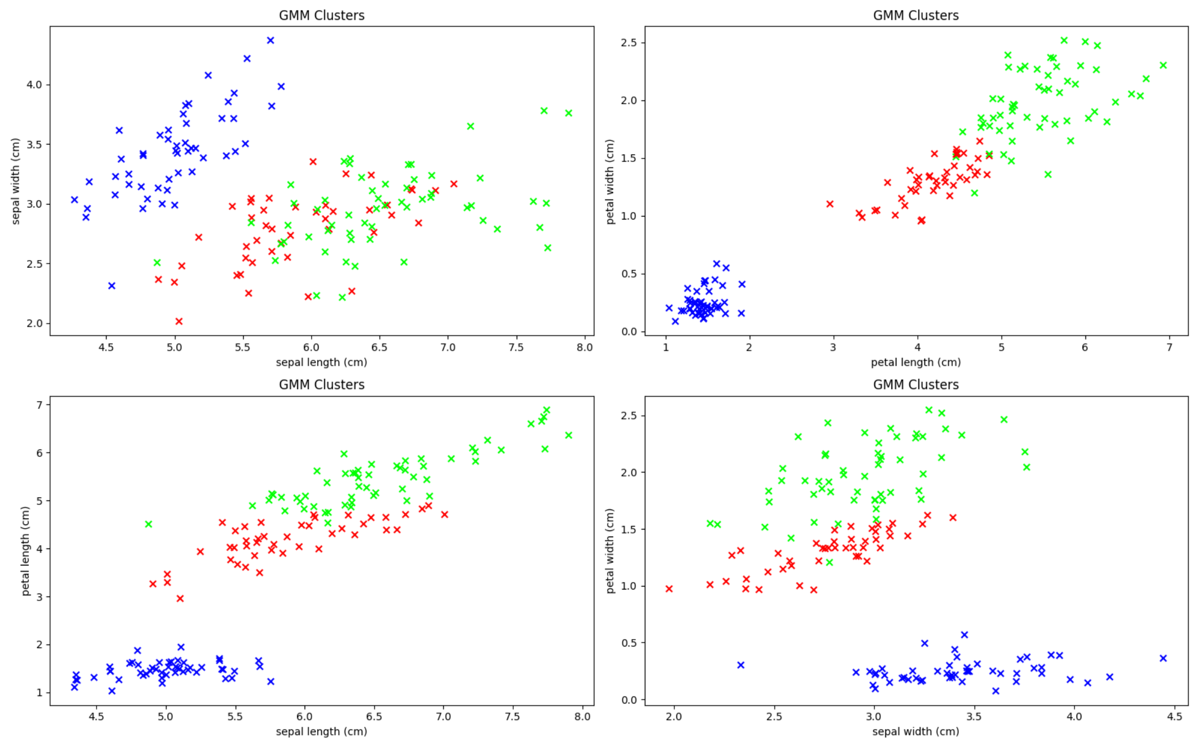
## Lets visualize the predicted clusters

In [13]:

```
def jitter(x):
    return x + np.random.uniform(low=-0.05, high=0.05, size=x.shape)

def plot_axis_pairs(X, axis_pairs, clusters, classes):
    n_rows = len(axis_pairs) // 2
    n_cols = 2
    plt.figure(figsize=(16, 10))
    for index, (x_axis, y_axis) in enumerate(axis_pairs):
        plt.subplot(n_rows, n_cols, index+1)
        plt.title('GMM Clusters')
        plt.xlabel(iris.feature_names[x_axis])
        plt.ylabel(iris.feature_names[y_axis])
        plt.scatter(
            jitter(X[:, x_axis]),
            jitter(X[:, y_axis]),
            c=clusters,
            cmap=plt.colormaps.get_cmap('brg'),
            marker='x')
    plt.tight_layout()

plot_axis_pairs(
    X=X,
    axis_pairs=[
        (0,1), (2,3),
        (0,2), (1,3) ],
    clusters=permuted_prediction,
    classes=iris.target)
```



In [ ]:

```

```

## Problem 5: Neural Networks

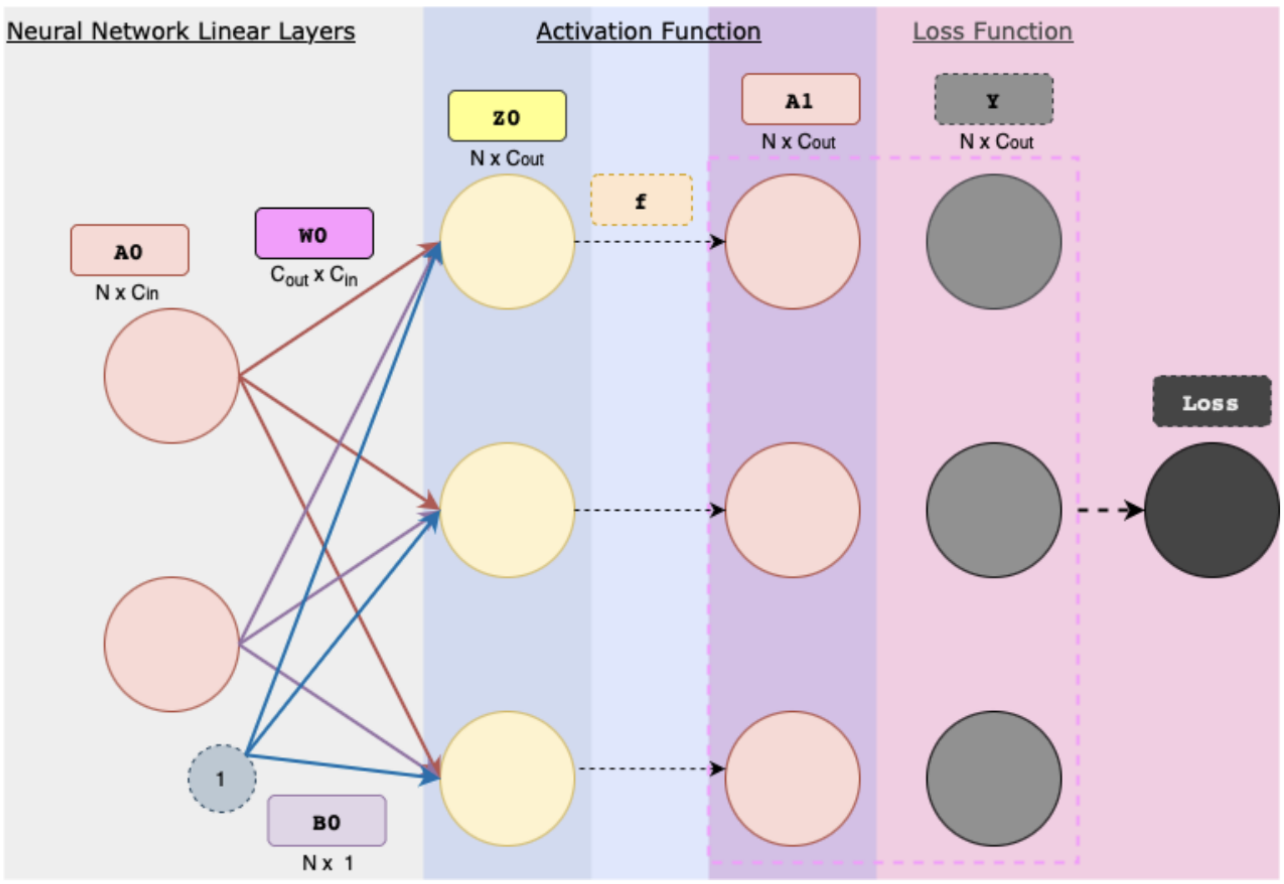
### [20 points]

A neural network is a method in artificial intelligence that teaches computers to process data in a way that is inspired by the human brain. It is a type of machine learning process, called deep learning, that uses interconnected nodes or neurons in a layered structure that resembles the human brain. It creates an adaptive system that computers use to learn from their mistakes and improve continuously. Thus, artificial neural networks attempt to solve complicated problems, like summarizing documents or recognizing faces, with greater accuracy.

The forward process of a neural network involves taking an input, passing it through the layers of the network, and producing an output. This is also referred to as the "feedforward" process. In this process, the input is multiplied by the weights of the connections between the neurons in the network, and then a non-linear activation function is applied to the result. This is repeated for each layer in the network until the final output is produced.

The backward process, also known as backpropagation, is the process of updating the weights of the connections in the network during training. This is done by computing the gradient of the error between the predicted output and the actual output with respect to the weights in the network. The gradient is then used to update the weights in the opposite direction of the gradient to minimize the error. This process is repeated for each layer in the network, propagating the error backwards from the output layer to the input layer.

The forward and backward processes work together in a neural network to train the network to make accurate predictions on new, unseen data.

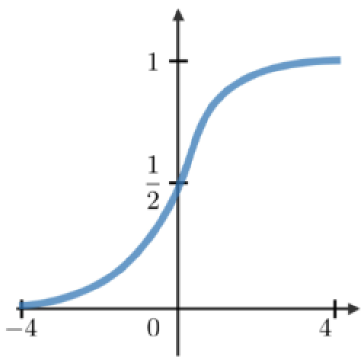
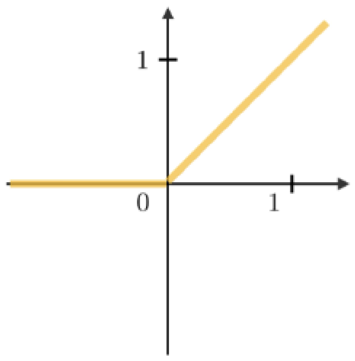
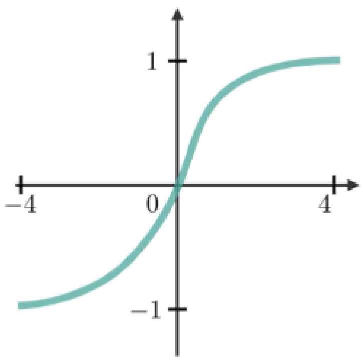


## Problem 5a: Activation Function

Here, we will introduce to you a few popular activation functions and how to implement them!

The primary purpose of having nonlinear components in the neural network (fNN ) is to allow it to approximate nonlinear functions. Without activation functions, fNN will always be linear, no matter how deep it is. The reason is that A ·W + b is a linear function, and a linear function of a linear function is also linear.

Popular choices of activation functions are Sigmoid, as well as ReLU and Tanh:

| Sigmoid | ReLU | Tanh |
|---|---|---|
| $\frac{1}{1+e^{-z}}$ | $\max(0, z)$ | $\frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$ |
|  |  |  |

In this section, your task is to implement the Activation class: Class attributes:

- Activation functions have no trainable parameters.
- Variables stored during forward-propagation to compute derivatives during back-propagation: layer output A.

Class methods:

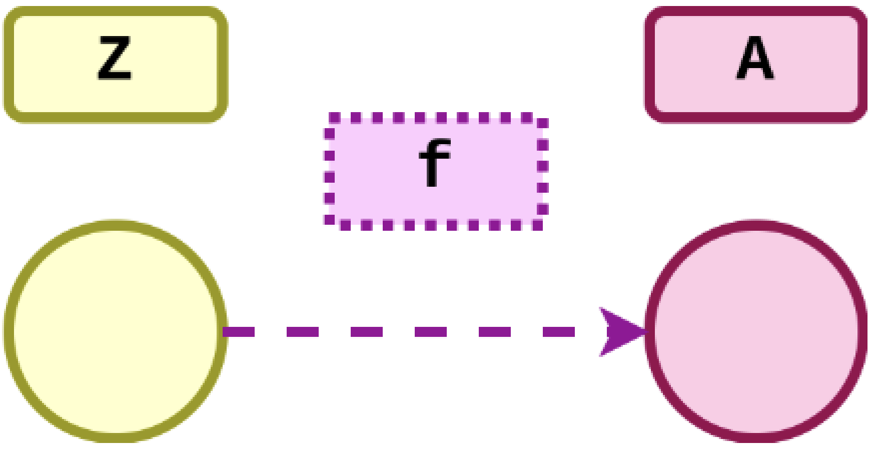
- forward: forward method takes in a batch of data Z of shape N ×C (representing N samples where each sample has C features), and applies the activation function to each element of Z to compute output A of shape N ×C.
- backward: backward method calculates and returns dAdZ, how changes in pre-activation features Z affect post-activation values A. It is used to enable downstream computation, as seen in subsequent sections.

Activation Function Components:

| Code Name | Math | Type | Shape | Meaning |
|---|---|---|---|---|
| `N` | $N$ | scalar | - | batch size |
| `C` | $C$ | scalar | - | number of features |
| `Z` | $Z$ | matrix | $N \times C$ | batch of $N$ inputs each represented by $C$ features |
| `A` | $A$ | matrix | $N \times C$ | batch of $N$ outputs each represented by $C$ features |
| `dAdZ` | $\partial A / \partial Z$ | matrix | $N \times C$ | how changes in pre-activation features affect post-activation values |

Note: By convention in this class, Z is the output of a linear layer, and A is the input of a linear layer. Here, Z is the output from the previous linear layer and A is the input to the next linear layer, i.e. let fl be the activation function of layer l,

$$A_{l+1} = f_l(Z_l).$$

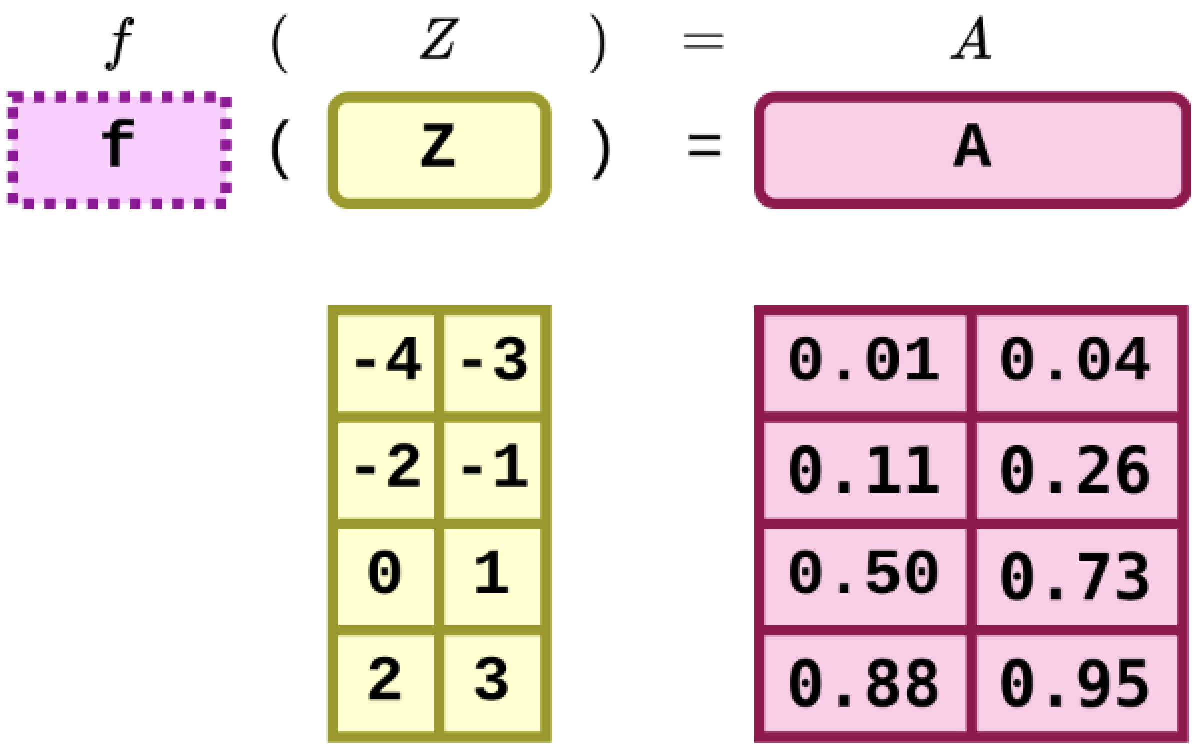


## Sigmoid

### Sigmoid Forward

During forward propagation, pre-activation features Z are passed to the activation function Sigmoid to calculate their post-activation values A.

$$
\begin{aligned}
A &= \texttt{Sigmoid.forward}(Z) \\
&= \varsigma(Z) \\
&= \frac{1}{1+e^{-Z}}
\end{aligned}
$$



**Sigmoid Backward**

Backward propagation helps us understand how changes in pre-activation features Z affect post-activation values A.

$$
\begin{aligned}
\frac{dA}{dZ} &= \texttt{sigmoid.backward}() \\
&= \varsigma(Z) - \varsigma^2(Z) \\
&= A - A \odot A
\end{aligned}
$$

## Tanh

### Tanh Forward

$$
\begin{aligned}
A &= \texttt{Tanh.forward}(Z) \\
&= \tanh(Z) \\
&= \frac{e^{Z} - e^{-Z}}{e^{Z} + e^{-Z}}
\end{aligned}
$$

$$
f \quad ( \quad Z \quad ) \quad = \quad A
$$

$$
\boxed{\mathtt{f}} \; ( \; \boxed{\mathtt{Z}} \; ) \; = \; \boxed{\mathtt{A}}
$$

$$
\begin{bmatrix} -4 & -3 \\ -2 & -1 \\ 0 & 1 \\ 2 & 3 \end{bmatrix}
\qquad
\begin{bmatrix} -0.99 & -0.99 \\ -0.96 & -0.76 \\ 0.00 & 0.76 \\ 0.96 & 0.99 \end{bmatrix}
$$

### Tanh Backward

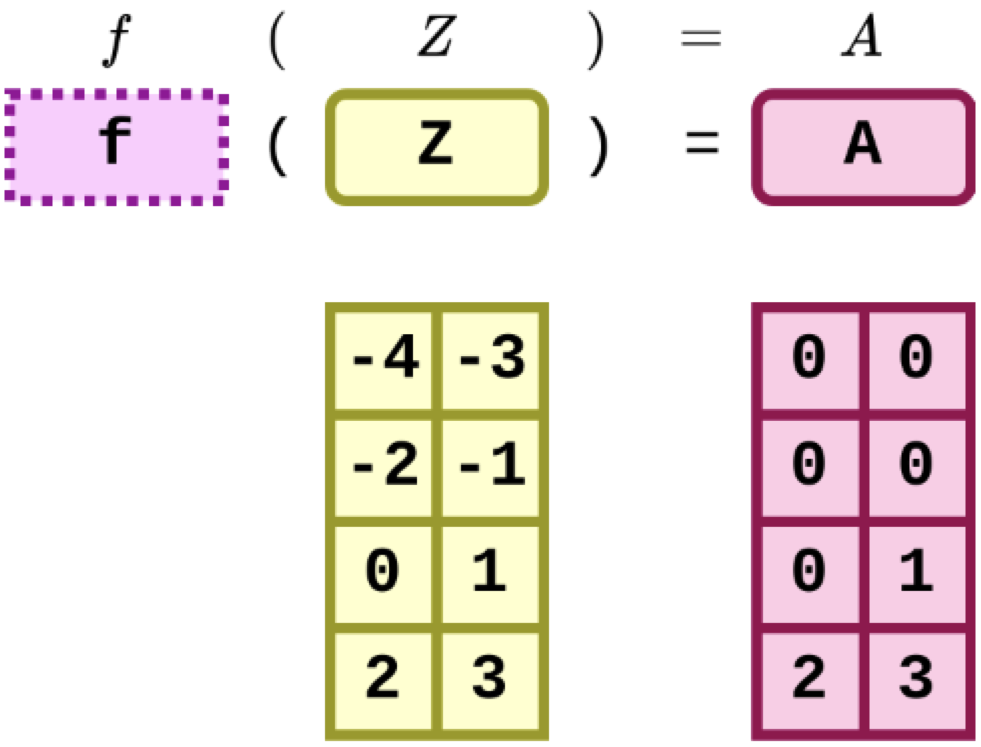
Fill in the blank in the equation below. Represent the final result in terms of A, similar to Sigmoid backward equation in the previous section.

$$
\begin{aligned}
\frac{dA}{dZ} &= \texttt{tanh.backward}() \\
&= \_?\_
\end{aligned}
$$

## RELU

### RELU Forward

Recall the equation of ReLU and fill in the blank below:



| | |
|---|---|
| -4 | -3 |
| -2 | -1 |
| 0 | 1 |
| 2 | 3 |

| | |
|---|---|
| 0 | 0 |
| 0 | 0 |
| 0 | 1 |
| 2 | 3 |

Hint: You might find the graph of ReLU helpful

### RELU Backward

Complete the piece-wise function for relu.backward:

In [ ]:

```python
import numpy as np


class Identity:

    def forward(self, Z):

        self.A = Z

        return self.A

    def backward(self):

        dAdZ = np.ones(self.A.shape, dtype="f")

        return dAdZ


class Sigmoid:
    """
    On same lines as above:
    Define 'forward' function
    Define 'backward' function
    Read the writeup for further details on Sigmoid.
    """
    def forward(self,Z):

        return self.A
    def backward(self):

        return dAdZ


class Tanh:
    """
    On same lines as above:
    Define 'forward' function
    Define 'backward' function
    Read the writeup for further details on Tanh.
    """
    def forward(self,Z):

        return self.A
    def backward(self):

        return dAdZ



class ReLU:
    """
    On same lines as above:
    Define 'forward' function
    Define 'backward' function
    Read the writeup for further details on ReLU.
    """
    def forward(self,Z):

        return self.A
    def backward(self):

        return dAdZ
```

## Problem 5b: Criterion - Loss Functions

Much as you did for activation functions you will now program some simple loss functions. Different loss functions may become useful depending on the type of neural network and type of data you are using. Here we will program Mean Squared Error Loss MSE and Cross Entropy Loss. It is important to know how these are calculated, and how they will be used to update your network. As before we will provide the formulas, and know that each of these functions can be done in less than 10 lines of code, so if your code begins to get more complex than that you may be overthinking the problem.

In this section, your task is to implement the forward and backward attribute functions of the Loss class in file loss.py:

Class attributes:

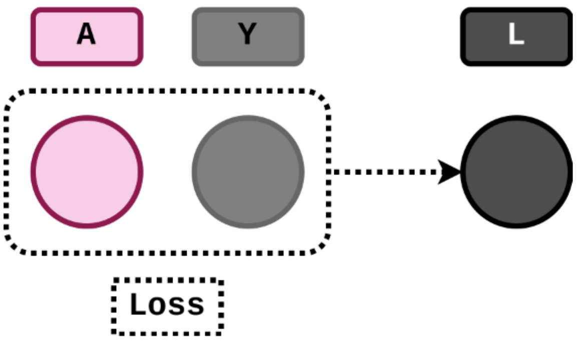
- Stores model prediction A to compute back-propagation.
- Stores desired output Y stored to compute back-propagation.

Class methods:

- forward: forward method takes in model prediction A and desired output Y of the same shape to calculate and return a loss value L. The loss value is a scalar quantity used to quantify the mismatch between the network output and the desired output.
- backward: backward method calculates and returns dLdA, how changes in model outputs A affect loss L. It is used to enable downstream computation, as seen in previous sections.

Table 4: Loss Function Components

| Code Name | Math | Type | Shape | Meaning |
|---|---|---|---|---|
| N | $N$ | scalar | - | batch size |
| C | $C$ | scalar | - | number of classes |
| A | $A$ | matrix | $N \times C$ | model outputs |
| Y | $Y$ | matrix | $N \times C$ | ground-truth values |
| L | $L$ | scalar | - | loss value |
| dLdA | $\partial L/\partial A$ | matrix | $N \times C$ | how changes in model outputs affect loss |



### MSE Loss

MSE stands for Mean Squred Error, and is often used to quantify the prediction error for regression problems. Regression is a problem of predicting a real-valued label given an unlabeled example. Estimating house price based on features such as area, location, the number of bedrooms and so on is a classic regression problem

**MSE Loss Forward Equation**

We first calculate the squared error SE between the model outputs A and the ground-truth values Y:

$$SE(A, Y) = (A - Y) \odot (A - Y) \tag{32}$$

Then we calculate the sum of the squared error $\mathbf{SSE}$, where $\iota_{\mathbf{N}}, \iota_{\mathbf{C}}$ are column vectors of size $N$ and $C$ which contain all 1s:

$$SSE(A, Y) = \iota_N^T \cdot SE(A, Y) \cdot \iota_C \tag{33}$$

Lastly, we calculate the per-component Mean Squared Error $\mathbf{MSE}$ loss:

$$MSELoss(A, Y) = \frac{SSE(A, Y)}{2 \cdot N \cdot C} \tag{34}$$

**MSE Loss Backward Equation**

$$\texttt{MSELoss.backward}() = \frac{A - Y}{N \cdot C}$$

## Cross-Entropy Loss

Cross-entropy loss if one of the most commonly used loss function for probability-based classification problems. In this course, most of the part 2 homework problems involve classification problems, hence you will use this loss function very often.

**Cross-Entropy Loss Forward Equation**

Cross-Entropy Loss Forward Equation Firstly, we use softmax function to transform the raw model outputs A into a probability distribution consisting of C classes proportional to the exponentials of the input numbers. ιN,ιC are column vectors of size N and C which contain all 1s.

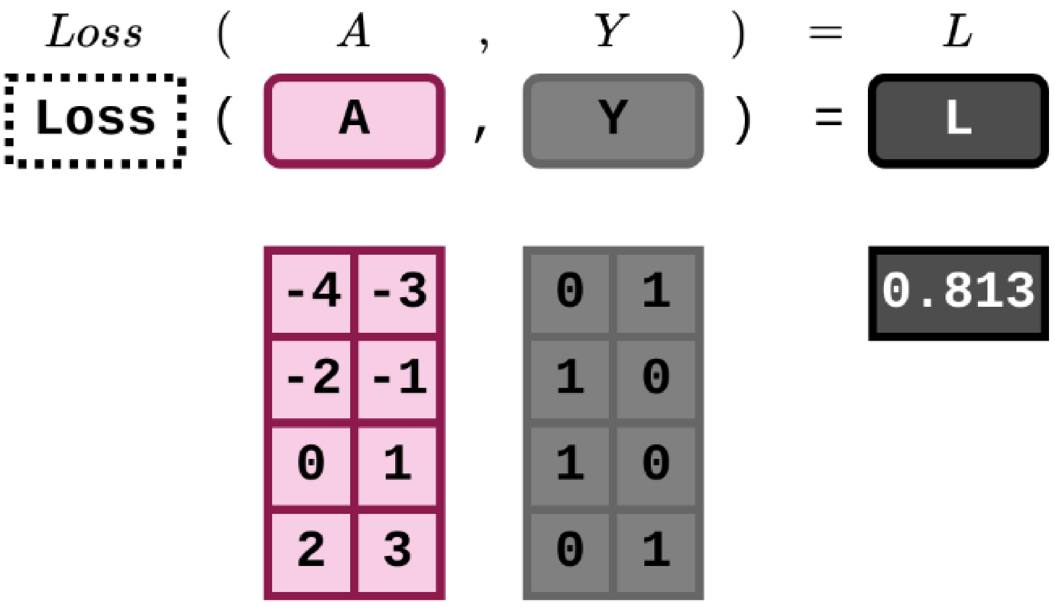
$$\begin{aligned}\texttt{softmax}(A) &= \sigma(A) \\ &= \frac{\exp(A)}{\sum_{j=1}^{C} \exp(A_{ij})}\end{aligned}$$

Now, each row of A represents the model's prediction of the probability distribution while each row of Y represents target distribution of an input in the batch. Then, we calculate the cross-entropy H(A,Y) of the distribution Ai relative to the target distribution Yi for i = 1,...,N:

$$\begin{aligned}\texttt{crossentropy} &= H(A, Y) \\ &= (-Y \odot \log(\sigma(A))) \cdot \iota_C\end{aligned}$$

Remember that the output of a loss function is a scalar, but now we have a column matrix of size N. To transform it into a scalar, we can either use the sum or mean of all cross-entropy. Here, we choose to use the mean cross-entropy as the cross-entrpy loss as that is the default for PyTorch as well:

$$\begin{aligned}\texttt{sum\_crossentropy\_loss} &:= \iota_N^T \cdot H(A, Y) \\ &= SCE(A, Y) \\ \texttt{mean\_crossentropy\_loss} &:= \frac{SCE(A, Y)}{N}\end{aligned}$$



In [ ]:

```python
import numpy as np


class MSELoss:

    def forward(self, A, Y):
        """
        Calculate the Mean Squared error
        :param A: Output of the model of shape (N, C)
        :param Y: Ground-truth values of shape (N, C)
        :Return: MSE Loss(scalar)

        """

        return mse

    def backward(self):


        return dLdA


class CrossEntropyLoss:

    def forward(self, A, Y):
        """
        Calculate the Cross Entropy Loss
        :param A: Output of the model of shape (N, C)
        :param Y: Ground-truth values of shape (N, C)
        :Return: CrossEntropyLoss(scalar)

        Refer the the writeup to determine the shapes of all the variables.
        Use dtype ='f' whenever initializing with np.zeros()
        """


    def backward(self):


        return dLdA
```

# 6 Hidden Markov Models (10 points)

**Complete the "TODO"s in Section 6.2 of `hmm-em.ipynb` based on the example outlined below.**

Hidden Markov Models (HMMs) can handle a wide range of distributions, such as, discrete tables, Gaussians, and mixtures of Gaussians.

HMMs assume latent (hidden) discrete multinomial variables $\{\mathbf{z}_n\}$, which generate the corresponding observations $\{\mathbf{x}_n\}$. The observers can see only $\{\mathbf{x}_n\}$, and the model is estimated using observations, $\{\mathbf{x}_n\}$.

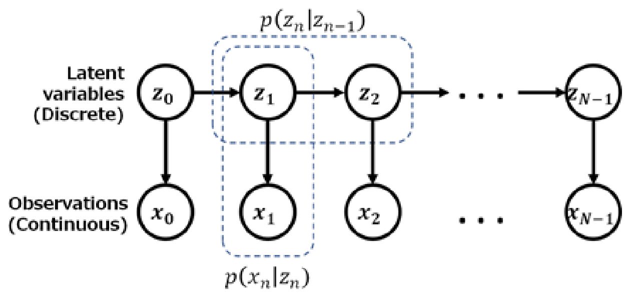


Figure 6.1: Hidden Markov Models

In HMM, $p(\mathbf{z}_n|\mathbf{z}_{n-1})$ is called a **transition probability**, and $p(\mathbf{x}_n|\mathbf{z}_n)$ is an **emission probability**.

**Note:**, A scalar variable is denoted by normal letters (such as, $x$), and a vector (or a matrix) is denoted by bold letters (such as, $\mathbf{x}$).

## 6.1 Sampling in Hidden Markov Models (Generate sample data)

First, we will generate sample data (observations) by using the distribution of Hidden Markov Models (HMM).

The distribution of the latent (hidden) variables $\{\mathbf{z}_n\}$ is discrete, and it then corresponds to a table of transitions.

For sampling, we will create a set of latent (hidden) variables, $\{\mathbf{z}_n\}$, in which it has 3 states (i.e., $K = 3$) with the following transition probabilities $p(\mathbf{z}_n|\mathbf{z}_{n-1})$.

$$A = \begin{bmatrix} 0.7 & 0.15 & 0.15 \\ 0.0 & 0.5 & 0.5 \\ 0.3 & 0.35 & 0.35 \end{bmatrix}$$

From now, we will use the letter $k \in \{0, 1, 2\}$ for the corresponding 3 states, and assume $\mathbf{z}_n = (z_{n,0}, z_{n,1}, z_{n,2})$, in which $z_{n,k'} = 1$ and $z_{n,k \neq k'} = 0$ in state $k'$.

Next we will create the corresponding observation $\{\mathbf{x}_n\}$ for sampling.
Here we assume 2-dimensional **Gaussian distribution** $\mathcal{N}(\mu_k, \mathbf{\Sigma}_k)$ for emission probabilities $p(\mathbf{x}_n|\mathbf{z}_n)$, when $\mathbf{z}_n$ belongs to $k$. ($k = 0, 1, 2$)
In order to simplify, we also assume that parameters $\mu_k, \mathbf{\Sigma}_k$ are independent for different components $k = 0, 1, 2$.

In this example, we set $\mu_k, \mathbf{\Sigma}_k$ as follows.

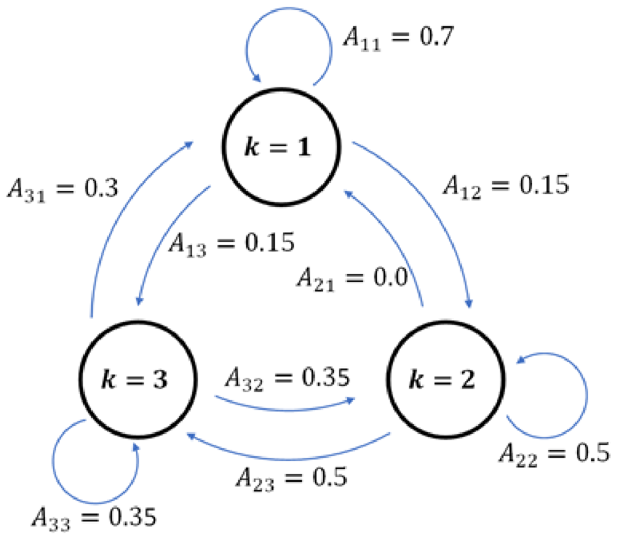


Figure 6.2: HMM Discrete Transition

$$\mu_0 = (16.0, 1.0), \quad \mathbf{\Sigma}_0 = \begin{bmatrix} 4.0 & 3.5 \\ 3.5 & 4.0 \end{bmatrix}$$

$$\mu_1 = (1.0, 16.0), \quad \mathbf{\Sigma}_1 = \begin{bmatrix} 4.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix}$$

$$\mu_2 = (-5.0, -5.0), \quad \mathbf{\Sigma}_2 = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 4.0 \end{bmatrix}$$

### 6.2 EM algorithm in Hidden Markov Models (HMM)

Now, using the given observation $\{\mathbf{x}_n\}$, let's try to estimate the optimal parameters in HMM.

When we denote unknown parameters by $\theta$, our goal is to get the optimal parameters $\theta$ to maximize the following (1).

$$p(\mathbf{X}|\theta) = \sum_{\mathbf{Z}} p(\mathbf{X}, \mathbf{Z}|\theta) \quad (1)$$

where $\mathbf{Z} = \{\mathbf{z}_n\}$ and $\mathbf{X} = \{\mathbf{x}_n\}$.

In this example, we use the following parameters as $\theta = \{\pi, \mathbf{A}, \mu, \mathbf{\Sigma}\}$.

- $\pi_k (k \in \{0, 1, 2\})$ : The possibility (scalar) for component $k$ in initial latent node $\mathbf{z}_0$. ($\Sigma_k \pi_k = 1$)
- $A_{j,k}$ $(j, k \in \{0, 1, 2\})$ : The transition probability (scalar) for the latent variable $\mathbf{z}_{n-1}$ to $\mathbf{z}_n$, in which $\mathbf{z}_{n-1}$ belongs to $j$ and $\mathbf{z}_n$ belongs to $k$. ($\Sigma_k A_{j,k} = 1$)
- $\mu_k$ : The mean (2-dimensional vector) for Gaussian distribution in emission probabilities $p(\mathbf{x}_n|\mathbf{z}_n)$ when the latent variable $\mathbf{z}_n$ belongs to $k$.

- $\mathbf{\Sigma}_k$ : The covariance matrix ($2\times2$ matrix) for Gaussian distribution in emission probabilities $p(\mathbf{x}_n|\mathbf{z}_n)$ when the latent variable $\mathbf{z}_n$ belongs to $k$.

In (1), the number of parameters will rapidly increase, when the number of states $K$ increases (in this example, $K = 3$). Furthermore, it has summation (not multiplication) in distribution (1), and the log likelihood will then lead to a complex expression in maximum likelihood estimation (MLE). Therefore, it will be difficult to directly apply maximum likelihood estimation (MLE) for the expression (1).

In practice, the expectation–maximization algorithm (shortly, **EM algorithm**) can often be applied to solve parameters in HMM.

In EM algorithm for HMM, we start with initial parameters $\theta^{old}$, and optimize (find) new $\theta$ to maximize the following expression (2). By repeating this operation, we can expect to reach the likelihood parameters $\hat{\theta}$.

$$Q(\theta, \theta^{old}) = \sum_{\mathbf{Z}} p(\mathbf{Z}|\mathbf{X}, \theta^{old}) \ln p(\mathbf{X}, \mathbf{Z}|\theta) \quad (2)$$

**Note**: For the essential idea of EM algorithm, see Chapter 9 in "" (Christopher M. Bishop, Microsoft)

Now we denote the discrete probability $p(\mathbf{z}_n|\mathbf{X}, \theta^{old})$ by $\gamma(z_{n,k})$ $(k = 0, 1, 2)$, in which $\gamma(z_{n,k})$ represents the probability of $\mathbf{z}_n$ for belonging to $k$. We also denote the discrete probability $p(\mathbf{z}_{n-1}, \mathbf{z}_n|\mathbf{X}, \theta^{old})$ by $\xi(z_{n-1,j}, z_{n,k})$ $(j, k = 0, 1, 2)$, in which $\xi(z_{n-1,j}, z_{n,k})$ represents the joint probability that $\mathbf{z}_{n-1}$ belongs to $j$ and $\mathbf{z}_n$ belongs to $k$.

In Gaussian HMM (in the above model), the equation (2) is written as follows, using $\gamma()$ and $\xi()$.

$$\begin{aligned} Q(\theta, \theta^{old}) = &\sum_{k=0}^{K-1} \gamma(z_{0,k}) \ln \pi_k + \sum_{n=1}^{N-1} \sum_{j=0}^{K-1} \sum_{k=0}^{K-1} \xi(z_{n-1,j}, z_{n,k}) \ln A_{j,k} \\ &+ \sum_{n=0}^{N-1} \sum_{k=0}^{K-1} \gamma(z_{n,k}) \ln p(\mathbf{x}_n|\mu_k, \mathbf{\Sigma}_k) \quad (3) \end{aligned}$$

where

$$\gamma(\mathbf{z}_n) = p(\mathbf{z}_n|\mathbf{X}, \theta^{old})$$
$$\xi(\mathbf{z}_{n-1}, \mathbf{z}_n) = p(\mathbf{z}_{n-1}, \mathbf{z}_n|\mathbf{X}, \theta^{old})$$

It's known that $\gamma()$ and $\xi()$ can be given by the following $\alpha()$ and $\beta()$, which are determined recursively. (i.e, We can first determine all $\alpha()$ and $\beta()$ recursively, and then we can obtain $\gamma()$ and $\xi()$ with known $\alpha(), \beta()$.)

$$\gamma(z_{n,k}) = \frac{\alpha(z_{n,k})\beta(z_{n,k})}{\sum_{k=0}^{K-1} \alpha(z_{n,k})\beta(z_{n,k})}$$
$$\xi(z_{n-1,j}, z_{n,k}) = \frac{\alpha(z_{n-1,j}) p(\mathbf{x}_n|\mu_k^{old}, \mathbf{\Sigma}_k^{old}) A_{j,k}^{old} \beta(z_{n,k})}{\sum_{j=0}^{K-1} \sum_{k=0}^{K-1} \alpha(z_{n-1,j}) p(\mathbf{x}_n|\mu_k^{old}, \mathbf{\Sigma}_k^{old}) A_{j,k}^{old} \beta(z_{n,k})}$$

where all $\alpha()$ and $\beta()$ are recursively given by

$$\alpha(z_{n,k}) = p(\mathbf{x}_n|\mu_k^{old}, \mathbf{\Sigma}_k^{old}) \sum_{j=0}^{K-1} A_{jk}^{old} \alpha(z_{n-1,j})$$

$$\beta(z_{n-1,k}) = \sum_{j=0}^{K-1} A_{k,j}^{old} p(\mathbf{x}_n|\mu_j^{old}, \mathbf{\Sigma}_j^{old}) \beta(z_{n,j})$$

Now we need the starting condition for recursion, $\alpha()$ and $\beta()$, and these are given as follows.

$$\alpha(z_{0,k}) = \pi_k^{old} p(\mathbf{x}_0|\mu_k^{old}, \mathbf{\Sigma}_k^{old})$$
$$\beta(z_{N-1,k}) = 1$$

**Note**: You can check the proofs of these Gaussian HMM properties in Chapter 13 of ”” (Christopher M. Bishop, Microsoft)

Once you have got $\gamma()$ and $\xi()$, you can get the optimal $\theta = \{\pi, \mathbf{A}, \mu, \mathbf{\Sigma}\}$ to maximize (3) as follows by applying Lagrange multipliers.

$$\pi_k = \frac{\gamma(z_{0,k})}{\sum_{j=0}^{K-1} \gamma(z_{0,j})}$$

$$A_{j,k} = \frac{\sum_{n=1}^{N-1} \xi(z_{n-1,j}, z_{n,k})}{\sum_{l=0}^{K-1} \sum_{n=1}^{N-1} \xi(z_{n-1,j}, z_{n,l})}$$

$$\mu_k = \frac{\sum_{n=0}^{N-1} \gamma(z_{n,k}) \mathbf{x}_n}{\sum_{n=0}^{N-1} \gamma(z_{n,k})}$$

$$\mathbf{\Sigma}_k = \frac{\sum_{n=0}^{N-1} \gamma(z_{n,k})(\mathbf{x}_n - \mu_k)(\mathbf{x}_n - \mu_k)^T}{\sum_{n=0}^{N-1} \gamma(z_{n,k})}$$

You repeat this process by replacing $\theta^{old}$ with this new $\theta$, and you will eventually get the optimal results $\hat{\theta}$ to maximize (1).

In practice, $\alpha()$ and $\beta()$ will quickly go to zero (because it's recursively multiplied by $p(\mathbf{x}_n|\mu_k^{old}, \mathbf{\Sigma}_k^{old})$ and $A_{j,k}^{old}$) and it will then exceed the dynamic range of precision in computation, when $N$ is large. For this reason, the coefficients, called **scaling factors**, will be introduced to normalize $\alpha()$ and $\beta()$ in each step $n$. The scaling factors will be canceled in EM algorithms, however, when you monitor the value of likelihood functions, you'll need to record scaling factors and apply these factors.