ECE521: Assignment 3

Unsupervised Learning and Probabilistic Models

Friday, March 24, 2017

Rahul Chandan | 999781801

Christine Lee | 999836646

1 K-means [18 pt.]

1.1 Learning K-means [8 pt.]

1. *Is the loss function $\mathcal{L}(\boldsymbol{\mu})$ convex in $\boldsymbol{\mu}$? Why or why not? Give a rigorous explanation. [3pt.]*

SOLUTION:

The loss function $\mathcal{L}(\boldsymbol{\mu}) \sum_{n=1}^{B} min_{k=1}^{K} \|x_n - \mu_k\|_2^2$ is not convex in $\boldsymbol{\mu}$. We can explain this mathematically: the loss function is not convex because $\mathcal{L}(\boldsymbol{\mu})$ it not defined on a convex set.

We can also prove this using a trivial example in the Cartesian plane where the solution is not unique. Consider the following data points: $\{(0,0), (0,1), (1,0), (1,1)\}$. If we set K = 2, there exists a local minimum for $\mu = \left\{\left(0, \frac{1}{2}\right), \left(1, \frac{1}{2}\right)\right\}$ and another local minimum for $\mu = \left\{\left(\frac{1}{2}, 0\right), \left(\frac{1}{2}, 1\right)\right\}$. Since there exists multiple local minima, it proves that the loss function is not convex.

2. *For the dataset data2D.npy, set K = 3 and find the K-means clusters $\boldsymbol{\mu}$ by minimizing the $\mathcal{L}(\boldsymbol{\mu})$ using the gradient descent optimizer. The parameters $\boldsymbol{\mu}$ should be initialized by sampling from the standard normal distribution. Include a plot of the loss vs the number of updates. Hints: you may want to use the Adam optimizer for this assignment with following hyper-parameter tf.train.AdamOptimizer(LEARNINGRATE, beta1 = 0.9, beta2 = 0.99, epsilon = 1e-5). The learning should converge within a few hundred updates. [2 pt.]*

SOLUTION:

The learning rate was tuned by observing the loss over 600 epochs for the following learning rates: $\eta = \{0.1, 0.01, 0.001\}$. The plot showing the loss history for the different learning rates can be found in **Figure 1**. From the plot, it was observed the optimal learning rate is around 0.01.

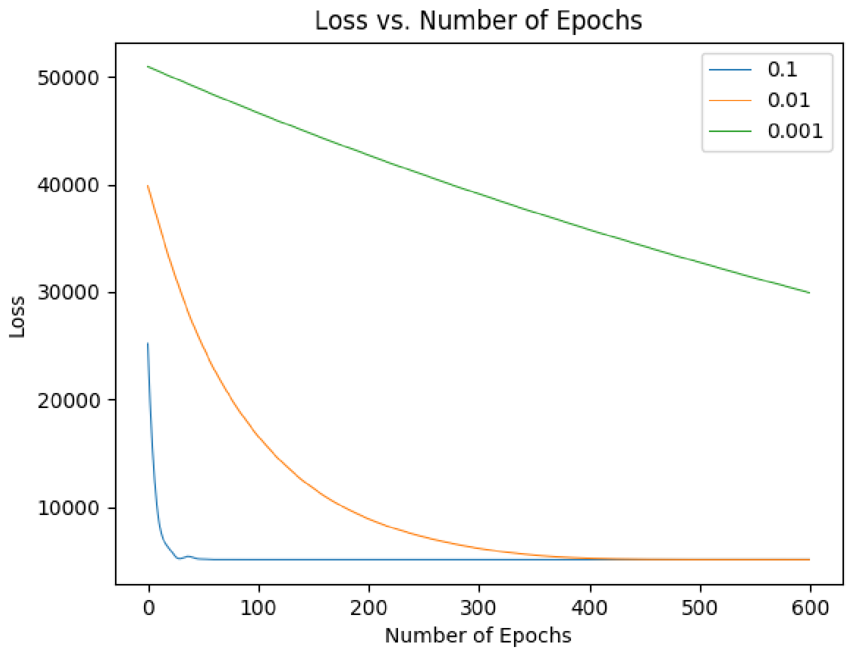


Figure 1: Loss vs. Number of Epochs for Different Learning Rates

Initializing the parameter $\boldsymbol{\mu}$ by sampling from the standard normal distribution, using the Adam optimizer with the hyperparameters: beta1 = 0.9, beta2 = 0.99, and epsilon = 1e-5, and using 600 epochs, the plot showing loss vs. number of epochs can be found below in **Figure 2**. The minimum loss was 5187.50488281.

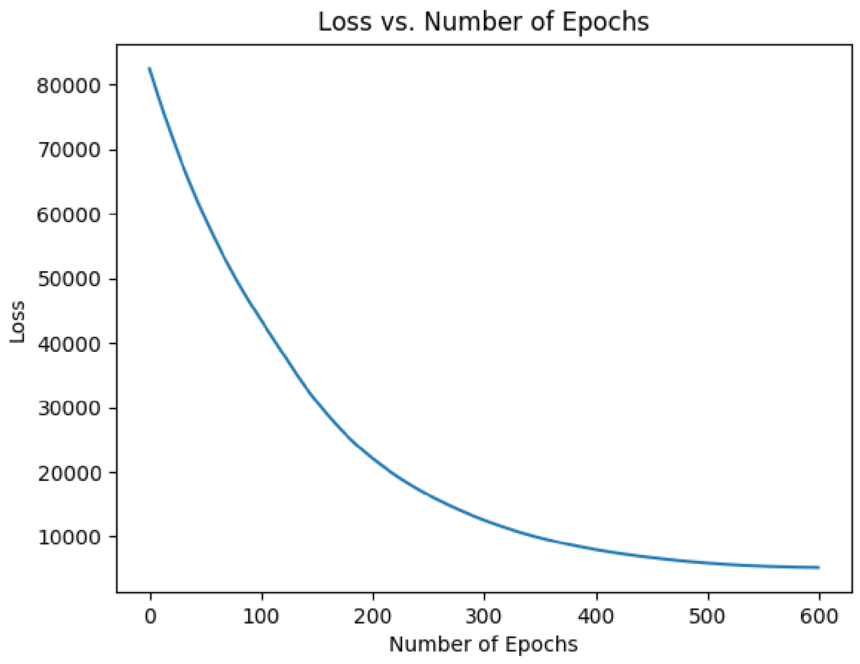


Figure 2: Loss vs. Number of Epochs

3. *Run the algorithm with K = 1; 2; 3; 4; 5 and for each of these values of K, compute and report the percentage of the data points belonging to each of the K clusters. Comment on how many clusters you think is "best" and why? (To answer this, it may be helpful discuss this value in the context of a 2D scatter plot of the data.) Include the 2D scatter plot of data points colored by their cluster assignments. [3 pt.]*

SOLUTION:

The results of running the algorithm with K = 1, 2, 3, 4, and 5 including the minimum loss, percentage of data points belonging to each of the K clusters, and the 2D scatter plot of data points per cluster can be found in figures **Figure 3**, **Figure 4**, **Figure 5**, **Figure 6**, and **Figure 7** below.

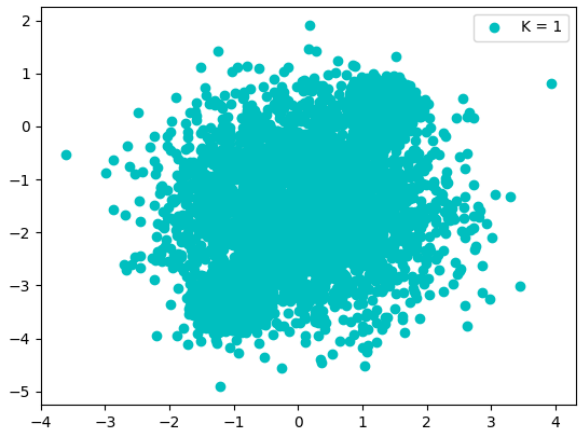


Figure 3: 2D Scatter Plot for K = 1

Minimum Loss: 38453.46875.

Percentage of Data Points in Each Cluster:

{1: 100.00%}.

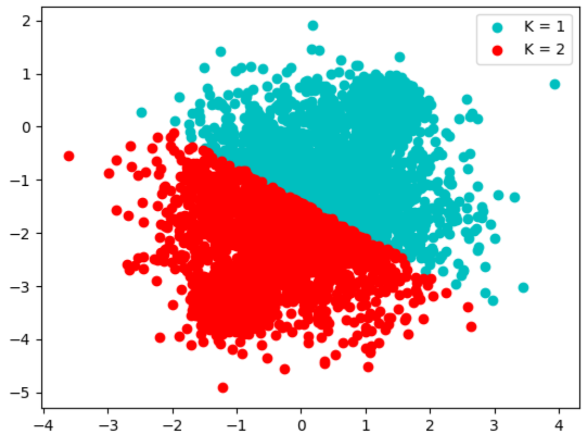


Figure 4: 2D Scatter Plot for K = 2

Minimum Loss: 9203.34570.

Percentage of Data Points in Each Cluster:

{1: 49.55%, 2: 50.45%}.

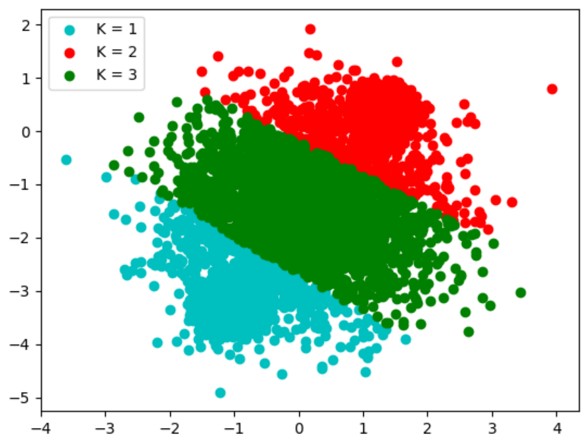


Figure 5: 2D Scatter Plot for K = 3

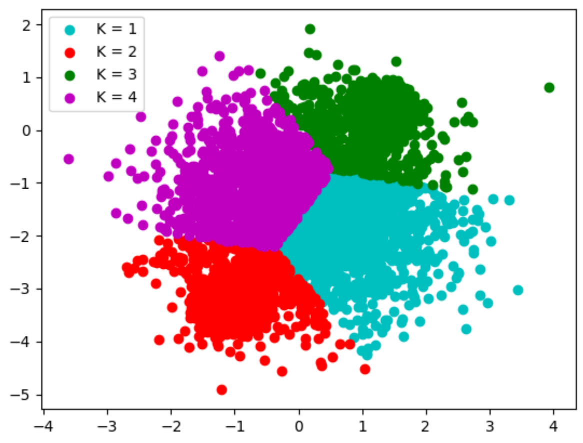


Figure 6: 2D Scatter Plot for K = 4

Minimum Loss: 5117.37451.
Percentage of Data Points in Each Cluster:
{1: 38.34%, 2: 37.96%, 3: 23.7%}.

Minimum Loss: 3374.03613.
Percentage of Data Points in Each Cluster:
{1: 13.53%, 2: 37.13%, 3: 37.31%, 4: 12.03%}.

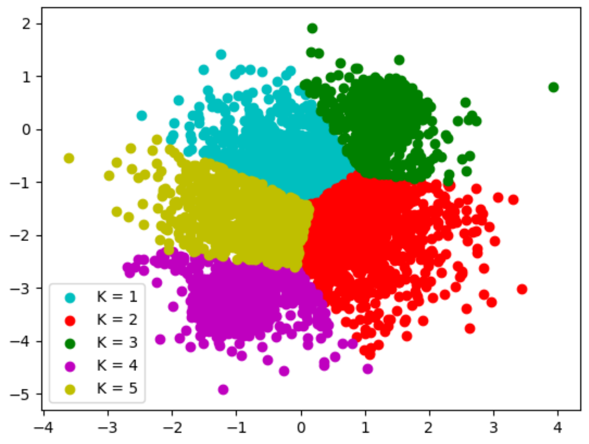


Figure 7: 2D Scatter Plot for K = 5

Minimum Loss: 2874.00903.
Percentage of Data Points in Each Cluster:
{1: 7.46%, 2: 11.72%, 3: 35.89%, 4: 36.35%, 5: 8.58%}.

From the results reported in figures **Figure 3**, **Figure 4**, **Figure 5**, **Figure 6**, and **Figure 7**, we think K = 3 clusters is the best out of the options K = 1, 2, 3, 4, 5. Although K = 5 clusters has the lowest loss out of all the other values of K, when K > 3, the percentage of data points in each cluster no longer stays balanced. For instance, we can see that for K greater than values of 3, certain clusters accumulate significantly more data than others. Therefore, K = 3 is the best out of the options, because it has the lowest loss while still keeping the percentage of clusters balanced.

4. *Hold 1/3 of the data out for validation. For each value of K above, cluster the training data and then compute and report the loss for the validation data. How many clusters do you think is best? [2 pt.]*

SOLUTION:

Holding 1/3 of the data out for validation, using 600 epochs, and a learning rate of 0.01, the validation loss for each value of K from 1 to 5, inclusive, can be fond in **Table 1** below.

Table 1: Table of Validation Losses for K = 1, 2, 3, 4, 5

K	Validation Loss
1	12969.2
2	3066.0
3	1693.23
4	1100.41
5	935.919

Based off of validation losses alone, we can observe that 5 clusters is the best as the validation loss for K = 5 is the lowest.

2 Mixtures of Gaussians [20 pt.]

2.1 The Gaussian Cluster Model [8 pt.]

1. *Derive the expression for the latent variable posterior distribution of a data point $P(z|x)$ in terms of the MoG parameters, $\{\boldsymbol{\mu}^k, \sigma^k, \pi^k\}$. [3 pt.]*

SOLUTION:

Using the Baye's Theorem, we can express the latent variable posterior distribution of a data point as follows:

$$P(z|x) = \frac{P(x|z)P(z)}{P(x)}$$

Applying marginalization, we can rewrite the above equation as follows:

$$P(z|x) = \frac{P(x|z)P(z)}{\sum_{z=1}^{K} P(x,z)}$$

where $P(z) = \prod_{k=1}^{K} \pi_k^{z_k}$ and $P(x|z) = \prod_{k=1}^{K} \mathcal{N}(x|\mu_k, \sigma_k^2)^{z_k}$ whose product is equivalent to $\prod_{k=1}^{K}[\pi_k \mathcal{N}(x|\mu_k, \sigma_k^2)]^{z_k}$.

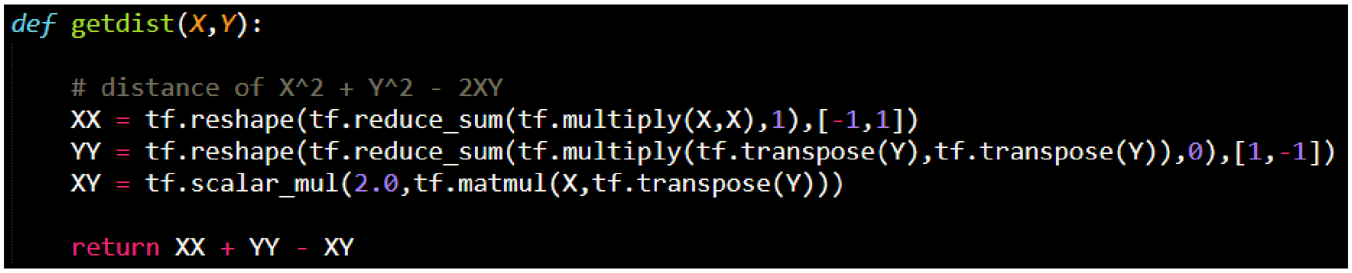
Therefore, we can express $P(z|x)$ in terms of MoG parameters $\{\boldsymbol{\mu}^k, \sigma^k, \pi^k\}$ as follows:

$$P(z|x) = \frac{\prod_{k=1}^{K}\left[\pi^k \mathcal{N}\left(x|\mu^k, {\sigma^k}^2\right)\right]^{z_k}}{\sum_{k=1}^{K} \pi^k \mathcal{N}\left(x|\mu^k, {\sigma^k}^2\right)}$$

2. *Modify the K-means distance function we derived above to compute the log probability density function for cluster k: $\log \mathcal{N}\left(x; \boldsymbol{\mu}^k, {\sigma^k}^2\right)$ for all pair of B data points and K clusters. Include the snippets of the Python code [2 pt.]*

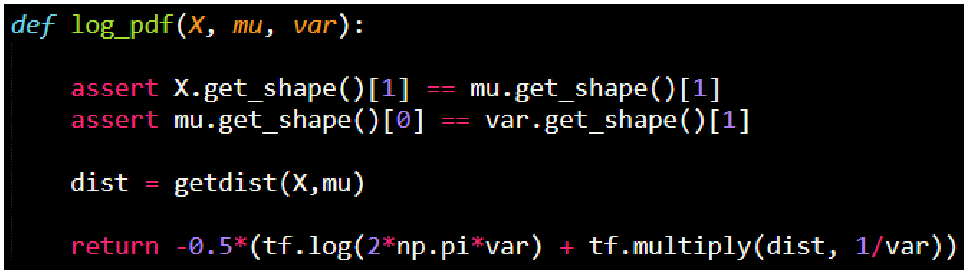
SOLUTION:

The snippet of the code to compute the log probability density function for cluster k: $\log \mathcal{N}\left(x; \boldsymbol{\mu}^k, {\sigma^k}^2\right)$ for all pairs of B data points and K clusters can be found in **Figure 9** below. To compute the log probability density function, we pass in x, μ, and σ. We then use the **`getdist`** function which is shown in **Figure 8** to calculate the squared Euclidean distance between x and μ. The **`log_pdf`** function returns the sum of the product of x-μ distance and $\frac{1}{\sigma}$ and $-\frac{1}{2}\log(2\pi\sigma)$.



```python
def getdist(X,Y):

    # distance of X^2 + Y^2 - 2XY
    XX = tf.reshape(tf.reduce_sum(tf.multiply(X,X),1),[-1,1])
    YY = tf.reshape(tf.reduce_sum(tf.multiply(tf.transpose(Y),tf.transpose(Y)),0),[1,-1])
    XY = tf.scalar_mul(2.0,tf.matmul(X,tf.transpose(Y)))

    return XX + YY - XY
```

Figure 8: `getdist` function



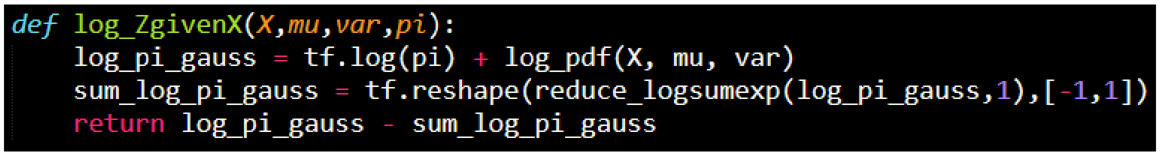
```python
def log_pdf(X, mu, var):

    assert X.get_shape()[1] == mu.get_shape()[1]
    assert mu.get_shape()[0] == var.get_shape()[1]

    dist = getdist(X,mu)

    return -0.5*(tf.log(2*np.pi*var) + tf.multiply(dist, 1/var))
```

Figure 9: Snippet of Python Code for Computing Log Probability Density Function

3. *Write a* *vectorized* *Tensor flow Python function that computes the log probability of the cluster variable z given the data vector* x: $P(z|x)$. *The log Gaussian pdf function implemented above should come in handy. The implementation should use the provided utils.logsumexp function. Include the snippets of the Python code and comment on why it is important to use the log_sum_exp function instead of using tf.reduce sum. [3 pt.]*

SOLUTION:

The snippet of the code to compute the log probability of the cluster variable *z* given the data vector *x:* $P(z|x)$ can be found in **Figure 10** below. To compute this, we calculate the sum of $\log(\pi)$, $\log \mathcal{N}\left(x; \boldsymbol{\mu}^k, {\sigma^k}^2\right)$, and the negative of the *reduce_logsumexp*.



```python
def log_ZgivenX(X,mu,var,pi):
    log_pi_gauss = tf.log(pi) + log_pdf(X, mu, var)
    sum_log_pi_gauss = tf.reshape(reduce_logsumexp(log_pi_gauss,1),[-1,1])
    return log_pi_gauss - sum_log_pi_gauss
```

Figure 10: Snippet of Python Code for Computing Log Probability of z given x

It is possible to use the *tf.reduce_sum* function by computing the Gaussian PDF instead of the log PDF, multiplying it by π, then applying the *reduce_sum* across the rows, and taking the log of each resulting column. The advantage of using the *log_sum_exp* function is its ability to increase accuracy. Summing exponential functions can result in numerical issues. The method is to subtract the max from each data point and add it back later to guarantee that all exponents will have a negative exponent; this means that the largest value that can be exponentiated is zero. Unlike the *tf.reduce_sum* function which can lead to underflow/overflow issues from using very small or very large numbers, the *log_sum_exp* function will prevent this. The *log_sum_exp* function will result in summing together numbers less than or equal to 1, which will not prevent any potential overflow.

2.2 Learning the MoG [12 pt.]

1. *Direct gradient-based optimization appears to learn the MoG parameters without inferring the cluster assignment variables, that is, without computing $P(z|x)$. In fact, this inference is implicit in the gradient computation. Show that for a single training example, the gradient of the marginal log likelihood function is the expected gradient of the log joint probability under its posterior distribution, $\nabla_\mu \log P(x) = \sum_k P(z=k|x)\,\nabla_\mu \log P(x, z=k)$. [2 pt.]*

SOLUTION:

We can use the following chain rule property: $\nabla \log f(x) = \frac{\nabla f(x)}{f(x)}$.

For the left side, we have

$$\nabla_\mu \log P(x) = \frac{\nabla_\mu P(x)}{P(x)}$$

For the right side we can apply the same property as the first step to get the following:

$$\sum_k P(z=k|x)\,\nabla_\mu \log P(x, z=k) = \sum_k P(z=k|x)\frac{\nabla_\mu P(x, z=k)}{P(x, z=k)}$$

Using Bayes' Theorem, we can rewrite the first term as:

$$P(z=k|x) = \frac{P(z=k, x)}{P(x)} \equiv \frac{P(x, z=k)}{P(x)}$$

The right side can therefore be written as:

$$\sum_k \frac{1}{P(x)}\nabla_\mu P(x, z=k)$$

Using Bayes' Theorem again, we can rewrite $P(x, z=k) \equiv P(z=k, x)$ to get the following expression:

$$\frac{1}{P(x)}\sum_k \nabla_\mu P\,(z=k|x)P(z=k)$$

Using the linearity property of the gradient operator, ∇, we can put the summation inside as follows:

$$\frac{1}{P(x)}\nabla_\mu \sum_k P\,(z=k|x)P(z=k)$$

Using Total Probability Theorem, the summation can be simplified to $P(x)$. Then the expression becomes

$$\frac{1}{P(x)}\nabla_\mu P(x) = \frac{\nabla_\mu P(x)}{P(x)} \equiv \nabla_\mu \log P(x)$$

Therefore, for a single training example, the gradient of the marginal log likelihood function is the expected gradient of the log joint probability under its posterior distribution, i.e.

$$\nabla_\mu \log P(x) = \sum_k P(z=k|x)\,\nabla_\mu \log P(x, z=k)$$

2. *Implement the loss function using log-sum-exp function and perform MLE by directly optimizing the log likelihood function using gradient descent in Tensor flow. Note that the standard deviation has the constraint of $\sigma \in [0, \infty)$. One way to deal with this constraint is to replace σ^2 with $\exp(\phi)$ in the math and the software, where ϕ is an unconstrained parameter. In addition, π has a simplex constraint, that is $\sum_k \pi^k = 1$. We can again replace this constrain with unconstrained parameter ψ through a softmax function $\pi^k = \exp(\psi^k) / \sum_{k'} \exp(\psi^{k'})$. A log-softmax function is provided for convenience, utils.logsoftmax. For the dataset data2D.npy, set K = 3 and report the best model parameters it has learnt. Include a plot of the loss vs the number of updates. [6 pt.]*

SOLUTION:

The learning rate was tuned by observing the loss over 600 epochs for the following learning rates: $\eta = \{0.005, 0.004, 0.003\}$. The plot showing the loss history for the different learning rates can be found in **Figure 11**. From the plot, it was observed the optimal learning rate is around 0.005.

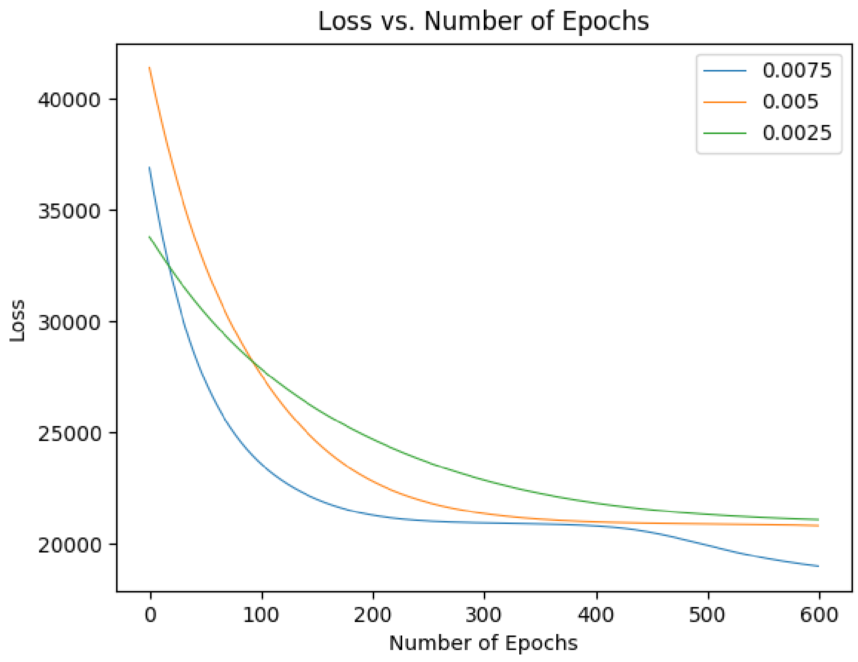


Figure 11: Loss vs. Number of Epochs for Various Learning Rates

Using the Adam optimizer with the hyperparameters: beta1 = 0.9, beta2 = 0.99, and epsilon = 1e-5, and using 600 epochs, the plot showing loss vs. number of epochs can be found below in **Figure 12**. The minimum loss was 5187.50488281.

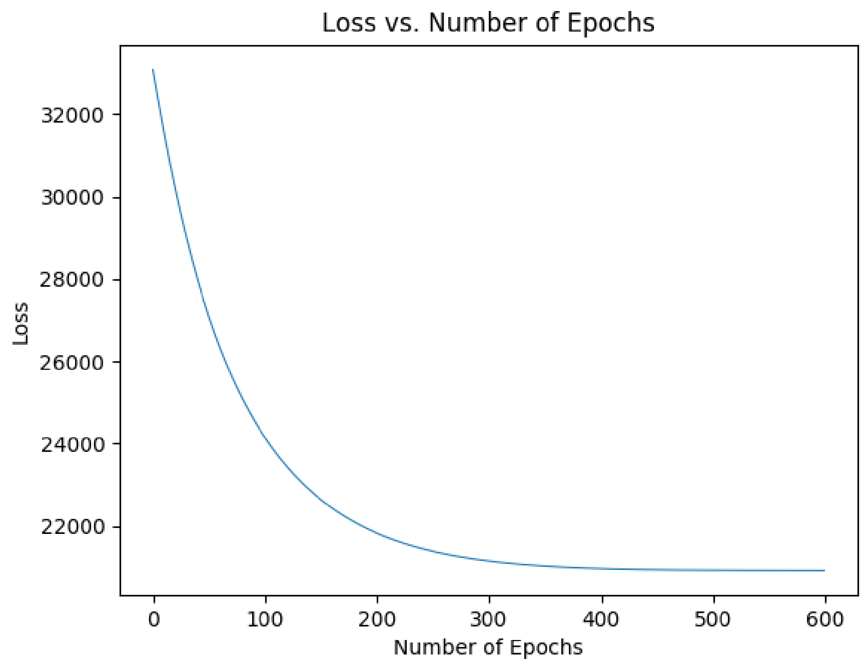


Figure 12: Loss vs. Number of Epochs

3. *Hold out 1/3 of the data for validation and for each value of K = 1; 2; 3; 4; 5, train a MoG model. For each K, compute and report the loss function for the validation data and explain which value of K is best. Include a 2D scatter plot of data points colored by their cluster assignments. [2 pt.]*

SOLUTION:

The results of running the algorithm with K = 1, 2, 3, 4, and 5 including the minimum loss, number of data points belonging to each of the K clusters, and the 2D scatter plot of data points colored by their cluster assignments can be found in figures, **Figure 13**, **Figure 14**, **Figure 15**, **Figure 16**, and **Figure 17**, below.

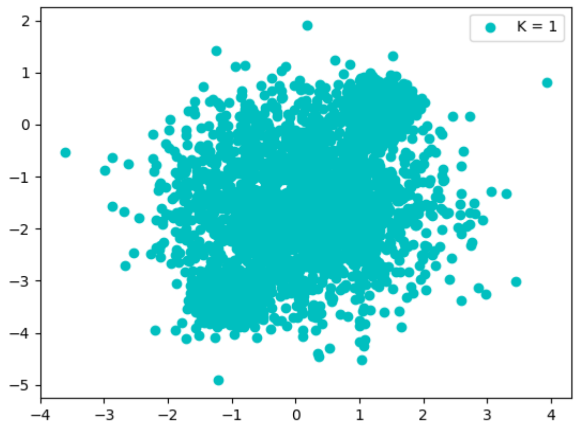


Figure 13: 2D Scatter Plot for K = 1

Minimum Loss: 6998.03369141.

Number of Data Points in Each Cluster:
{1: 6666}.

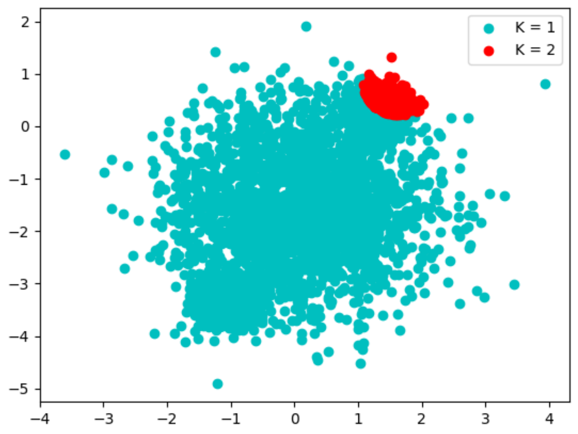


Figure 14: 2D Scatter Plot for K = 2

Minimum Loss: 6856.33007812.

Number of Data Points in Each Cluster:
{1: 5804, 2: 862}.

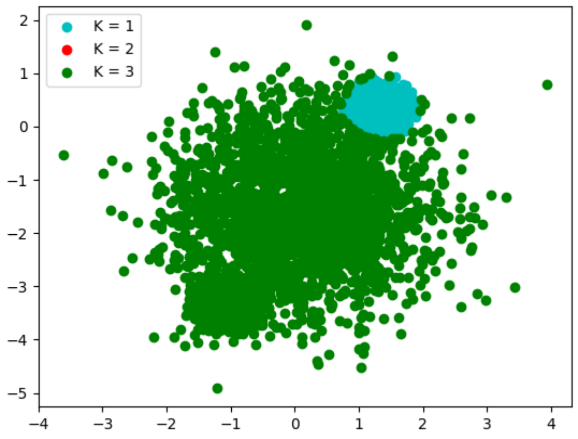


Figure 15: 2D Scatter Plot for K = 3

Minimum Loss: 6355.3
88671875.

Number of Data Points in Each Cluster:
{1: 2249, 2: 0, 3: 4417}.

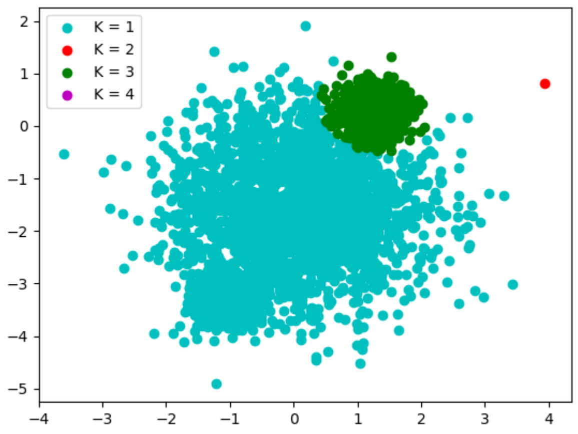


Figure 16: 2D Scatter Plot for K = 4

Minimum Loss: 6476.31347656.

Number of Data Points in Each Cluster:
{1: 4329, 2: 1, 3: 2336, 4:0}.

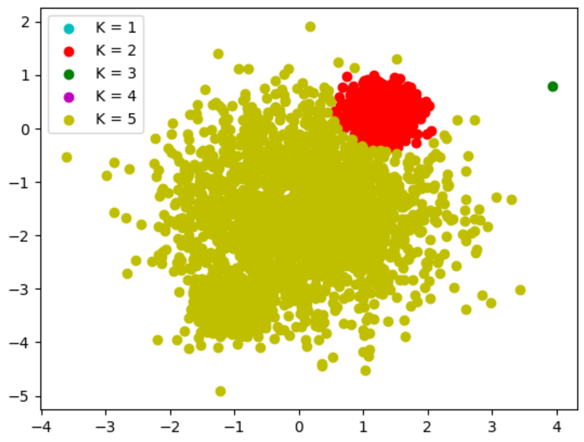


Figure 17: 2D Scatter Plot for K = 5

Minimum Loss: 6211.66503906.

Number of Data Points in Each Cluster:
{1: 0, 2: 2313, 3: 1, 4: 0, 5: 4352}.

From the results reported in figures **Figure 13**, **Figure 14**, **Figure 15**, **Figure 16**, and **Figure 17**, we think K = 2 clusters is the best out of the options K = 1, 2, 3, 4, 5. Although K = 5 clusters has the lowest loss out of all the other values of K, for K values greater than 2, the number of data points in each cluster no longer stays balanced. For K > 2, there is at least one cluster that either does not contain any data points or contains just one out of the 6666 data points. Therefore, K = 2 is the best out of the options, because it has a relatively low loss while still keeping clusters with a reasonable number of data points.

4. *Run both the K-means and the MoG learning algorithms on data100D.npy. Comment on how many clusters you think are within the dataset and compare the learnt results of K-means and MoG. [2 pt.]*

SOLUTION:

A table of validation losses for both K-means and MoG learning algorithms for values of K ranging from 1 to 10 can be found in **Table 2** below. From the results obtained, we can observe that a reasonable number of clusters within this dataset is 8. The validation loss for K-means drops significantly after K = 7, and plateaus around there onwards. The validation loss for MoG is the lowest for K = 7 / 8, and starts increasing back up after K = 8. Therefore, the most reasonable value for K from the results given below is 8.

Table 2: Table of Training and Validation Losses for K-means and MoG for K = {1, …, 10}

K	Training Loss (K-means)	Training Loss (MoG)	Validation Loss (K-means)	Validation Loss (MoG)
1	667063.375	56883.9101563	332982.0	28344.4
2	516923.25	42725.4023438	258126.0	21271.7
3	384331.4375	41805.2617188	191249.0	20978.6
4	234122.03125	35723.6601562	118777.0	17890.9
5	234122.0	38534.40625	118777.0	19185.9
6	289264.0625	35873.8085938	145581.0	18023.9
7	245925.796875	30589.7792969	124109.0	15268.2
8	143452.21875	33090.8671875	71848.9	16514.9
9	141842.359375	37052.2773438	71086.8	18374.9
10	143398.671875	36881.3945312	71849.2	18387.3

3 Discover Latent Dimensions

3.1 Factor Analysis [Bonus: 6 pt.]

1. *Deriving the marginal log likelihood of the factor analysis model for a single training example is a Gaussian distribution with the following mean and covariance matrix:* $\log P(x) = \log \int_s P(x|s)\, P(s) ds = \mathcal{N}(x; \boldsymbol{\mu}, \Psi + WW^T)$*. (You may directly quote the multivariate Gaussian results at the end of this handout.) [1 pt.]*

SOLUTION:

$$P(x_n) = P(x_n; s_n)P(s_n)$$

$$P(s_n) = N(s_n; 0, I)$$

$$P(x_n; s_n) = N(x_n)$$

In the assignment handout identities, we set the following:

$$x = s_n; y = x_n; \Lambda = I; \mu = 0; A = W; b = \mu; L^{-1} = \Psi$$

Therefore,

$$P(x_n) = N(x_n; W(0) + \mu, \Psi + W(I)^{-1}W^T) = N(x_n; \boldsymbol{\mu}, \Psi + WW^T)$$

Given that there is only one training example, B = 1. As seen above, the new expression for$P(x_n)$also does not contain s_n. Therefore,

$$logP(x) = log \prod_{n=1}^{1} \int_{s_n} P(x_n; s_n)\, P(s_n) ds_n = logN(x; \boldsymbol{\mu}, \Psi + WW^T)$$

2. *Write a TensorFlow implementation that learns Factor Analysis models by directly maximizing the log likelihood function. Namely, we would like to adapt the weight matrix, the mean of the data and the data covariance matrix by maximizing the marginal log likelihood:*

$$\max_{W,\boldsymbol{\mu},\Psi} \sum_{n=1}^{B} \log P(x_n)$$

Note that for the determinant of the covariance matrix, a numerical stable implementation is to use a Cholesky decomposition that is $\log\det\{A\} = \sum_i \log diag\{L\}_i^2$ *and L is the Cholesky factor. This trick can be implemented in TensorFlow as:*

```
log_det = 2.0 * tf.reduce_sum(tf.log(tf.diag_part(tf.cholesky(A))))
```

For the tiny hand-written digits dataset containing two classes "3" and "5" tinymnist.npy, train a factor analysis model by setting the number of latent dimension K = 4 and report training, validation and test marginal log likelihood. Plot each row of the learnt weight matrix as a set of 8x8 images similar to the neural network visualization in assignment 2. Comment on the visualization and discuss what kind of latent dimensions factor analysis has discovered from the dataset. (You would like to discuss what kind of variability has the weight matrix captured about the handwritten digits of "3" and "5", e.g. one latent dimension is used to model the variability of the top part of those digits.) [3 pt.]

SOLUTION:

The plots for each row of the learnt weight matrix can be found in the figures: **Figure 18**, **Figure 19**, **Figure 20**, and **Figure 21**.

As the classification problem is binary, it only requires a single filter to distinguish between the '3' and the '5' handwritten digits. This explains why all 4 rows resemble each other. Additionally, the effort is focused on the top right corner of the image and the far left of the image, which is where the two digits differ the most. The bottom half of '3' and '5' are very similar, which is why the bottom half of all the filters have relatively low weights.

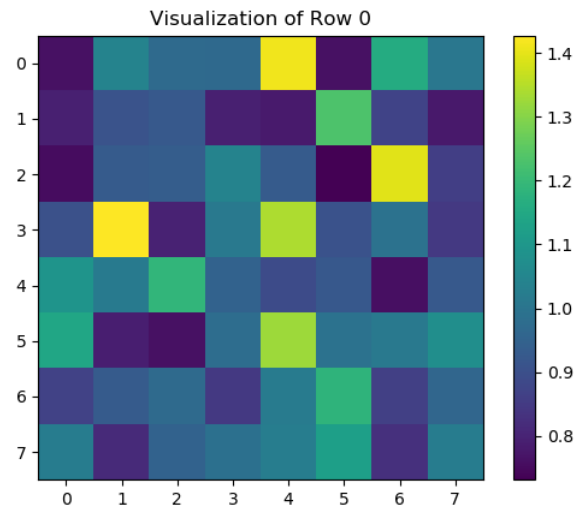


Figure 18: Visualization of Row 0

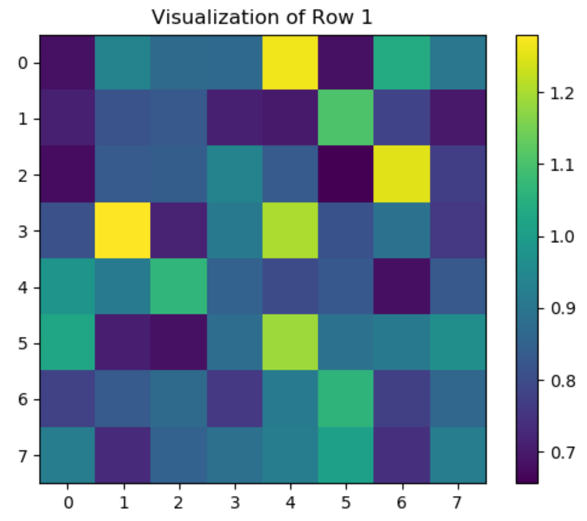


Figure 19: Visualization of Row 1

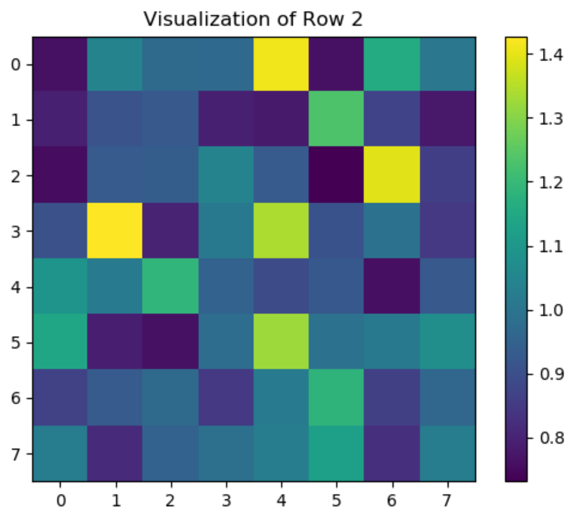


Figure 20: Visualization of Row 2

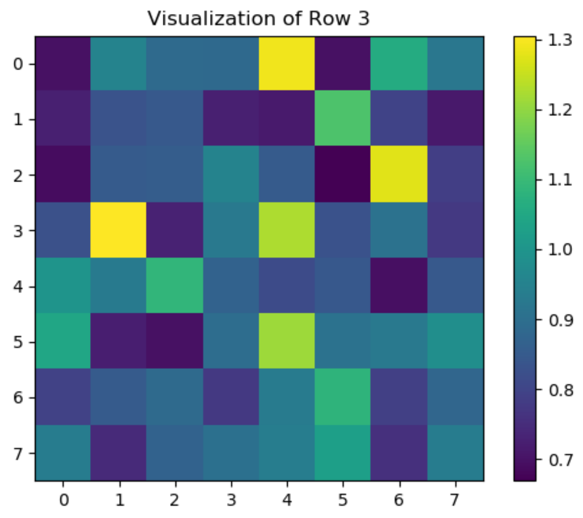


Figure 21: Visualization of Row 3

3. *Geoffrey Hinton's explanation on PCA and FA: Generate a toy dataset of 200 3-dimensional data points* $\{x^{(1)}, \ldots, x^{(200)}\}$ *by first generating the latent states s from a 3-D multivariate Gaussian distribution with zero mean and identity covariance matrix* $s \sim \mathcal{N}(s; 0, I)$, $s = \begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix} \epsilon\, \mathbb{R}^3$. *Now transform the latent states to 3-dimensional observations* $x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$ *using the following formula:*

$$x_1 = s_1$$
$$x_2 = s_1 + 0.001 s_2$$
$$x_3 = 10 s_3$$

Use such dataset to train a PCA with a single principle component and a factor analysis model with a single latent dimension. Show that PCA learns the maximum variance direction (i.e. x_3 *direction) while FA learns the maximum correlation direction (i.e.* $x_1 + x_2$ *direction). [2 pt.]*

SOLUTION:

The figures illustrating the visualization of FA and visualization of PCA can be found in the figures: **Figure 22** and **Figure 23** below.

As expected, Factor Analysis places weight on the $x_1 + x_2$, whereas PCA weighs heavily on x_3.

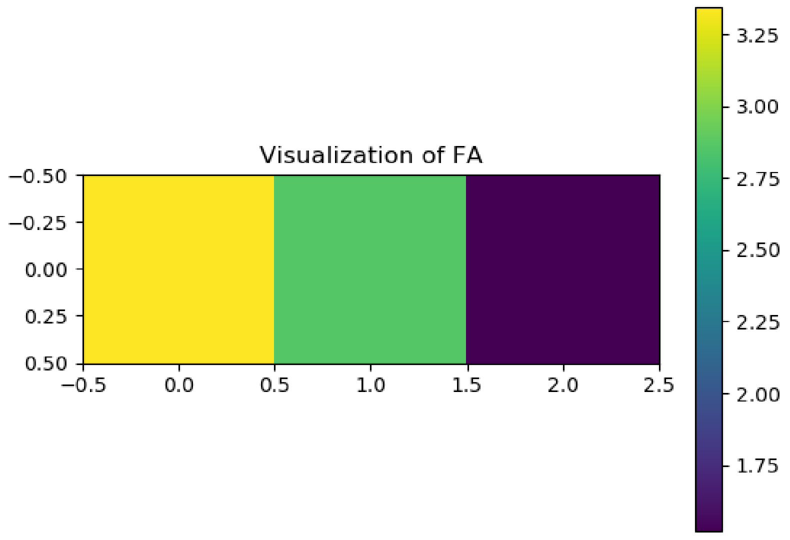


Figure 22: Visualization of FA

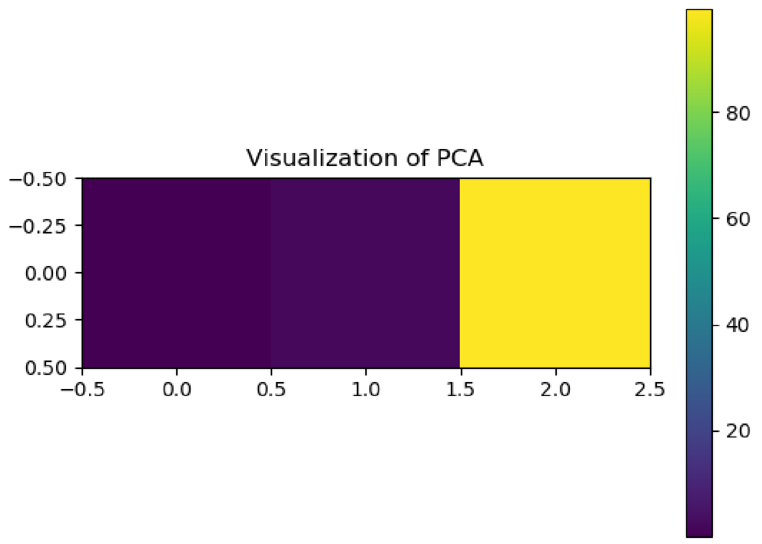


Figure 23: Visualization of PCA