# CSE 474: Project 3

Peter M. VanNostrand
Department of Computer Science
University at Buffalo
Buffalo, NY 14260
*pmvannos@buffalo.edu*

**Abstract**

In this project we developed three machine learning models. The first is a simple k-means classifier written using the sklearns library. The second is a k-means classifier with autoencoder written in sklearns and Keras, and the last is a Gaussian Mixture Model classifier with autoencoder also written also written using sklearns and Keras. All three networks perform 10 class classification and were trained and tested on the MNIST Fashion dataset.

## 1 Introduction

Clustering is a common unsupervised learning task in practical data analysis, and is particularly useful as it does not require the existence of a large prelabeled dataset for training. The simplest form of clustering is k-means clustering, where for every datapoint the Euclidean distance between that point and all other points in the dataset is computed, and then those points that are closest together are grouped into clusters. This is simple to implement and works well for some datasets, but struggles on very high dimensional datasets and those with non-spherical clusters. An alternative approach to clustering is the Gaussian Mixture Model, in the method the dataset is used to compute a multi-dimensional probability space and then samples are probabilistically grouped into clusters based upon their proximity to previous samples. This allows for soft clustering, but still struggles with high dimensional datasets. To reduce the dimensionality of data used in both techniques we can employ the use of an autoencoder. Autoencoders are essentially neural networks which attempt to take an input, compress it into a small representation of its key features and then uncompress that representation to regenerate the input. By using the smaller feature set generated in the middle of this process as the input to our clustering algorithms we can reduce the dimensionality of the input data, potentially allowing the clustering algorithms to run more quickly and accurately.

## 2 Dataset

For this exercise the MNIST Fashion dataset was used for all training and validation. The dataset is comprised of 70,000 28x28 pixel grayscale images of 10 different types of clothing. Each picture has a corresponding integer label from 0-9 which correspond to image descriptions as listed in Table 1

Table 1: MNIST Fashion Labels

| **Label** | **Description** |
|---|---|
| 0 | T-shirt/top |
| 1 | Trouser |
| 2 | Pullover |
| 3 | Dress |
| 4 | Coat |
| 5 | Sandal |
| 6 | Shirt |
| 7 | Sneaker |
| 8 | Bag |
| 9 | Ankle boot |

As the images are 28x28 greyscale pixels there is only one channel in the image with pixel intensities ranging from 0-255, Therefore there are $28 \times 28 \times 1 = 784$ features per image.

## 2 Preprocessing

In all three algorithms implemented in this work the Fashion-MNIST dataset is used for training and validation. First the dataset is imported using the provided keras function "fashion_mnist.load_data()"

This function returns a pair of arrays, each pair contains an array of images represented as a $n_{samples} \times 28 \times 28 \times 1$ matrix and a second array of $n_{samples}$ integers which represent the correct labels for the corresponding images. The first returned pair contains 60,000 images and labels for training and the second pair contains 10,000 images and labels for images.

As each image in the dataset is represented by an array of pixel values ranging from [0,255] we divide both the training samples and testing samples by 255.0 to normalize these values to [0,1].

Depending on the algorithm used further preprocessing may be performed as described below.

### 2.1 Part 1: k-Means

In this part, once the dataset is imported and normalized we then rearrange the image representations to prepare the inputs for simple k-means classification. For ease we reshape the training and testing images to an array of size $n_{samples} \times 784$ such that each row in the array represents on sample. At this point the data is ready for the k-means classification

### 2.2 Part 2: k-Means with Autoencoder

In this part no further preprocessing is performed after the process described above.

### 2.3 Part 3: GMM with Autoencoder

In this part no further preprocessing is performed after the process described above.

## 3 Architecture

### 3.1 Part 1: k-Means

For part one a simple k-Means algorithm is used from the sklearn library, using the sklearn.cluster.kmeans function the input data is classified into k=10 clusters. As I have been working on a similar task in another class I have also included a full implementation of k-means that I wrote from scratch in Python and adapted for use on this dataset. The custom implementation and sklearn implementation perform the same process and produce similar results, although all further discussion of the k-means algorithm will use sklearn's implementation for consistency.

### 3.2 Part 2: k-Means with Autoencoder

To reduce the dimensionality of the data used for clustering I implemented a convolutional autoencoder consisting of an encoder and a decoder. The encoder has the following layers

- Convolutional2D layer, relu activation
- Maxpooling2D layer
- Convolutional2D layer, relu activation
- Maxpooling2D layer
- Convolutional2D layer, relu activation

These layers were selected as they reduce the dimensionality of the data resulting in a "compressed" representation of the input sample with only 128 dimensions instead of the original 784 dimensions. The ReLU activation function is used in hidden layers to avoid gradient vanishing. The following layers makeup the decoder

- Convolutional2D layer, relu activation
- UpSampling2D
- Convolutional2D layer, relu activation
- UpSampling2D
- Convolutional2D layer, relu activation
- UpSampling2D
- Convolutional2D layer, relu activation

These layers were selected as they increase the dimensionality of the data again to produce an output similar to the original input sample. The ReLU activation function is used in hidden layers to avoid gradient vanishing.

### 3.3 Part 3: GMM with Autoencoder

Part 3 uses the convolutional autoencoder from part 3 to reduce the dimensionality of the input data from 784 dimensions to 128 dimensions. Then the sklearn Generate Mixture Model function is used to perform clustering with ten components. The resulting Generative Mixture Model is used to classify both the training and testing data.

## 4 Results

### 4.1 Part 1

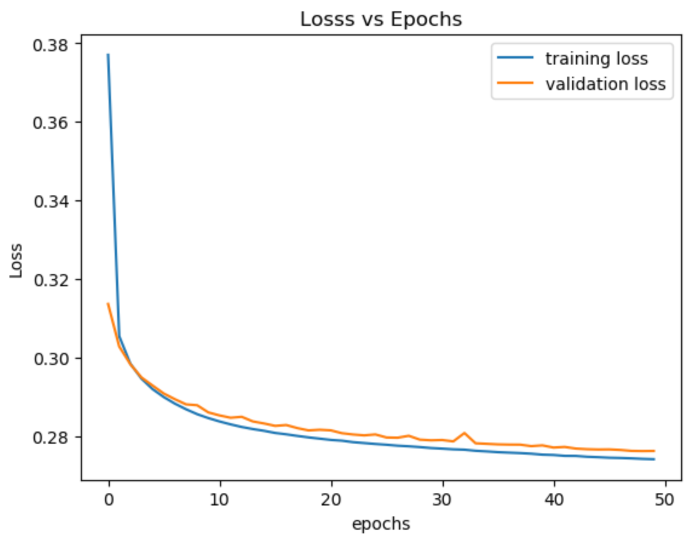
Running our simple neural network on the full 784-dimension input data resulted in a training accuracy of 34.76% and a validation accuracy of 34.90%. For clustering I also chose to measure the purity of each determined cluster. That is the ratio of nodes correctly identified to all nodes in that cluster. The purity for simple k-Means is shown below.

Table 2: Simple k-Means Purity

| Cluster | Training | Testing |
|---:|---:|---:|
| 0 | 0.2778 | 0.2777 |
| 1 | 0.9390 | 0.9193 |
| 2 | 0.7161 | 0.7296 |
| 3 | 0.4605 | 0.4651 |
| 4 | 0.6873 | 0.6911 |
| 5 | 0.3740 | 0.3878 |
| 6 | 0.9015 | 0.8981 |
| 7 | 0.5960 | 0.6050 |
| 8 | 0.9524 | 0.9555 |
| 9 | 0.4852 | 0.5192 |

## 4.2 Part 2

Below we can see a plot of the training and validation accuracy during the training of our convolutional autoencoder



This shows us that our model converges well but not perfectly. After approximately 20 epochs we see drop-off in the accuracy gain for each additional epoch so I chose to terminate the training at 50 epochs which is well into the linear region, but before the training and validation losses diverge which would indicate model overfitting.

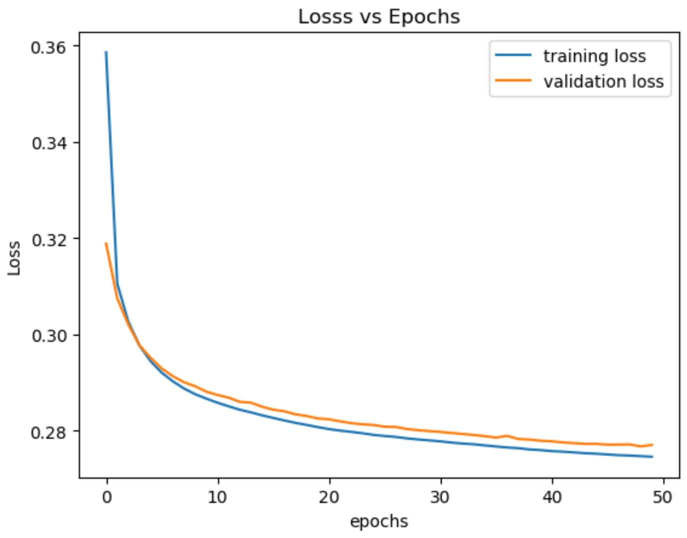
Applying k-means to the output of our encoder produced a training accuracy of 35.24% and a validation accuracy of 34.64%. The cluster purities for this case are shown below.

Table 3: Autoencoder k-Means Purity

| Cluster | Training | Testing |
|---|---|---|
| 0 | 0.0026 | 0.0024 |
| 1 | 0.3525 | 0.3464 |
| 2 | 0.7270 | 0.7117 |
| 3 | 0.9595 | 0.9619 |
| 4 | 0.5011 | 0.5105 |
| 5 | 0.4143 | 0.3926 |
| 6 | 0.9481 | 0.9256 |
| 7 | 0.5585 | 0.5634 |
| 8 | 0.5327 | 0.5396 |
| 9 | 0.4473 | 0.4551 |
| 10 | 0.5856 | 0.5805 |
| 11 | 0.2762 | 0.2725 |

## 4.3 Part 3

Below we can see a plot of the training and validation accuracy during the training of our convolutional autoencoder. As this is the same autoencoder as in part 2 the losses are essentially identical and the above analysis applies.



This time we applied Gaussian Mixture Modeling rather than k-Means after the autoencoder step. This produce a training accuracy of 40.95% and a validation accuracy of 40.44%. The cluster purity for this technique is shown below.

Table 4: Autoencoder GMM Purity

| Cluster | Training | Testing |
|---|---|---|
| 0 | 0.6305 | 0.6094 |
| 1 | 0.7890 | 0.7731 |
| 2 | 0.6143 | 0.6182 |
| 3 | 0.3684 | 0.3771 |
| 4 | 0.5356 | 0.5472 |
| 5 | 0.2612 | 0.2676 |
| 6 | 0.4609 | 0.4700 |
| 7 | 0.7583 | 0.7663 |
| 8 | 0.4082 | 0.3977 |
| 9 | 0.8404 | 0.8253 |

## 5 Conclusion

In this project we explored the popular MNIST Fashion dataset comprised of images of common pieces of clothing. Then we explored the necessary preprocessing that must be performed on this data to prepare it for use in neural network training. With our preprocessed data we presented clustering methods to group the images into their 10 groups without knowledge of the correct labels for each image. We then used the available labels to measure the accuracy of each model. We observed that while basic k-Means is simple to implement it can benefit slightly from the application of an autoencoder to reduce the input dimensionality. We then saw that for some datasets GMM may be a more accurate approach than k-Means as it is able to accurately cluster data with non-spherical groups.