Comparison of K-Means and CNN as a Feature Creator.

CS 696 Big Data Analytics

Meghana Ravishankar
A25248100 A25248100

Table of contents

Section	Page Numbers
Introduction	3
Methods	4
CNN	4
K-means+ Supervised learning	5
Results and Discussions	6
Conclusion	9
References	9
Appendix	9

Introduction

CNN:

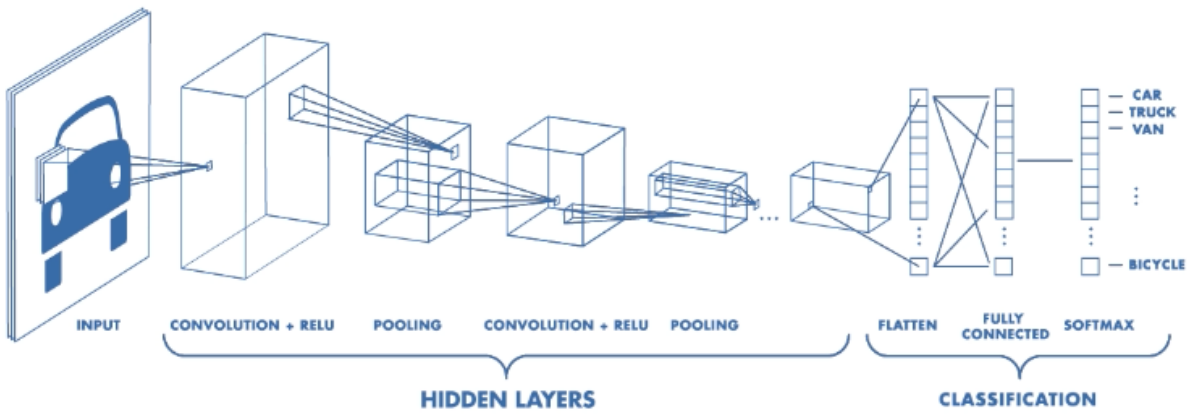

Figure 1: A Typical Convolution Neural Network

CNN (Convolutional Neural Network) is a neural network inspired by working of human brain, it consists of various layers mainly an input layer, and output layer and various hidden layers. The CNN consists of 2 main parts, layers for feature extraction and layers for classification. In the hidden layers/Feature extraction part the network performs a series of convolutions and pooling operations during which the features are detected. For example, If there is a picture of a zebra, this is the part where the network would recognize its stripes, two ears, and four legs. In the classification part the fully connected layers serve as a classifier on top of the extracted features. They also work on assigning a probability for the object on the image being what the algorithm predicts it is.

K-Mean:

K-means clustering aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster. K-means clustering has been used as a feature learning (or dictionary learning) step, in either (semi)supervised learning or unsupervised learning. The basic approach is first to train a k-means clustering representation, using the input training data (which need not be labelled). Then, to project any input datum into the new feature space. We have a choice of "encoding" functions, but we can use for example the thresholded matrix-product of the datum with the centroid locations, the distance from the datum to each centroid, or simply an indicator function for the nearest centroid, or some smooth transformation of the distance. This use of k-means has been successfully combined with simple, linear classifiers for semi-supervised learning in NLP (specifically for named entity recognition) and in computer vision. On an object recognition task, it was found to exhibit comparable performance with more sophisticated feature learning

approaches such as autoencoders and restricted Boltzmann machines. However, it generally requires more data than the sophisticated methods, for equivalent performance, because each data point only contributes to one "feature" rather than multiple.

Feature extraction:

Feature extraction is reducing the amount of resources required to describe a large set of data. When performing analysis of complex data one of the major problems that stems from the number of variables involved is large amount of memory and computation power, also it may cause a classification algorithm to overfit to training samples and generalize poorly to new samples. Feature extraction is a general term for methods of constructing combinations of the variables to get around these problems while still describing the data with enough accuracy. Most common ways to reduce features are PCA, LDA and other statistical methods, this project uses K-Means as a feature extraction method or rather a feature creator.

Selecting or rather having less features to find better accuracy is important as it helps maintain less computation time and power, features are always chosen such that they are different to each other, and similar ones are eliminated. Therefore, using clustering techniques to find similar features and grouping them together to find most dissimilar features is valid.

Methods

Two different methods were used for comparison, one model focused on building a CNN and using it to classify MNIST data and plot the accuracy curve for train and test data. MNIST consist of 60,000 training data and 10,000 test data. The model summary is as given below:

CNN Model 1 Summary

Layer (type)	Output Shape	Param #
conv2d_3 (Conv2D)	(None, 26, 26, 32)	320
conv2d_4 (Conv2D)	(None, 24, 24, 64)	18496
max_pooling2d_2 (MaxPooling2	(None, 12, 12, 64)	0
dropout_3 (Dropout)	(None, 12, 12, 64)	0
flatten_2 (Flatten)	(None, 9216)	0
dense_3 (Dense)	(None, 128)	1179776
dropout_4 (Dropout)	(None, 128)	0

dense_4 (Dense) (None, 10) 1290

Total params: 1,199,882
Trainable params: 1,199,882
Non-trainable params: 0

CNN Model 2 Summary

Layer (type)	Output Shape	Param #
conv2d_3 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_2 (MaxPooling2	(None, 13, 13, 32)	0
dropout_3 (Dropout)	(None, 13, 13, 32)	0
flatten_2 (Flatten)	(None, 5408)	0
dense_3 (Dense)	(None, 128)	692352
dropout_4 (Dropout)	(None, 128)	0
dense_4 (Dense)	(None, 10)	1290

Total params: 693,962
Trainable params: 693,962
Non-trainable params: 0

Model 2: K-means and Supervised Learning

This model focused on building a new dataset by finding new features using K-means. The data underwent preprocessing like normalization and standardization. The data was normalized by dividing it by 255(because it an image dataset) while scikit learn library StandardScalar was used to standardize data. Standardizing helps the K-means algorithms to form better clusters. Unscaled data slow down or even prevent the convergence of many gradient-based estimators. Many estimators are designed with the assumption that each feature takes values close to zero or more importantly that all features vary on comparable scales. Metric-based and gradient-based estimators often assume approximately standardized data (centered features with unit variances). K-Means was used to cluster data into 10 different clusters. The number of clusters were chosen with respect to number of labels, using pre-

clustering methods like elbow will not be helpful as we are using K-Means as a feature extraction method rather than a clustering method, besides Elbow method is extremely time consuming for large dataset like MNIST. After K-Means (Scikit uses K-Means ++ as initialization method by default), the distance of each data point from every 10-cluster form was calculated (Euclidean distance was used), which was then arranged into 10 columns (for each cluster center). The new data was then used to classify using Supervised learning method, K-Nearest Neighbor.

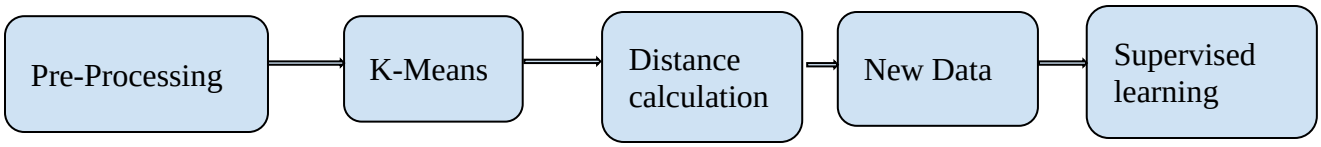

Figure 2: Pipeline for Model 2

Results and Discussion

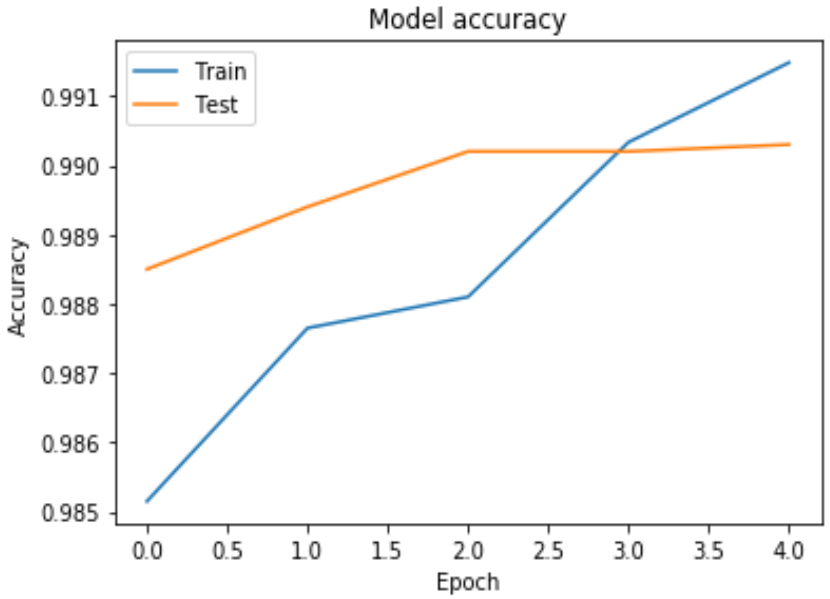

Figure 3: CNN Model-1 accuracy of train and test set for each epoch.

Run time details for Model 1:

Train on 60000 samples, validate on 10000 samples
Epoch 1/5
60000/60000 [==============================] - 185s 3ms/step - loss: 2.1706 - acc: 0.8119 - val_loss: 0.0966 - val_acc: 0.9712
Epoch 2/5
60000/60000 [==============================] - 193s 3ms/step - loss: 0.1219 - acc: 0.9647 - val_loss: 0.0558 - val_acc: 0.9829
Epoch 3/5
60000/60000 [==============================] - 199s 3ms/step - loss: 0.0854 - acc: 0.9756 - val_loss: 0.0446 - val_acc: 0.9871
Epoch 4/5
60000/60000 [==============================] - 201s 3ms/step - loss: 0.0683 - acc: 0.9800 - val_loss: 0.0371 - val_acc: 0.9886
Epoch 5/5
60000/60000 [==============================] - 201s 3ms/step - loss: 0.0588 - acc: 0.9828 - val_loss: 0.0354 - val_acc: 0.9887

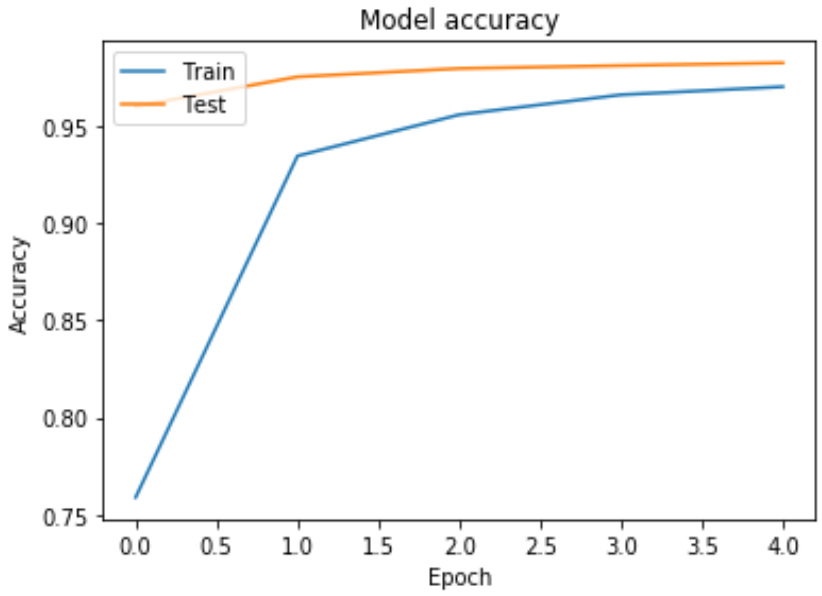

Figure 4: CNN Model-2 accuracy of train and test set for each epoch.

Run time details for Model 2:
Train on 60000 samples, validate on 10000 samples
Epoch 1/5
60000/60000 [==============================] - 66s 1ms/step - loss: 2.7146 - acc: 0.7589 - val_loss: 0.1469 - val_acc: 0.9604
Epoch 2/5
60000/60000 [==============================] - 63s 1ms/step - loss: 0.2386 -

acc: 0.9348 - val_loss: 0.0869 - val_acc: 0.9756
Epoch 3/5
60000/60000 [==============================] - 64s 1ms/step - loss: 0.1532 - acc: 0.9561 - val_loss: 0.0677 - val_acc: 0.9799
Epoch 4/5
60000/60000 [==============================] - 63s 1ms/step - loss: 0.1192 - acc: 0.9663 - val_loss: 0.0604 - val_acc: 0.9814
Epoch 5/5
60000/60000 [==============================] - 63s 1ms/step - loss: 0.1038 - acc: 0.9705 - val_loss: 0.0635 - val_acc: 0.9828

The time taken by the CNN-1 was 18 minutes for 5 epochs and gave 99% accuracy. While CNN-2(CNN-1,without 2nd layer) took around 5 minutes for 5 epochs and gave 97%.

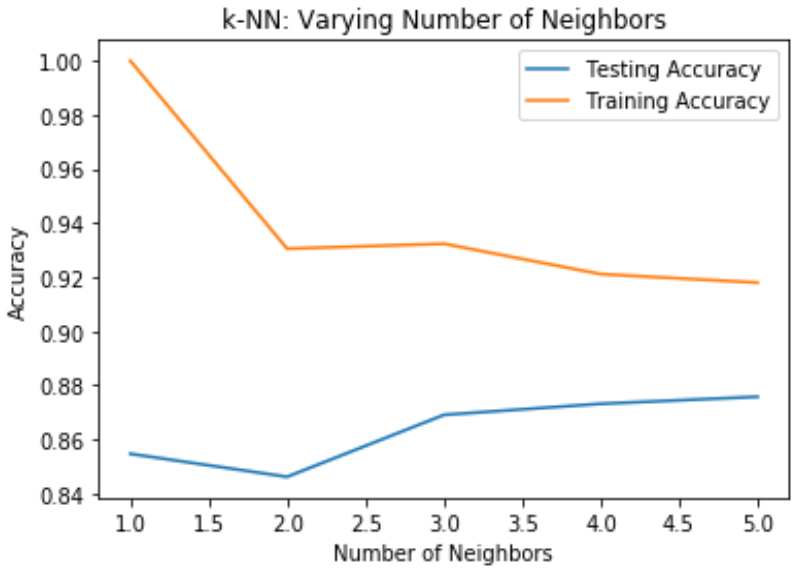

Figure 5: Training and testing accuracy for Model 2(K-means +Supervised Learning)

Training time for K-mean 175.60867285728455
Testing time for K-mean 20.59686303138733
Classification Time 36.00390267372131
[1. 0.93055 0.932366 0.92115 0.918]
[0.8547 0.8462 0.8691 0.8732 0.8758]

The project has further scope, other distance calculations can be used in K-means or maybe different clustering method can be used all together.

Conclusion

It is important to think about dimensionality and feature reduction techniques rather than burdening a neural network to do feature extraction and classification. Geoffrey Hinton believes Neural Networks are very inefficient way of learning. He gives an example of how a diamond shape and square may look same to a convolution neural network, which causes us to use same image in multiple angles. Collection and changing the data and training a network for multiple hours can be avoided if we find good methods to find important features which store most of the information.

Everything is give and take, is it better to use CNN to get good efficiency but unexplained output, or is it better to invest a little more time to find and apply algorithms which give less accuracy but a logical explanation for the output.

References

- https://www.quora.com/How-can-I-use-k-means-for-feature-extraction
- https://www-cs.stanford.edu/~acoates/papers/coatesng_nntot2012.pdf
- https://www.sciencedirect.com/science/article/pii/S0957417413006659
- https://ieeexplore.ieee.org/document/8260799
- https://link.springer.com/chapter/10.1007/11892755_64
- https://medium.freecodecamp.org/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050
- http://moreisdifferent.com/2017/09/hinton-whats-wrong-with-CNNs

Appendix

CNN:

```
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
import numpy as np
from sklearn.cluster import KMeans
import time
batch_size = 128
num_classes = 10
epochs = 5

# input image dimensions
img_rows, img_cols = 28, 28

# the data, split between train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

```
x_train = x_train.reshape(60000,28,28,1)
x_test = x_test.reshape(10000,28,28,1)

print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), #32x28x28
                 activation='relu',
                 input_shape=(28,28,1)))
model.add(Conv2D(64, (3, 3), activation='relu')) #64x28x28
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax')) #10 units
model.summary()
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])

history=model.fit(x_train,                                                    y_train,
          batch_size=batch_size,
          epochs=epochs,verbose=1,
          validation_data=(x_test, y_test))
import matplotlib.pyplot as plt

# Plot training & validation accuracy values
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()

# Plot training & validation loss values
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
```

```
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```

K-Mean:

```
import numpy as np
import pandas as pd
from sklearn.cluster import KMeans
from scipy.spatial import distance
import time
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
#tf.keras.backend.clear_session()
df_train = pd.read_csv(
    filepath_or_buffer=r'C:\Users\meghs\Documents\Big data\Project 3\mnist-in-csv\mnist_train.csv')

df_test=pd.read_csv(
    filepath_or_buffer=r'C:\Users\meghs\Documents\Big data\Project 3\mnist-in-csv\mnist_test.csv')


# Assign features and corresponding labels
y_train = df_train.iloc[:,0].values
x_train = df_train.iloc[:, 1:].values

y_test =df_test.iloc[:,0].values
x_test = df_test.iloc[:, 1:].values

#df_train.head()
print('Training data and labels',x_train.shape,y_train.shape,
      'Test data and labels',x_test.shape,y_test.shape)
#normalize
x_train = x_train/255
x_test = x_test/255

#K_mean for feature extraction_training
start = time.time()
```

```
kmeans_model = KMeans(n_clusters=10, random_state=1).fit(x_train)
labels = kmeans_model.labels_
centers=kmeans_model.cluster_centers_
dist=[]
for j in range(10):
    for i in range(len(x_train)):
        dist.append(distance.euclidean(x_train[i], centers[j]))
        #print(j)
end = time.time()
print ('Training time for kmean',end - start)
#K_mean for feature extraction_test
start_ = time.time()
kmeans_model_ = KMeans(n_clusters=10, random_state=1).fit(x_test)
labels_ = kmeans_model.labels_
centers_=kmeans_model.cluster_centers_
dist_=[]
for j in range(10):
    for i in range(len(x_test)):
        dist_.append(distance.euclidean(x_test[i], centers_[j]))
        #print(j)
end_ = time.time()
#end = time.time()
#print ('Time taken by Kmean to make new features',end - start)
print ('Testing time for kmean',end_ - start_)
#make new data
mnist_train={'feature1': dist[0:60000],'feature2': dist[60000:120000],'feature3':
dist[120000:180000],
                'feature4': dist[180000:240000],'feature5': dist[240000:300000],'feature6':
dist[300000:360000],
                'feature7': dist[360000:420000],'feature8': dist[420000:480000],'feature9':
dist[480000:540000],
           'feature10': dist[540000:600000] }

mnist_new_train = pd.DataFrame(mnist_train)


#make new data
mnist_test={'feature1': dist_[0:10000],'feature2': dist_[10000:20000],'feature3':
dist_[20000:30000],
                'feature4': dist_[30000:40000],'feature5': dist_[40000:50000],'feature6':
dist_[50000:60000],
                'feature7': dist_[60000:70000],'feature8': dist_[70000:80000],'feature9':
dist_[80000:90000],
          'feature10': dist_[90000:100000] }

mnist_new_test = pd.DataFrame(mnist_test)
```

```
mnist_new_train=StandardScaler().fit_transform(mnist_new_train)
mnist_new_test=StandardScaler().fit_transform(mnist_new_test)

#knn

# Setup arrays to store train and test accuracies
neighbors = np.arange(1, 6)
train_accuracy = np.empty(len(neighbors))
train_error = np.empty(len(neighbors))
test_accuracy = np.empty(len(neighbors))
test_error = np.empty(len(neighbors))
#test_accuracy2 = np.empty(len(neighbors))
# Loop over different values of k
s=time.time()
for i, k in enumerate(neighbors):
    # Setup a k-NN Classifier with k neighbors: knn
    knn = KNeighborsClassifier(n_neighbors=k)

    # Fit the classifier to the training data
    knn.fit(mnist_new_train,y_train)


    #Compute accuracy on the training set

    train_accuracy[i] = knn.score(mnist_new_train, y_train)
    #train_error[i] =(1-knn.score(X_train, y_train))*100

    #Compute accuracy on the testing set
    knn.predict(mnist_new_test)
    test_accuracy[i] = knn.score(mnist_new_test, y_test)
    #test_error[i] =(1-knn.score(X_test, y_test))*100
    #knn.predict(X_pca2)
    #test_accuracy2[i] = knn.score(X_pca2, y_target2)
e = time.time()


print ('Time',e - s)


# Generate plot
print(train_accuracy,test_accuracy)
plt.title('k-NN: Varying Number of Neighbors')
plt.plot(neighbors, test_accuracy, label = 'Testing Accuracy')
plt.plot(neighbors, train_accuracy, label = 'Training Accuracy')
```

```
#plt.plot(neighbors, test_accuracy2, label = 'Testing Data2 Accuracy')
plt.legend()
plt.xlabel('Number of Neighbors')
plt.ylabel('Accuracy')
plt.show()
```