CSE 258 Assignment II
Puneeth BommiReddy
PID: A53093725

I. Dataset

Statistics and Properties

The MNIST database contains images of handwritten digits (0 to 9), that is quite popular for image recognition tasks. There are 60,000 images in the train set and 10,000 in the test set. It is available online at http://yann.lecun.com/exdb/mnist/ . The 60,000 original test images are split into two sets, 55,000 images to be still used for testing and, the remaining 5000 images, which will be used as a validation set. This is the default way the dataset is split in Tensorflow's tutorial on MNIST classification[2]. This assignment uses only the training images as we are trying to classify/cluster existing points. Most of the following information on the origin of the dataset is presented from Yann Lecun's website.

Example MNIST Images - 5,0,4,1
Source: https://www.tensorflow.org/get_started/mnist/beginners

The original black and white (bilevel) images from the NIST (National Institute of Standards and Technology) database were size normalized to fit in a 20x20 pixel box while preserving their aspect ratio. The resulting images contain grey levels as a result of the anti-aliasing technique used by the normalization algorithm. the images were centered in a 28x28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28x28 field.

The 60,000 pattern training set contained examples from approximately 250 writers. It was made sure that the sets of writers of the training set and test set were disjoint.

Exploratory Analysis

Supervised learning techniques have been tested extensively on this dataset. Neural nets have been shown perform very well at this task. Some recent Convolutional Neural Network (CNN) architectures achieve near perfect (error rate = 0.23%) classification. It could be argued that near human accuracy has been achieved. Thus, not only is the dataset separable, a neural net architecture is able to sufficiently discriminate between the digit images, while implicitly learning hidden features.[1]

I have set out to perform the unsupervised classification of these digit images. K-Means is an obvious choice for featureless clustering. It will serve as the baseline for this task.

Since neural net architectures were effective in the supervised learning task, it makes sense to adopt a similar architecture for an improved classifier. The Autoencoder is one such structure. Theoretically the hidden layer of an autoencoder should encode minimal encodings required to distinguish between all major classes of the input fed to it. The challenge is in isolating these encodings and clustering them, which is the aim of this assignment.

II. Predictive Task

The aim of this assignment is to perform the unsupervised classification of the MNIST images. First using an existing technique, K-Means; then an autoencoder based model is proposed and evaluated.

Features

In in its raw form, each datum is a 28x28 grayscale image.The design of image feature vectors for a given dataset is not a trivial task. Standard feature vector sets like SIFT (Shift Invariant Feature Transform) are large and designed for general images. Such generated vectors are very large and developed without using the statistics of the data in feature generation. This has motivated me to choose a neural net architecture to learn features that are generated with the express intent to discriminate between the input classes.

At this point it may be argued that a convolutional layer would best learn the image features. But when using an autoencoder model, this introduces added complexity (in the reconstruction layers) and in general increasing the number of network variables in memory (RAM). This will hence be a task for future exploration.

Validity and Baseline

If any classifier performs (significantly) better than random chance (classification rate >> 1/number of classes = 10% for MNIST) on the 10,000 test images, then I can conclude that it is due to some pattern learned from the training data. If the classification rate is close to random chance, the classifier may not be learning any real pattern either due to it's lack of representational freedom or incorrect usage of the classifier (possibly during training).

As mentioned earlier, feature generation is non-trivial. This points to the featureless clustering technique that is K-Means clustering, which will serve as the baseline. For this task, we will assume nothing but the number of classes is known to us, making it a truly unsupervised problem.
K-Means will be compared to the autoencoder based model. The sparse encodings of the image classes in the 'code layer' of the autoencoder are essentially feature vectors generated by the network for each class. They will be interesting to visualize and interpret.

K-Means

March 12, 2017

- Scikit-learn's KMeans routine is used for training

```
In [ ]: import numpy as np
        from scipy.stats import mode
        from sklearn.cluster import KMeans
        from tensorflow.examples.tutorials.mnist import input_data
```

- There are ten digits (0 - 9) each belonging to their own class. Number of classes = 10

```
In [ ]: mnist = input_data.read_data_sets("MNIST/")
        n_classes = 3
```

- The training images (train_x) are vectorized from 28x28 grayscale images to 784x1 vectors
- The KMeans++ algorithm is chosen to intialize the clustering algorithm for best performance.
- The number of clusters (n_clusters) is set equal to the number of classes. This is because the task assumes completely unsupervised learning knowing just the number of classes.
- Increasing n_clusters may improve performance. But we would not know which group of clusters belong to the each digit class. Eg. If we had 26 clusters arbitrarily named a,b,c...z, we would not know which set to assign to digit '1'. It could be any subset of {a,b,c...z} and there is no obvious way to predict it.

```
In [ ]: #train_x = mnist.train.images
        train_x = np.concatenate((mnist.train.images[mnist.train.labels == 0],mnist.train.images
        kmeans = KMeans(n_clusters=n_classes,init='k-means++').fit(train_x)
```

- KMeans produces its own class labels. In order to test peformance, we need to design a 1-1 mapping (idx_transform) between KMeans' auto-geneated labels and the true digit class labels of the points.
- The aforementioned is done by assigning to each cluster the class label of the majority of its points i.e. the label of the 'mode' of the points.
- It may now be argued that since we are developing a mapping from the KMeans labels to the true class labels, we could have chosen more clusters to improve performance. Again, it is *only for the testing phase* that we develop this mapping. In a real world application the algorithm should output the class clusters without any knowledge of the true labels.

```
In [ ]: idx_transform  = np.array([0]*n_classes)
        for digit in range (n_classes):
            idx_transform[digit] = np.round(mode(kmeans.predict(mnist.train.images[mnist.train.l
```

```
In [ ]: #labels = mnist.train.labels
        #images = mnist.train.images

        labels = np.concatenate((mnist.train.labels[mnist.train.labels == 0],mnist.train.labels[
        images = np.concatenate((mnist.train.images[mnist.train.labels == 0],mnist.train.images[

        t = np.array([idx_transform[x] for x in labels])
        y = kmeans.predict(images)
```

```
In [ ]: err_rate = 0.0
        for i in range(len(t)):
            err_rate += t[i] != y[i]
        err_rate /= len(t)
        print err_rate
```

- Configurations of KMeans that were tested:
  - Intilialization:
    - random: random vectors from the input set are chosen as centers
    - kmeans++: chooses new cluster centers intelligently based on the euclidean distance of the input vectors from the current cluster centers.
  - No of Iterations: were varied from the default minimum, 300 to 1000. The variation produced little to no change in performance (over a few random initialization trials) , confirming the suspicion that the algorithm reaches its termination condition.
  - No of classes:
    - 2 classes: Using just '0' and '1' images the algorithm achieves an astonishing ~0.9% error rate. Variation is within 0.02% over random iterations.
    - 3 classes: Using '0','1','2' images the algorithm achives ~9% error rate. Variation is within 0.04% over random iterations.
    - All 10 classes: Achieves ~12% error rate, as before. Variation is within 0.04% over random iterations.
    - variation images

```
In [ ]:
```

MNIST_Autoencoder-4Class

March 12, 2017

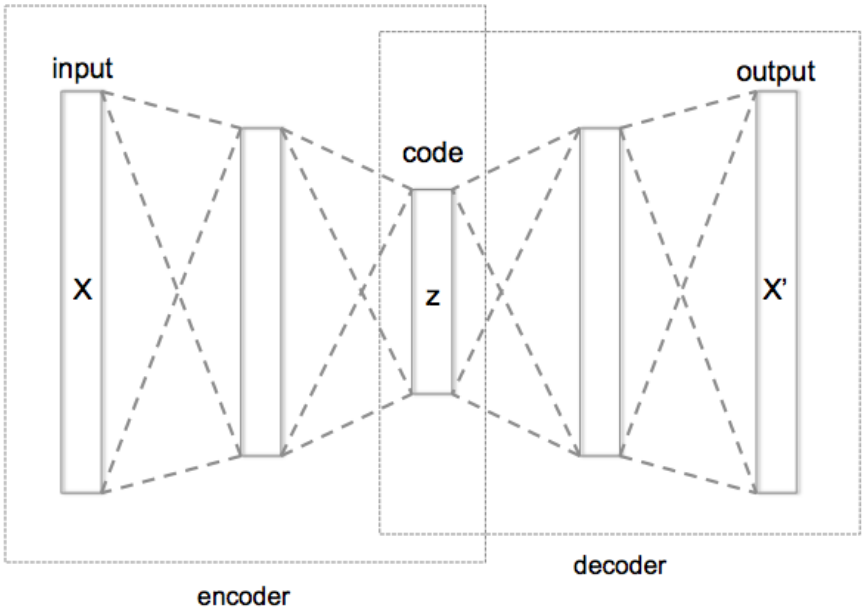
1 Libraries and Data Import

In [ ]:
```
# Libraries

    #%matplotlib notebook
    import pylab
    import numpy as np
    import tensorflow as tf
    from scipy.stats import mode
    from sklearn.cluster import KMeans
    from matplotlib import pyplot as plt
    from mpl_toolkits.mplot3d import Axes3D
    from IPython.core.display import clear_output
    from tensorflow.examples.tutorials.mnist import input_data
```

In [ ]:
```
tf.reset_default_graph
sess = tf.InteractiveSession()
```

In [ ]:
```
mnist = input_data.read_data_sets("MNIST_data/")
train_x = np.concatenate((mnist.train.images[mnist.train.labels == 0],mnist.train.images
#train_x = mnist.train.images

n_train = train_x.shape[0]
n0 = np.sum(mnist.train.labels == 0); n1 = np.sum(mnist.train.labels == 1); n2 = np.sum(
e1 = n0+n1;
input_size = train_x.shape[1]
x = tf.placeholder(tf.float32, [None, input_size])
momentum = tf.placeholder(tf.float32,[])
```

2 Network Architecture




```
Source: Wikimedia Commons
- Input: 784x1 image vector
- Encoder: One hidden layer - 28 neurons
- Code: 3 neurons (~no of classes)
- Decoder: 4 layers of 784 neurons i.e. [784,784,784,784]
- Error Function: ||X-X'|| (2 Norm)
```

- The network model is in essence, an autoencoder with assymteric hidden layers.
- 'hidden_shape' is vector conatining the number of hidden neurons in each layer. Eg. Hidden layer 1 has 28 neurons.
  - It is assymetric because the task is assymteric. We want to reduce the input to a low dimensional vector conataining only the inter-class variation. Only a few neurons are required to summarize class information. Hence there is only one small input layer containing 28 neurons. Any small number could be chosen. I choose 28 simply because it is a small number equal to the size of the image on each side. *Experimenting with larger input layers didn't improve reconstruction or classification.*
  - But this representation should be indicative of each class i.e. it should be possible to reconstruct the input from this vector. This requires a complicated network structure which is why the last 4 layers are large fully connected layers. 784 is simply a large number , chosen simply because it is the vectorized size of each image. *Experimenting with smaller layers after code_lyr increases reconstruction error significantly.*
- The code layer 'code_lyr' should be proportional in size to the number of classes. Here we are classifying 3 digits '0','1','2', which is why we choose 3 neurons in the code layer.

  - It represents the essential low dimensional representation of the input data.
  - The code layer is main source of appromation/reconstruction error in the network. Every input is first converted into the low dimensional code layer representation. It is then reconstructed in subsequent layers.
  - If the model could store very little information about an input in the code layer, it makes sense that the code layer firstly contains information about each class. This is because inter-class variation (Eg. difference between '0' and '1') is much more than intra-class variation (such as whether the digit '2' has a loop at it left-bottom corner or not).
  - In other words, storing inter-class information, reduces the reconstruction error more than when storing intra-class information. If there are additional degrees of freedom, the code layer could store some information about the intra-class variation too.
  - In summary, increasing the dimensionality of the code layer improves reconstruction error at the cost of storing intra-class variation, which is detrimental to classification.

- The activation functions 'act_fns' of each layer are chosen to be hyperbolic tangents. Changing the activation (to sigmoid, relu) had little effect on reconstruction error or classification.

In [ ]:
```
# Hyperparameters

hidden_shape = [28,3,784,784,784,784] #No of neurons in each hidden layer
code_lyr = 2 #index of hidden layer containing sparse vector
act_fns = ['tanh']*len(hidden_shape) #Activation functions of layers
num_lyr = len(hidden_shape) #No of hidden layers
net_shape = [input_size] + hidden_shape + [input_size]
learning_rate = tf.placeholder(tf.float32,[])
beta = tf.placeholder(tf.float32,[]) #sparsity_penalty weight
```

In [ ]:
```
# Parameters

params = {'W':{},'b':{}} #W: Weights, b: biases
for lay_num in range(num_lyr+1):
    params['W'][lay_num] = tf.Variable(tf.random_normal([net_shape[lay_num], net_shape[l
    params['b'][lay_num] = tf.Variable(tf.random_normal([net_shape[lay_num+1]],stddev =
```

- In additon to recontruction error 'cost', the code layer representation 'code' of each input 'x' is stored for future classification.
- The activation function of the code layer is a squashing function (tanh or sigmoid). This allows us to meaningfully visualize the separation in the code layer space.
- In an attempt to manually separate the classes, I built a sparsity penalty that penalized the code layer representations for being too close to the center of the code layer space (Eg. (0,0,0) for 3 layer with tanh activation/ (0.5,0.5,0.5) with 3 layer sigmoid).
- It became clear that it was much more important to find accurate represenations of each class than trying to forcefully separate their data points. The sparsity penalty would usually, unrealistically group different classes near the boundaries of the code layer space, thus reducing classification accuracy.
- However, the sparsity penalty is not completely useless. When we are confident that the input were reconstructed to sufficient accuracy, which is synomymous with good separation of code layer representations of the input, we could use the sparsity penalty.
- In fact the sparsity penalty should be designed in a custom fashion.

- Train the network without the sparsity penalty. Make sure reconstruction error is low i.e. the network is good at obtaining a low dimensional representation of the inputs.
- Visualise the inputs in the code layer and design a sparsity penalty that would push the data points to different corners of the code layer space.
- Now increase the weight to the sparsity penalty 'beta' slowly until the data gets separated.

- A simpler method of classification it to use the code layer representations instead of the input, in a clustering algorithm like KMeans. The removal of intra-class variation should in theory improve the classification rate.

In [ ]:
```
# Network Graph
def act_fn(a,lay_num):
    if act_fns[lay_num] == 'tanh':
        return tf.tanh(a)
    elif act_fns[lay_num] == 'relu':
        return tf.nn.relu(a)
    elif act_fns[lay_num] == 'elu':
        return tf.nn.elu(a)
    elif act_fns[lay_num] == 'sig':
        return tf.sigmoid(a)

def AutoEncoder(x,params):
    z = x
    for lay_num in range(num_lyr):
        z = act_fn(tf.add(tf.matmul(z,params['W'][lay_num]),params['b'][lay_num]),lay_nu
        if lay_num == code_lyr-1:
            wall_pusher = -tf.reduce_mean(tf.sqrt(tf.add(tf.square(z[:,0]),tf.square(z[:
            sparsity_penalty = wall_pusher
            code = z
    x_ = tf.add(tf.matmul(z,params['W'][lay_num+1]),params['b'][lay_num+1])
    cost = tf.sqrt(tf.reduce_mean(tf.squared_difference(x,x_))) + tf.multiply(beta,spars
    return (cost,code,x_)
```

In [ ]:
```
cost,code,x_ = AutoEncoder(x,params)
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

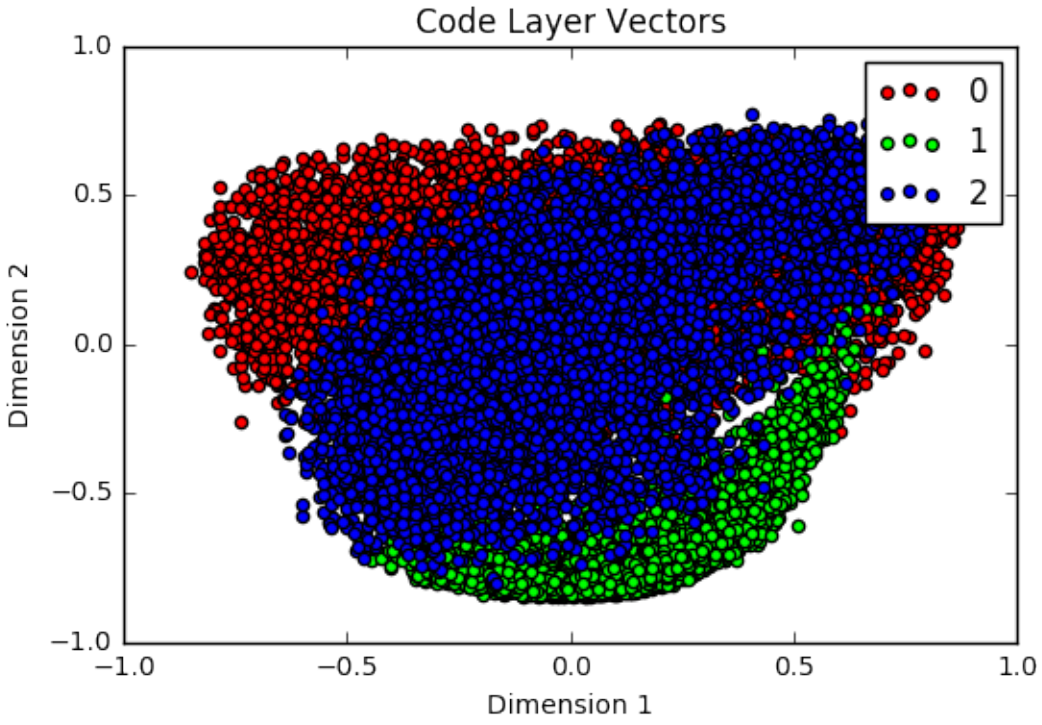
In [ ]:
```
# Parameter Initialization
sess.run(tf.initialize_all_variables())
n_iter = 0
summary_writer = tf.summary.FileWriter('/home/puneeth/logs', sess.graph)
```

In [ ]:
```
# Learning

while True:
    try:
        _,cost_,code_,x__ = sess.run([optimizer,cost,code,x_],feed_dict={x:train_x,learn
        #code_[a:b] = code__
        if n_iter % 10 == 0:
```

```
            print('iter'+str(n_iter)+' cost:'+str(cost_))
        if n_iter+1 % 20 == 0:
            clear_output()
        n_iter += 1
        #a  = (a+11000)%55000
        #b  = (b+11000)%55000
    except KeyboardInterrupt:
        print "Training is stopped"
        break
```

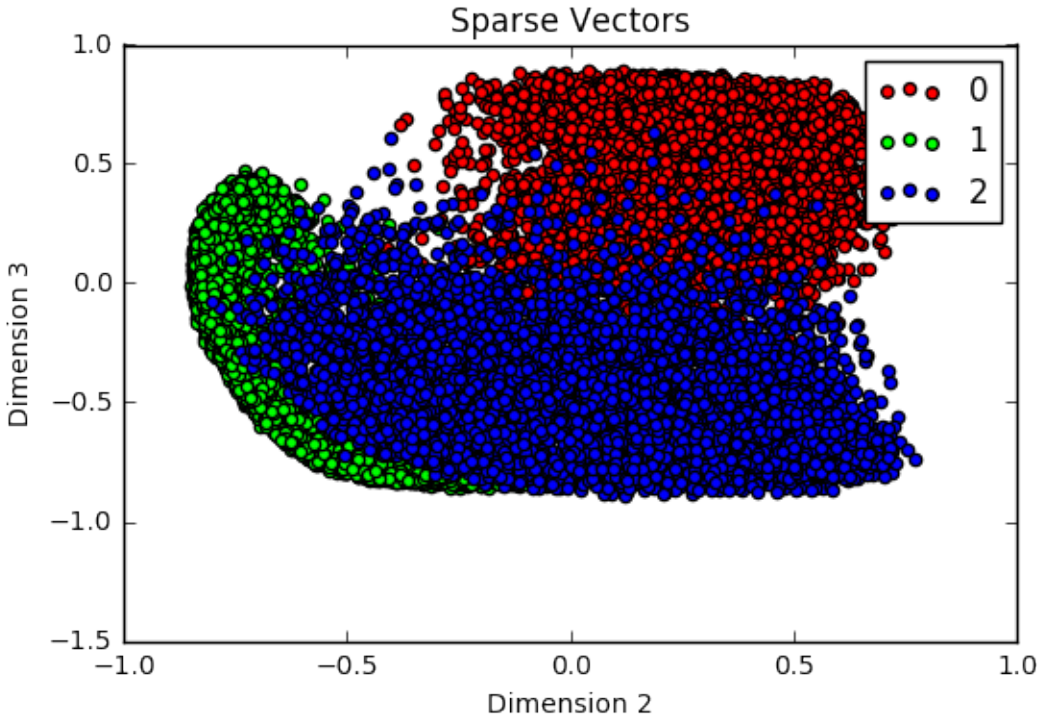
3 Code Layer Vector Plots

In [48]:
```
#plots
plt.title('Code Layer Vectors'); plt.xlabel('Dimension 1'); plt.ylabel('Dimension 2');
plt.scatter(code_[:n0][:,0],code_[:n0][:,1],c=[1,0,0], label = '0')
plt.scatter(code_[n0:e1][:,0],code_[n0:e1][:,1],c=[0,1,0], label = '1')
plt.scatter(code_[e1:][:,0],code_[e1:][:,1],c=[0,0,1], label = '2')
plt.legend()
plt.show()
```

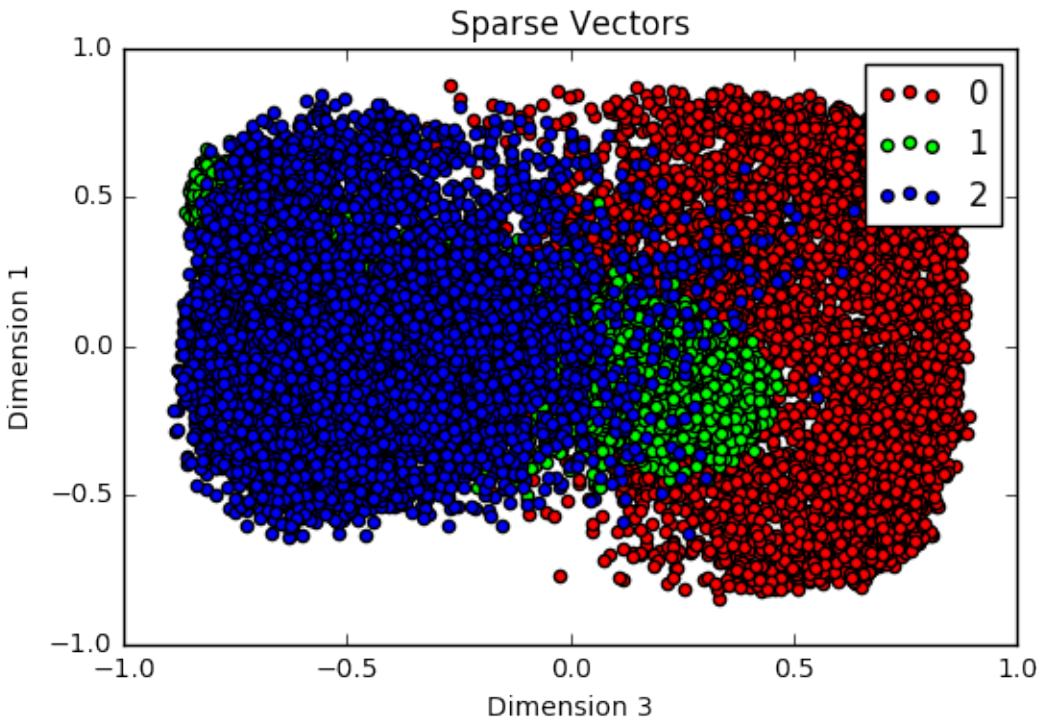


In [49]:
```
plt.title('Sparse Vectors'); plt.xlabel('Dimension 2'); plt.ylabel('Dimension 3');
plt.scatter(code_[:n0][:,1],code_[:n0][:,2],c=[1,0,0], label = '0')
plt.scatter(code_[n0:e1][:,1],code_[n0:e1][:,2],c=[0,1,0], label = '1')
```

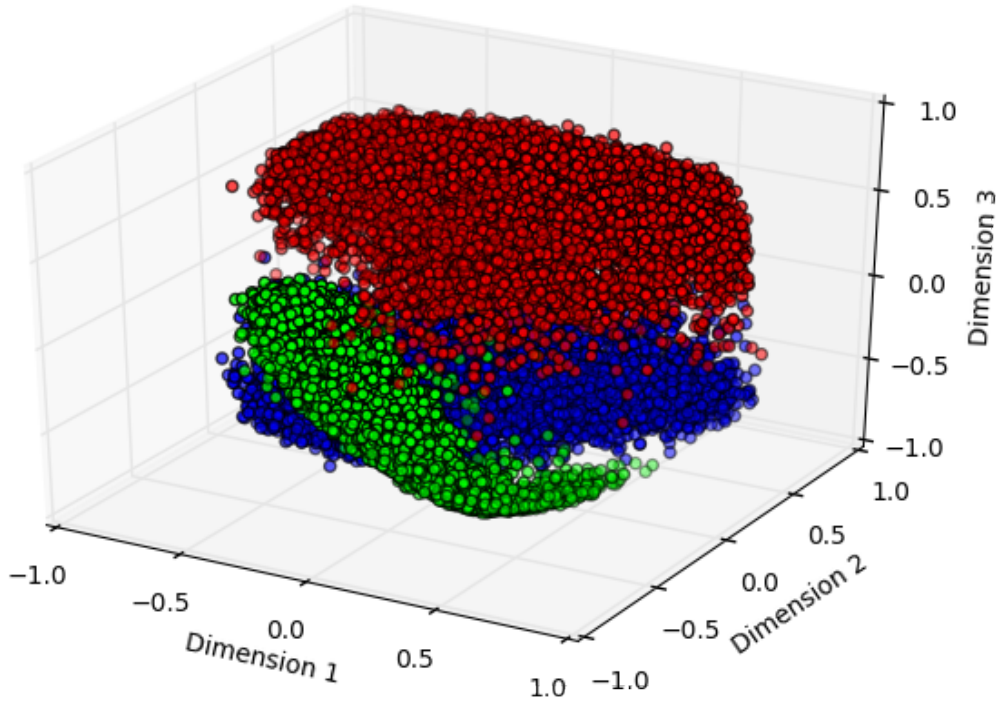
```
plt.scatter(code_[e1:][:,1],code_[e1:][:,2],c=[0,0,1], label = '2')
plt.legend()
plt.show()
```




```
In [50]: plt.title('Sparse Vectors'); plt.xlabel('Dimension 3'); plt.ylabel('Dimension 1');
         plt.scatter(code_[:n0][:,2],code_[:n0][:,0],c=[1,0,0], label = '0')
         plt.scatter(code_[n0:e1][:,2],code_[n0:e1][:,0],c=[0,1,0], label = '1')
         plt.scatter(code_[e1:][:,2],code_[e1:][:,0],c=[0,0,1], label = '2')
         plt.legend()
         plt.show()
```




```
In [51]: fig = pylab.figure()
         ax = Axes3D(fig)
         ax.scatter(code_[:n0][:,0],code_[:n0][:,1],code_[:n0][:,2],c=[1,0,0])
         ax.scatter(code_[n0:e1][:,0],code_[n0:e1][:,1],code_[n0:e1][:,2],c=[0,1,0])
         ax.scatter(code_[e1:][:,0],code_[e1:][:,1],code_[e1:][:,2],c=[0,0,1])
         ax.set_xlabel('Dimension 1')
         ax.set_ylabel('Dimension 2')
         ax.set_zlabel('Dimension 3')
         plt.show()
```



- k-means clustering on the codes (dimension-reduced version of inputs)
  - It runs much faster as the number of dimensions is small.
  - Learns slightly better than k-means on original data (3 classes). Error Rate ~ 7.5%

```
In [ ]: n_classes = 3
        kmeans = KMeans(n_clusters=n_classes,init='k-means++').fit(code_)

In [55]: idx_transform  = np.array([0]*n_classes)

         a1 = 0
         b1 = 0
         for i1 in range(n_classes):
             b1 += np.sum(mnist.train.labels == i1)
             idx_transform[i1] = np.round(mode(kmeans.predict(code_[a1:b1]))[0])
             a1 = b1

         labels = np.concatenate((mnist.train.labels[mnist.train.labels == 0],mnist.train.labels
         #labels = mnist.train.labels
         t1 = np.array([idx_transform[x1] for x1 in labels])
         y1 = kmeans.predict(code_)

         err_rate = 0.0
         for i in range(len(t1)):
```

```
            err_rate += t1[i] != y1[i]
        err_rate /= len(t1)
        print "Error Rate=" + str(err_rate)
```

```
Error Rate=0.0754109869537
```

In [ ]:

IV. Conclusions and Future Work

The classification error rate (over a few random runs of K-Means) is tabulated below.

Model/Data	2-Class [‘0’ & ‘1’]	3-Class [‘0’,‘1’,‘2’]	10-Class [‘0’-‘9’]
K-Means (Baseline)	12%	9%	0.9%
Autoencoder	?	7.5%	0.8%

- Unfortunately I was not able to train the network long enough for the autoencoder to converge for 10-classes. Using a convolutional layer should both increase accuracy (the input are images) and, reduce training time. [Future Work]
- It is clear that the autoencoder improves the accuracy by restricting the code layer to capture only inter-class variation. [Baseline and Validity conditions are met]
- The layers following the code layer must be dense to minimize reconstruction error. Those preceding it can be reduced in size to decrease complexity/training time.
- These codes are spatially separated in the code layer space.
- Increasing the dimensionality of the code space often reduces the error rate. Eg. taking n=4 dimensions in the code layer for the 3-Class problem (‘0’, ‘1’, ‘2’) reduces the autoencoder error rate to ~6.5%. But it is expected that beyond a certain point it should not make a difference as intra-class variation should come into play. In general more classes means a large code layer space.
- Random initialization of weights and biases is essential to fast training and avoiding local minima that stem from similar simultaneous updates across many weights.
- A sparsity function can always be developed manually, after looking at the output (visualization) of the autoencoders codes. This function can be used to make sure our code clusters converge to predictable locations. But their design cannot be automated. It usually leads to large misclassification error because, in the eyes of the sparsity function, points of all classes are penalized equally to move to the desired deterministic locations.
- K-Means can be used as an alternative to the sparsity function.

V. Literature Survey

As mentioned in the exploratory analysis, MNIST is very well studied dataset for supervised learning. Neural nets have been shown perform very well at this task. Some recent Convolutional Neural Network (CNN) architectures achieve near perfect (error rate = 0.23%) classification. It could be argued that near human accuracy has been achieved. Thus, not only is the dataset separable, a neural net architecture is able to sufficiently discriminate between the digit images, while implicitly learning hidden features. In addition,the results of running nearly every standard supervised learning technique on the MNIST dataset is compiled here[1].

K-Means has been previously tested on the MNIST dataset. A detailed explanation which matches my results is posted online at http://johnloeber.com/docs/kmeans.html [3]. They too achieve 12% error-rate on the dataset.

The use of autoencoders in dimensionality reduction and subsequent classification has been around for sometime. In most cases the training was done in a greedy layer wise fashion [4a]. Its use in unsupervised classification is usually limited to learning feature vectors [4b]. It was my intention to explore the use of their dimensionality reduction/feature extraction properties to design an unsupervised classifier.

References

[1] The MNIST dataset : http://yann.lecun.com/exdb/mnist/
[2] Tensorflow's tutorial on MNIST classification:
https://www.tensorflow.org/get_started/mnist/beginners
[3] An online blog that verifies the result of KMeans on MNIST in a straightforward fashion:
http://johnloeber.com/docs/kmeans.html
[4] Stanford UFLDL tutorials on Autoencoders:

a) http://ufldl.stanford.edu/wiki/index.php/Stacked_Autoencoders
b) http://ufldl.stanford.edu/tutorial/unsupervised/Autoencoders/