Homework 1 for Deep Learning and Data Mining

Xiaoqian Chen

June 3, 2019

Abstract

In this report, we use a classic USPS handwritten digits data set to start our journey to the Deep learning. An auto-encoder is a neural network that tries to reconstruct its input. A multilayer perception (MLP) with only 1 hidden layer is applied to auto encoding the digit image. In the experiment, there are 4 alternative dimension for the hidden layer, obtained by Principle Components Analysis. After the test of various learning options, we know that all zeros is not recommended and choose some same random numbers as weights initialization.In addition, small learning rate, about 5% of training set as batch size is practically reasonable for this USPS data set. The early stopping learning strategy give 491 epochs is good choice to avoid over fitting. For 16×16 digits image, the compression network with hidden layer size h99=185 is relatively optimal choice, which keeps most necessary information and reduces dimension of input layer.

1 Data Set

The data set is the USPS collection of handwritten digits. It consists of scans (images) of digits that people wrote. The input is a 16 by 16 image of grey scale pixels, showing an image of a handwritten digit. The output is simply which of the 10 different digits ($0 \sim 9$). There are 1100 examples for each class, totally $N = 11000$ cases in the data set.

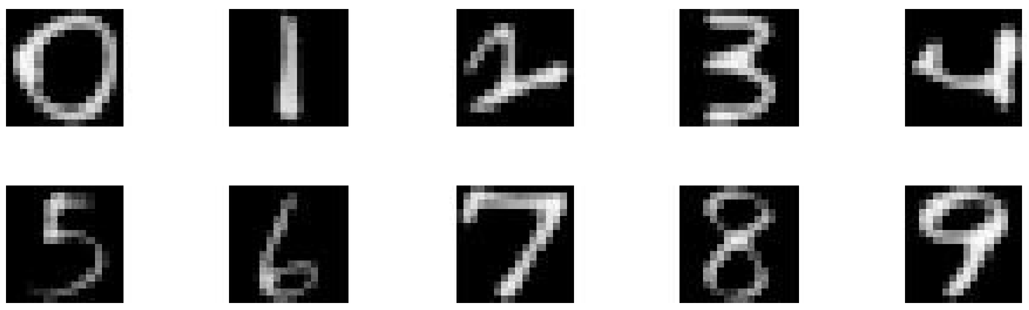


Figure 1: Handwritten Digits

The data set contains $8,000$ examples for training (about 73% of the total data set), and $3,000$ examples for testing (about 27% of the total data set). The digits have been size-normalized and centered in a fixed-size image (16×16 pixels) with values range from 0 to 255, where 0 is black and 255 is white. For simplicity, each image has been flattened and converted to a 1×256 vector.
For output layer, there are $r = 10$ distinct classes ($0 \sim 9$), presented by 10 independent standard basis vector $\{e_1, \ldots, e_{10}\}$, such that $e_1 = \{1, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$. e_1 stand for digit 0, e_2 stand for digit 1, and so on.

Data set description		
	Training set	Test set
Set Size	8000	3000
Value Range of Input Descriptors	{0, ..., 255}	{0, ..., 255}
Size of each Input Descriptors	256	256
Value Range of Class	{0,1}	{0,1}
Size of each Class	10	10

Table 1: Data set description

2 Multilayer Perception Architecture

An Auto-Encoder is a type of artificial neural network used to learn efficient data coding in an unsupervised manner. The aim of an Auto-Encoder is to learn a representation (encoding) for a set of data, typically for dimensionality reduction, by training the network to ignore signal noise. Along with the reduction side, a reconstructing side is learn, where the Auto-Encoder tries to generate from the reduced encoding a representation as close as possible to its original input, hence its name.

A perception is a linear classifier; that is, it is an algorithm that classifies input by separating two categories with a straight line. Input is typically a feature vector x multiplied by weights w and added to a bias b: $y = w \times x + b$. A multilayer perception (MLP) is a deep, artificial neural network. It is composed of more than one perception. They are composed of an input layer to receive the signal, an output layer that makes a decision or prediction about the input, and in between those two, an arbitrary number of hidden layers that are the true computational engine of the MLP.

There is the simplest form of an auto-encoder, which is a feed-forward, non-recurrent neural network very similar to the many single layer perceptions. Here, Multilayer Perception Architecture for an auto-encoder is a 3 layers fully-connected network, having an input layer, an output layer and one hidden layers connecting them. Neurons between two adjacent layers are fully pairwise connected, but neurons within a single layer share no connections. There is only one hidden layer with dimension h. The dimension of the output layer is equal to the dimension of input layer. The response function is applied for all neurons in the hidden layer and output layer, such as sigmoid function, RELU function.

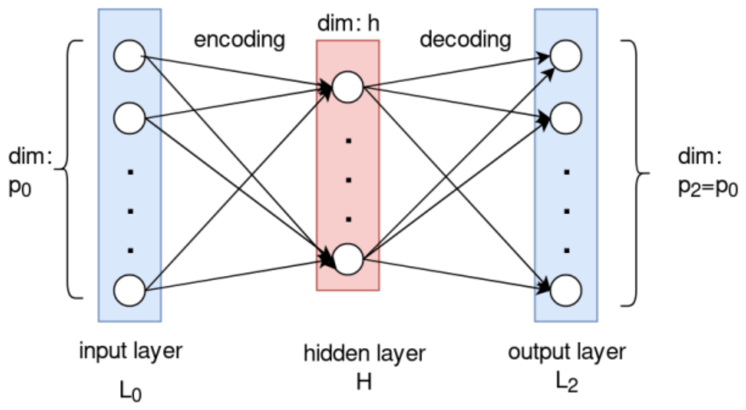


Figure 2: MLP architecture of an Auto-Encoder

For our USPS handwritten digits recognition, each input case is a flattered vector ($p_0 = 256$), connecting with the hidden layer (size h) by linear function with weights $W_1(256 \times h)$ and bias $b_1(1 \times h)$. As response function, RELU function set for all neurons. The hidden layer will compute a linear function which is then passed into a RELU activation function. The output layer and hidden layer connected by linear function with weights $W_2(h \times 256)$ and bias $b_2(1 \times 256)$. The output layer has the same number of nodes as the input layer ($p_2 = p_0 = 256$), and with the purpose of reconstructing its own inputs (instead of predicting the target value Y).

Traditionally, Auto-Encoder were used for dimensionality reduction or feature learning. The goal of the auto encoding is to obtain output vectors get very close to the input vectors. So that the output layer is in same size as input layer. More specifically, the input will be our target, since we want to keep information as much as possible. If hidden layer size $h < p_0$, we call it compression construction. Otherwise, $h > p_0$, where the percentage of active "neurons" in H for each input is rather small, we name it as sparsity construction.

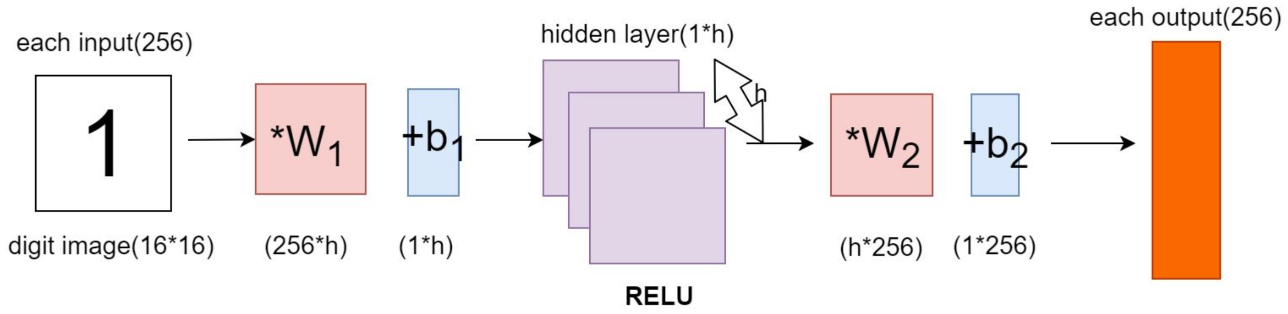


Figure 3: Auto encoding MLP with 1 hidden layer in size h

3 Principal Component Analysis

Principal component analysis (PCA) is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables (entities each of which takes on various numerical values) into a set of values of linearly uncorrelated variables called principal components.

PCA is a powerful technique used for data exploration and compression in data analysis and machine learning. Effectively, it reduces the dimension of observational data by eliminating redundancy. By reducing the dimension of observational data, we can construct more demonstrative visualizations for low dimensional data sets or reduce storage and processing requirements in high dimensional data sets. In some cases, dimension reduction may even improve the accuracy of predictive models.

Given a matrix of N-case training data A $(N \times p)$, reduced system $A_{reduced}(N \times q)$ with q $(q \leq p)$ principal components with the following PCA algorithm.

1. Mean normalize $A = A - mean(A)$.
2. Compute the covariance matrix of the mean normalized data, $\Sigma = \frac{1}{N-1}AA^T$.
3. Find the eigenvectors and eigenvalues of Σ.
4. Sort the eigenvectors in descending eigenvalue order as $\text{s}_1, s_2, \ldots, s_p, where s_1 \geq s_2 \geq \ldots s_p$.
5. Let $A_{reduced}$ be new matrix with the first q column vectors. $A_{reduced}$ is the new basis for our system.

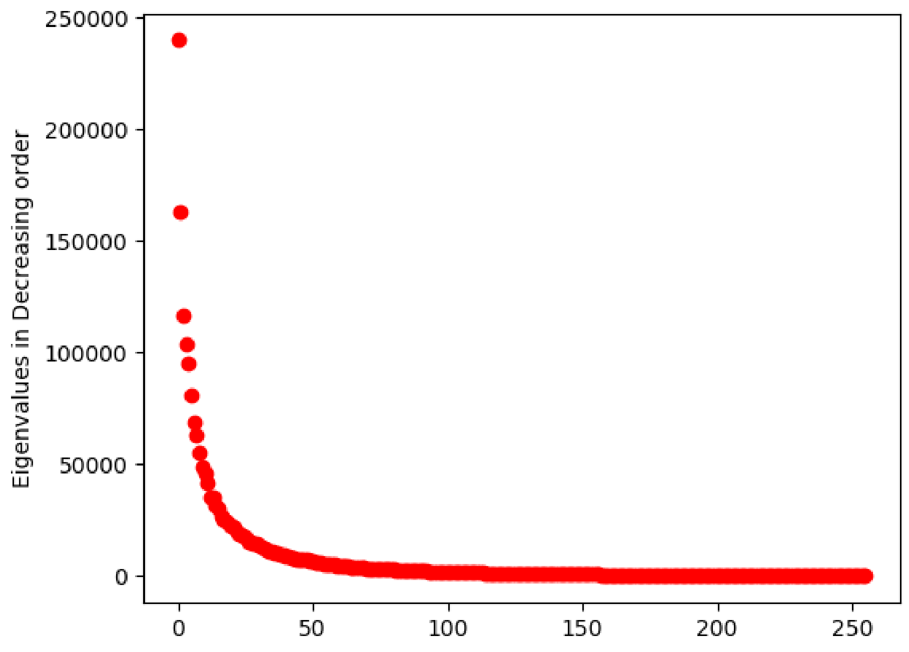


Figure 4: Eigenvalues in decreasing order

In USPS data set, size of all input data is $N = 11000$. Define h95, ($h95 \geq p$), as the smallest number of eigenvalues, that preserves 95% of the total sum of eigenvalues. In another word, the number of components explain 95% variance of data. Similarly define h99, ($h95 \geq p$), to preserve 99% of the total sum of eigenvalues. $95\%{=}\sum_{i=1}^{h95} s_i \frac{}{\sum_{i=1}^{p} s_i}$ $99\%{=}\sum_{i=1}^{h99} s_i \frac{}{\sum_{i=1}^{p} s_i}$ For the all USPS data set, we calculate the 256 eigenvalues and sort them in decreasing order, which presented in the Figure 2. From the 100-th in decreasing order, eigenvalue approaches 0. The reset about 156 eigenvalues preserve rare information of the original data set. For USPS training data set, $h55 = 101, h99 = 185$. There are strong correlations between some input features, at lease 71 units in the USPS input data. It make sense for handwritten digits , since each digit only ”lights” some part of the images. For example, the edge of the image keeps black for all images, which is helpless for recognizing class.

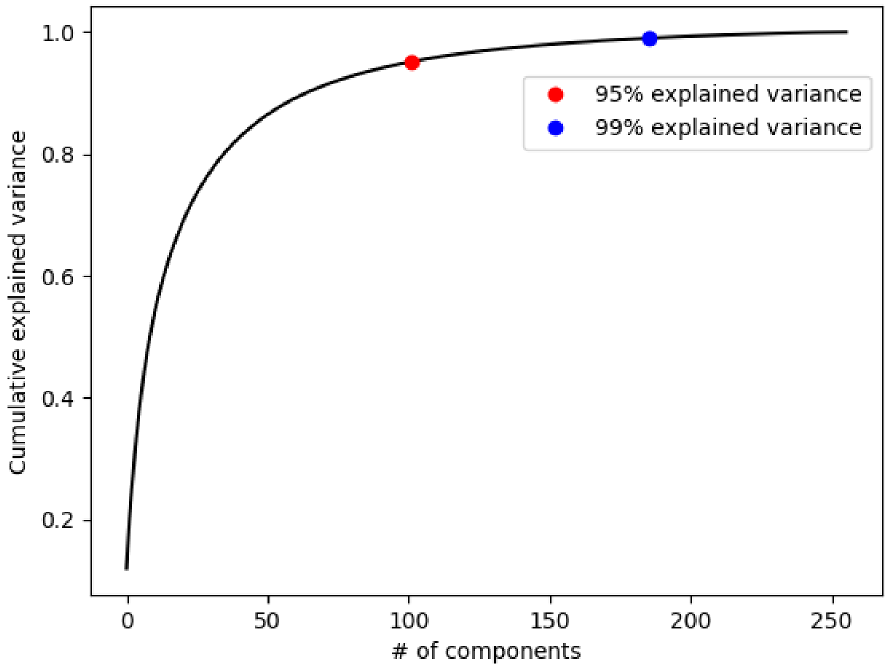


Figure 5: Cumulative Explained Variance

To estimate one larger plausible value H99, apply PCA analysis to each class data set, then gain 10 different number of components, which perserves 99% of sum of eigenvalues in the corresponding class.
H99$=\sum_{i=1}^{N_{class}} h99$ *in each class* For the all USPS data set, the size of class $N_class = 10$, H99 is the sum of these 10 number of components. For USPS training data set, $H99 = 1420$, $(1420 = 114 + 161 + 154 + 155 + 153 + 137 + 119 + 156 + 135 + 136)$.

In addition, define another ”very large” alternative value HH. HH $= 3\ \times H99$ Now, we have 4 alternative size of hidden layer.

h95	h99	H99	HH
101	185	1420	4260

Table 2: Alternative Size of Hidden Layer

4 Auto encoding

In this project, we train our neural network by Python3.5 with Tensorflow, which is an open-source machine learning library for research and production. Now, let construct our 3-layer Network as auto-encoder.

There are some main parameters and learning rule about this auto-encoder.

response function	loss function	optimal options
RELU	mean square error	batch train learning rate early stopping

Table 3: auto-encoder settings

The mean square error (MSE) was used as loss function and RELU was used as response function. The main goal is to minimize the result of sum of MSE for training set to get a adapted network. MSE=$\sum_{i=1}^{N}(Y_i - Y_i)^2$
The Rectified Linear Unit is very commonly used activation function in deep learning models. The function returns 0 if it receives any negative input, but for any positive value x it returns that value back. So it can be written as RELU(x)= max{0,x}
We need terminologies like epochs, batch size, iterations only when the data is too big which happens all the time in machine learning and we cant pass all the data to the computer at once. So, to overcome this problem we need to divide the data into smaller sizes and give it to our computer one by one and update the weights of the neural networks at the end of every step to fit it to the data given. There are some optimal options for MLP, including batch learning, learning rate and early stopping strategy.

Batch learning:
In batch training the adjustment delta values are accumulated over all training items, to give an aggregate set of deltas, and then the aggregated deltas are applied to each weight and bias. We can divide the training set of 8000 examples into batches of 400 then it will take 20 iterations to complete 1 epoch.

Learning rate:
When using a gradient descent algorithm, we typically use a smaller learning rate for batch

mode training than incremental training, because all of the individual gradients are summed together before determining the step change to the weights.

Early stopping:
One Epoch is when an entire data set is passed forward and backward through the neural network only once. We need to pass the full data set multiple times to the same neural network. As the number of epochs increases, more number of times the weight are changed in the neural network and the curve goes from under-fitting to optimal to over-fitting curve. Early stopping is a method that allows you to specify an arbitrary large number of training epochs and stop training once the model performance stops improving on a hold out validation data set.

Intermediary outputs to monitor learning quality:
For monitoring learning quality, define Root Mean Square Error(RMSE), the gradient of Mean Squared Error (G) and another interpretable gradient(IG). After each epoch, record these three indicator. RMSE $= \sqrt{MSE} = \sqrt{\sum_{i=1}^{N}(Y_i - Y_i)^2}$ G = ——W(N+1)-W(N)—— IG =1$\overline{\sigma \times M||W(N+1)-W(N)||}$ where σ is the learning rate and $M = 2p_0h + h + p_0$ is the size of weights.

- epoch: one forward pass and one backward pass of all the training examples.
- iteration: one forward pass and one backward pass of one batch of images the training examples.
- weights initialization: initialization of the all weights and bias
- batch size: the number of training examples in one forward/backward pass. The higher the batch size, the more memory space you'll need.
- learning rate: gradient descent step size
- early stopping: If the early shopping rule is applied,we mark minimal validation loss. Once validation loss go upward, we stop training.

Firstly, we choose $epochs = 200$, batch size 400, which means we have 8000/400=20 batches in one epoch. And learning rate 0.001. Also apply early stopping training.

Listing 1: Hyper-parameters

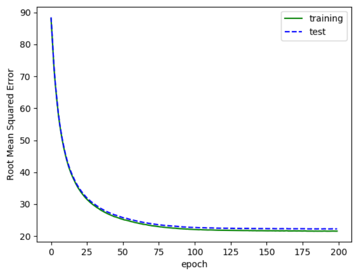
```
h = 185  # Number of units in the first hidden layer,
         # eg 101,185,1420,4260
epochs = 200  # Total number of training epochs
batch_size = 400  # Training batch size
learning_rate = 0.001  # gradient descent step size
do_early_stopping = Ture #if apply early stopping
display_fre =10 # frequency of compute and record indicators
```

Listing 2: Creating the network graph

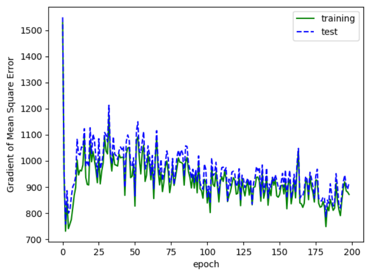
```
# Create the graph for the linear model
# Placeholders for inputs (x) and outputs(y)
x=tf.placeholder(tf.float32,shape=[None,img_size_flat],name='X')
y=tf.placeholder(tf.float32,shape=[None,img_size_flat],name='Y')

# Create the network layers
fc1 = fc_layer(x, h, 'FC1', use_relu=True)
output = fc_layer(fc1, img_size_flat, 'OUT', use_relu=Ture)

# Define the loss function, optimizer, and gradient
loss = tf.losses.mean_squared_error(labels=output,
          predictions=x)
optimizer=tf.train.AdamOptimizer(learning_rate=learning_rate,
               name='Adam-op').minimize(loss)
grad_and_var=tf.train.AdamOptimizer(learning_rate=learning_rate,
                  name='Adam-op').compute_gradients(loss)
```

Secondly, we build our network with the Tensorflow library. The fa_layer is a linear function for forward propagation (function code does not show in the report). x is our input data and y is the target data, which is same as input data in the auto-encoder. Set mean square error as our loss function and optimize it. After creating the graph, it is time to train our model. here we use defaulted weight initialization function in the tensorflow.

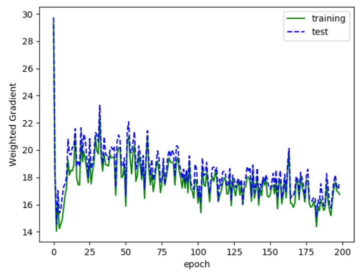
After each epochs, we will compute and record these learning quality indicators (RMSE, G and IG) , so that we could observe how the training goes on. If the mean square error of test data set decrease, we record the best so far loss function and model. In the end, if the loss of test data set go upward ,even the training set gives the decreasing loss, we choose the network with minimal loss for test data set. This part code attached in the end of the report.



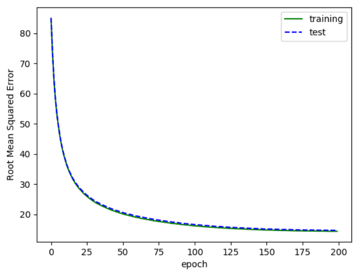
(a) h95=101



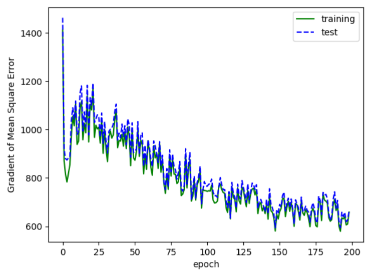
(b) h95=101



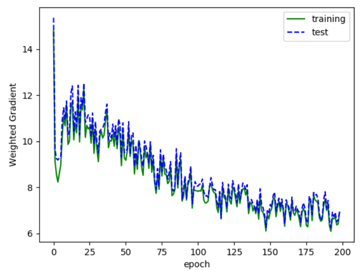
(c) h95=101



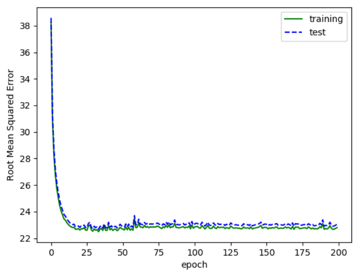
(d) h99=185



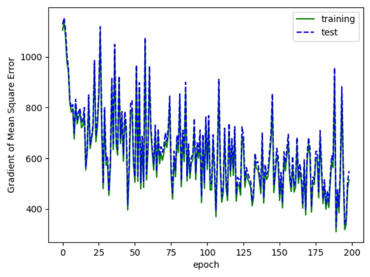
(e) h99=185



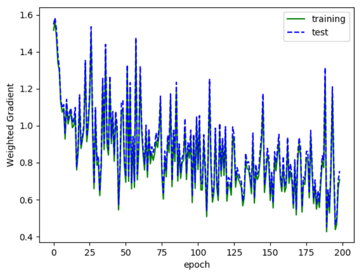
(f) h99=185



(g) H99=1420



(h) H99=1420



(i) H99=1420

Figure 6: Auto-encoder learning indicators

With the above parameters and network set, we have the auto-encoder with 4 alternative hidden layer size. The choice of HH=4260 have to much parameters to estimate, USPS training set with size of 8000 is not enough. So that we cloud not train the encoder for HH. More detailed discussion will be show in the Implement of various learning part. You could see there is not relatively outstanding perform in mean square error with the large hidden layer size, so that we will choose h=h95 in the implement of various learning options.

5 Impact of various learning options

- Weight Initialization:

    1. ones: Initializer that generates tensors initialized to 1.
    2. random_normal: Initializer that generates tensors with a normal distribution.
    3. random_uniform: Initializer that generates tensors with a uniform distribution.
    4. variance_scaling: Initializer capable of adapting its scale to the shape of weights tensors.

In the Weight Initialization, we do not know what the final value of every weight should be in the trained network, but with proper data normalization it is reasonable to assume that approximately half of the weights will be positive and half of them will be negative. A reasonable-sounding idea then might be to set all the initial weights to zero, which we expect to be the best guess in expectation. This turns out to be a mistake, because if every neuron in the network computes the same output, then they will also all compute the same gradients during back propagation and undergo the exact same parameter updates. In other words, there is no source of asymmetry between neurons if their weights are initialized to be the same.

Therefore, we still want the weights to be very close to zero, but as we have argued above, not identically zero. As a solution, it is common to initialize the weights of the neurons to small numbers and refer to doing so as symmetry breaking. The idea is that the neurons are all random and unique in the beginning, so they will compute distinct updates and integrate themselves as diverse parts of the full network.

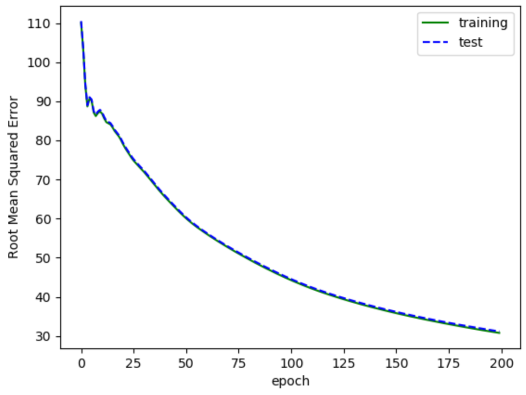
For example, random samples from a zero mean, unit standard deviation Gaussian. With this formulation, every neurons weight vector is initialized as a random vector sampled from a multi-dimensional Gaussian, so the neurons point in random direction in the input space. It is also possible to use small numbers drawn from a uniform distribution.

In our project, we try the 4 kind of initialization of weights with tf.initializers. Different weight initialization have relatively little impact on the final performance in practice.

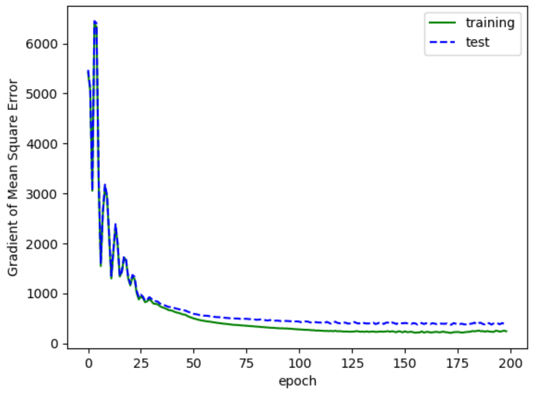
- Batch Size:
    1. Large batch size (50% of training set=4000)
    2. Small batch size (5)
    3. reasonable batch size (we choose 80 in our case)

Larger batch size means more "accurate" gradients, in the sense of them being closer to the "true" gradient one would get from whole-batch (processing the entire training set at every step) training. This leads to lower variance in the gradients. But it would be slow due to the number of iteration need per example. On the other side, a small batch size is often more efficient computationally. There would be huge noise in the update since the gradient update direction is only reliant on small part of data point. It currently seems that with deep networks it's preferable to take many small steps than to take fewer larger ones. And it bring the stochastic factor into training, which may lead to lower loss. Some online experience is about (50$\tilde{1}$000).

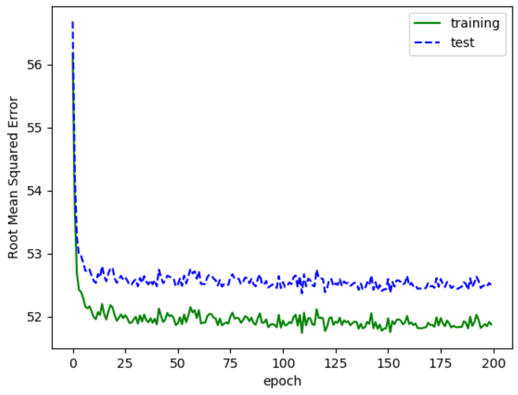
For exploring the effort of batch size, we choose two extreme small and large value to see the difference. Choosing a huge batch size of 4000 , which show in the Figure 7 (a). It's more accurate since the variance drops when you average something. and more stable gradients. Also, it is slow due to the number of iteration need per example. In Figure 7 (b) with a batch size of 5, there is a huge element of noise in the update. batch size of Small is quick but inaccurate. Thirdly, for USPS data set, we choose batch size 80 (Figure 7 (c)), then they would be 100 batches in one epoch. However, our loss function, Root Mean Squared Error, decreases in a faster speed and converge to a lower level.



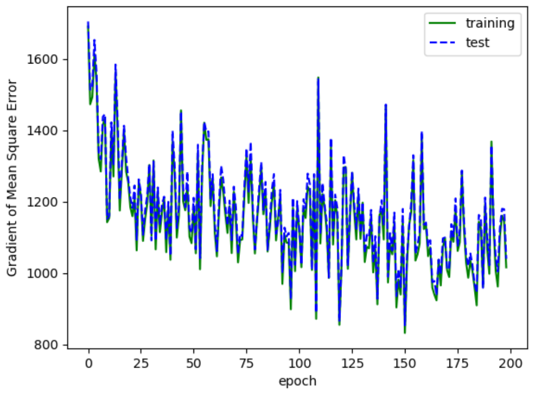
(a) RMSE, batch size =4000



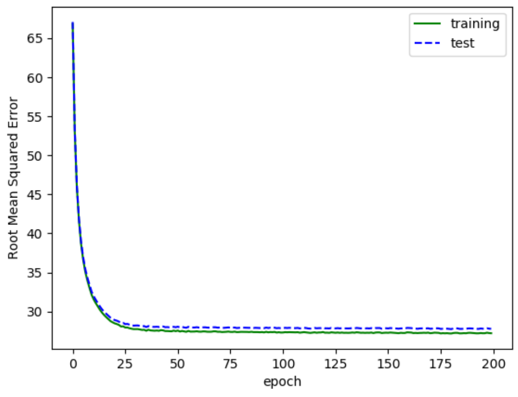
(b) Gradient of MSE, batch size =4000



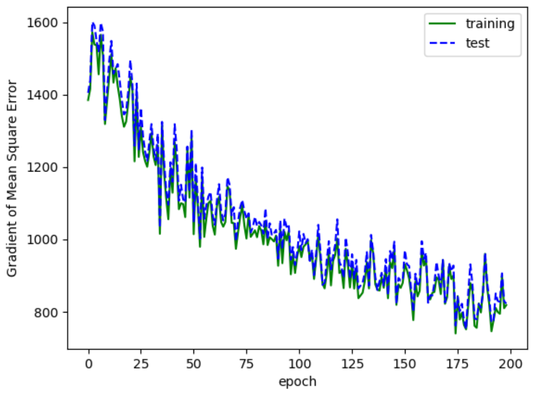
(c) RMSE, batch size =5



(d) Gradient of MSE, batch size =5



(e) RMSE, batch size =80

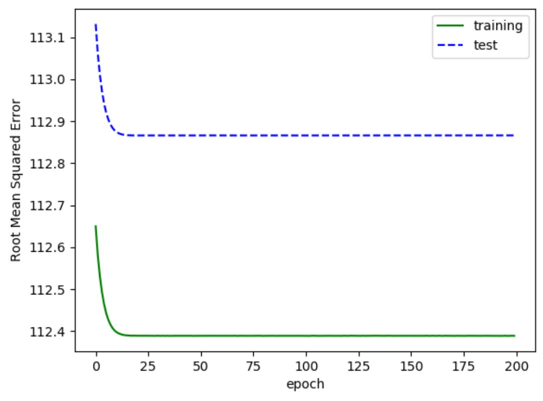


(f) Gradient of MSE, batch size =80

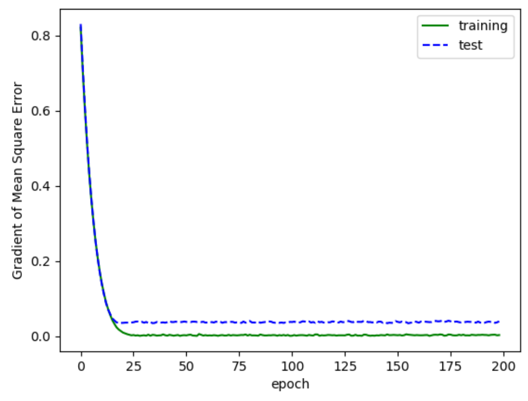
Figure 7: Batch Size

- Learning rate / gradient descent step size
    1. large Learning rate case(0.1)
    2. small Learning rate case(0.001)

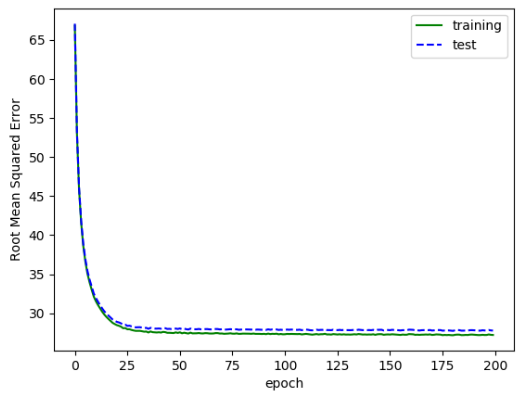
Learning rate is a hyper-parameter that controls how much we are adjusting the weights of our network with respect the loss gradient. The lower the value, the slower we travel along the downward slope. While this might be a good idea (using a low learning rate) in terms of making sure that we do not miss any local minimal, it could also mean that well be taking a long time to converge, especially if we get stuck on a plateau region. Conversely, if you specify a learning rate that is too large, the next point will perpetually bounce haphazardly across the bottom of the well like a quantum mechanics experiment gone horribly wrong. There are two examples in the Figure 8. In our project, since the data size is not too big, we prefer small learning rate, such that Learning rate = 0.001.



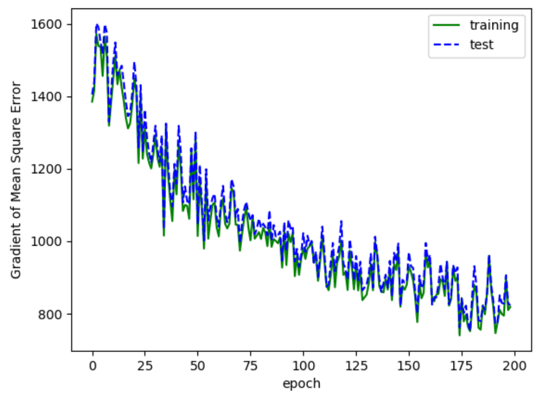
(a) RMSE, learning rate = 0.1



(b) Gradient of MSE, learning rate = 0.1



(c) RMSE, learning rate = 0.001



(d) Gradient of MSE, learning rate = 0.001

Figure 8: Learning Rate

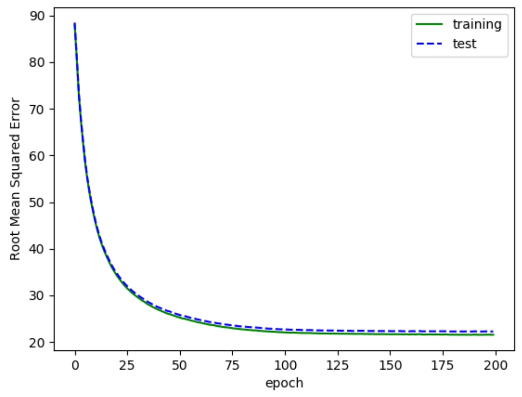
There's an intense amount of research going on examining this phenomenon in theoretical and empirical detail, and evidence currently seems to be pointing towards the higher variance gradients of batch gradient descent actually being tied to generalization. Higher batch sizes can allow for stable training with higher learning rates, so in a sense you might observe that the upper bound of acceptable learning rates is determined by your batch size. I think of this as, "the more accurate your gradients, the larger steps you can take without detriment."

- Dimension of the hidden layer
  1. h95=101: number of components explain 95% variance in the entire training set
  2. h99=185: number of components explain 99% variance in the entire training set
  3. H99=1420: sum of components size explain 99% variance in each class data set
  4. HH=4260: $HH = H99 \times 3$

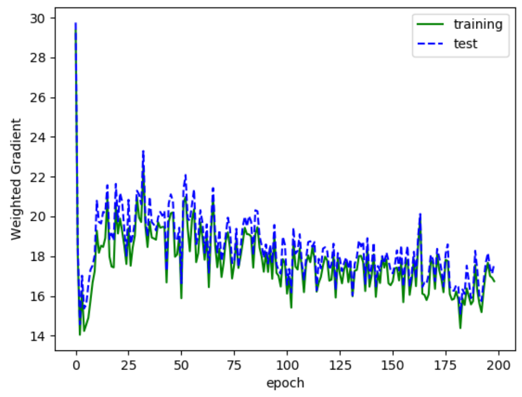
When dimension $h < p_0 = 256$, we have two compressing network example with h=h95=101 and h=h99=185. Even h95 is almost a half of h99, but it still shows good training result, relatively small and stable MSE (about 25). More hidden layer keep more information from the original image. But it is not perfect. The result become very sensitive, we could more promise a stable network with sparsity. The larger dimension of hidden layer, the more boiling gradients. we have Mean Residual Error approaching 0 , but also lost stable level. In practice, blindly purse the high dimension in the hidden layer is unwise, since it sacrifice the computational efficiency and stability.

In the case of $h = HH = 4260$, we totally have $M = 2p_0h + h + p_0 = 1,095,076$ unknown parameters need to estimate in the network. However, we only have $8000 * 256 = 2.048,000$ input to do estimate. It is not enough. Even we force the network to do learning, the result is inaccurate.

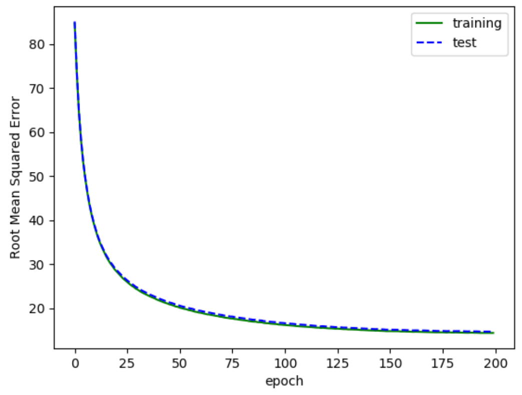
To better understand the difference, we reconstruct the output image. The left is original image, the right is the reconstructed image. You can tell that even with the high dimension in the hidden layer, image keep more information, however the difference is inconspicuous. Hence, we will choose h99 as the size of hidden layer in auto-encoder.



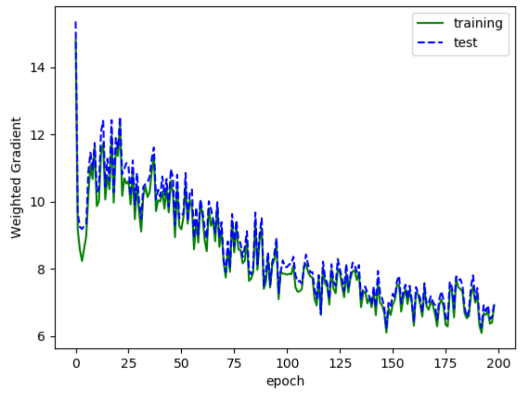
(a) RMSE, h = 101



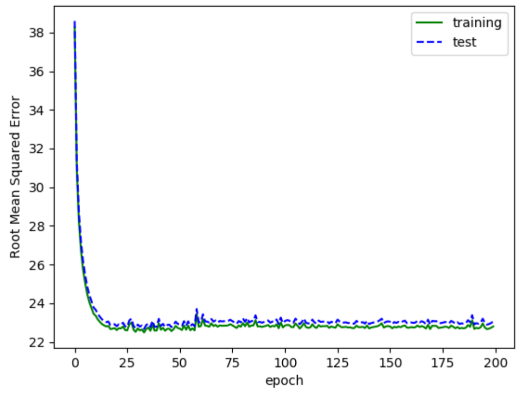
(b) Weighted Gradient of MSE, h = 101



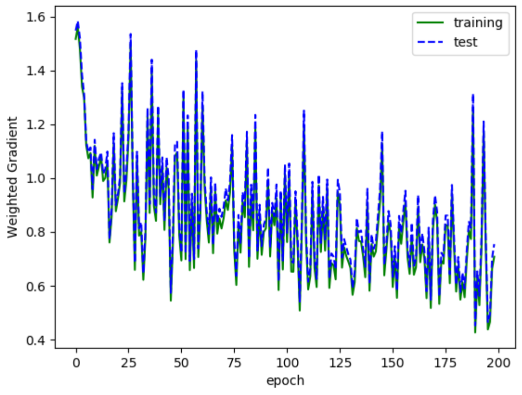
(c) RMSE, h = 185



(d) Weighted Gradient of MSE, h = 185

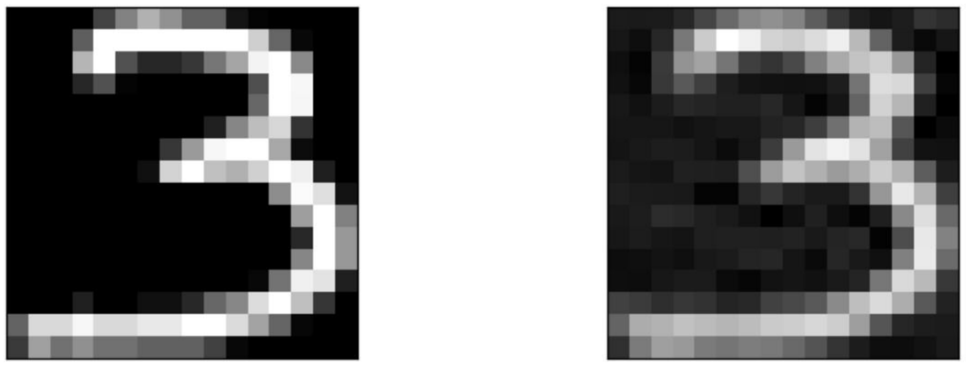


(e) RMSE, h = 1420

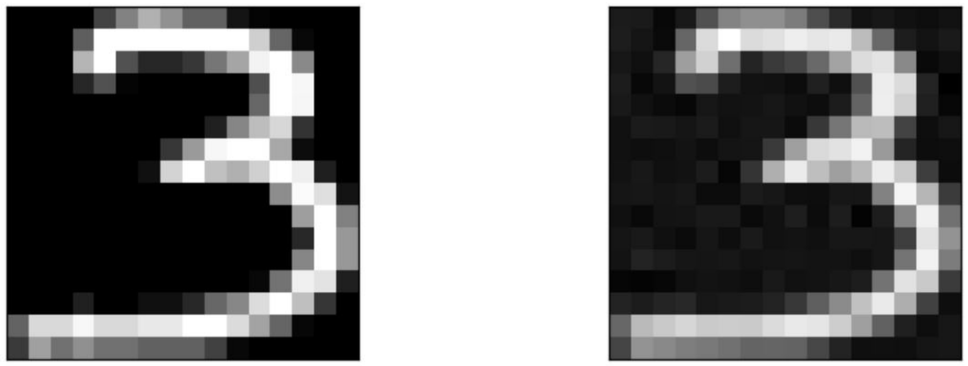


(f) Weighted Gradient of MSE, h = 1420

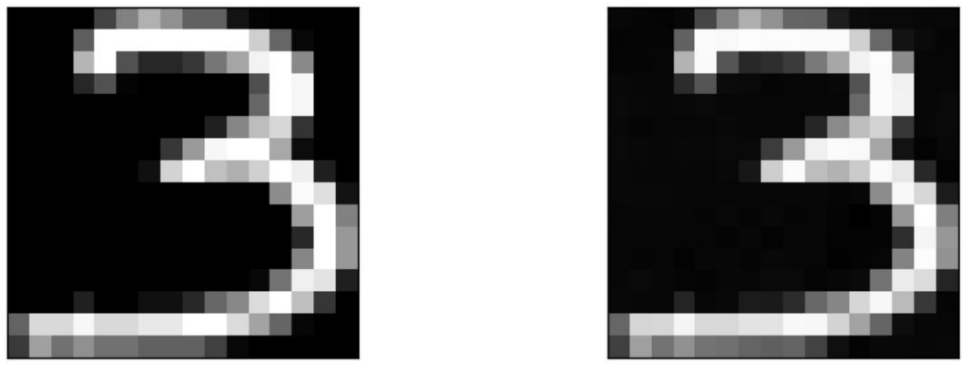
Figure 9: dimension of hidden layer



(a) h = 101



(b) h = 185



(c) h = 1420

Figure 10: reconstruction examples

- early stopping

To see if we could get more trained network, we set *epochs* = 500. Also the early stopping rule is applied to the learning to avoid the over fitting. We have lowest loss for test set after 491 epoch. Hence we will choose the network variable in the end of 491 epoch as our auto-encoder.

```
Early stopping:test loss was lowest after 491 epoch. We chose the model that we had then.
```

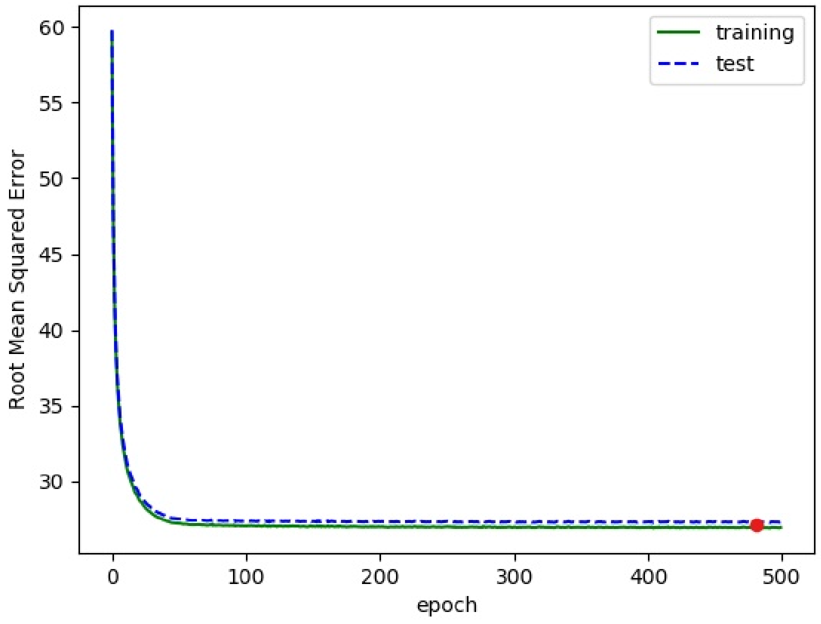


Figure 11: early stopping

- CONCLUSION

After the implement with various learning options, we get the auto-encoder with the following setting:

1. size of hidden layer=h99=185
2. weight initialization=defaulted initialization in thesorflow
3. batch size=80
4. learning rate=0.001
5. epoch = 491