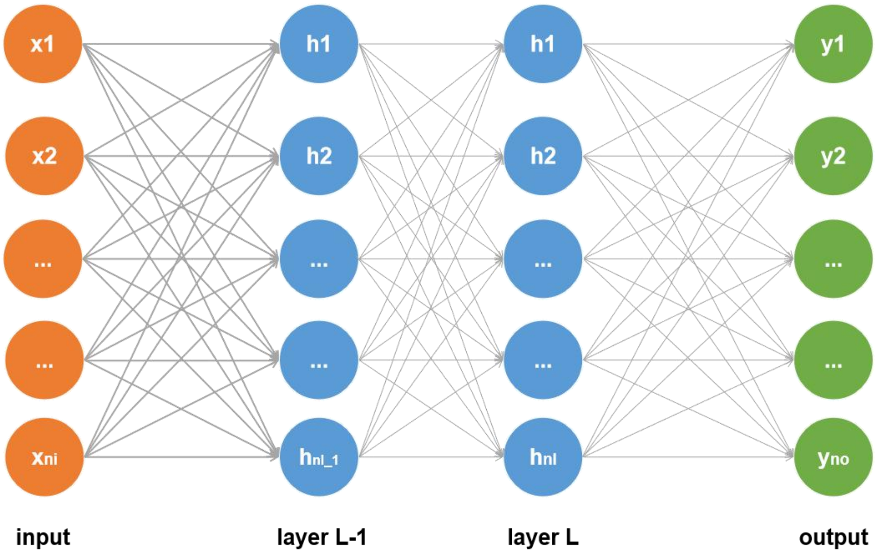
Neural Network

Neural Network is to try to solve the problems by approximation using weights networks. This approach is quite useful. It almost approximate every function by different layered-networks and activation function. All the deep neural network are based on the simple neural network. let's take a close look at neural network. We use a picture to show the neural network.



Forward Neural Network

input

$$\begin{bmatrix} x_1 & x_2 & \cdots & \cdots & x_d \end{bmatrix} \begin{bmatrix} w_{11} & \cdots & w_{1,ni} \\ \vdots & \ddots & \vdots \\ w_{d1} & \cdots & w_{d,ni} \end{bmatrix} + \begin{bmatrix} b_1 & \cdots & b_{ni} \end{bmatrix} = \begin{bmatrix} a_1^1 & \cdots & a_{ni}^1 \end{bmatrix}$$

$$\begin{bmatrix} h_1^1 & \cdots & h_{ni}^1 \end{bmatrix} = g\left(\begin{bmatrix} a_1^1 & \cdots & a_{ni}^1 \end{bmatrix}\right)$$

hidden

$$\begin{bmatrix} h_1^{L-1} & \cdots & h_{n_{L-1}}^{L-1} \end{bmatrix} \begin{bmatrix} w_{11} & \cdots & w_{1,n_L} \\ \vdots & \ddots & \vdots \\ w_{n_{L-1},1} & \cdots & w_{n_{L-1},n_L} \end{bmatrix} + \begin{bmatrix} b_1 & \cdots & b_{n_L} \end{bmatrix} = \begin{bmatrix} a_1^L & \cdots & a_{n_L}^L \end{bmatrix}$$

$$\begin{bmatrix} h_1^L & \cdots & h_{n_L}^L \end{bmatrix} = g\left(\begin{bmatrix} a_1^L & \cdots & a_{n_L}^L \end{bmatrix}\right)$$

output

$$\begin{bmatrix} h_1^L & \cdots & h_{n_L}^L \end{bmatrix} \begin{bmatrix} w_{11} & \cdots & w_{1,n_o} \\ \vdots & \ddots & \vdots \\ w_{n_L,1} & \cdots & w_{n_L,n_o} \end{bmatrix} + \begin{bmatrix} b_1 & \cdots & b_{no} \end{bmatrix} = \begin{bmatrix} a_1^o & \cdots & a_{n_o}^o \end{bmatrix}$$

$$\begin{bmatrix} y_1^o & \cdots & y_{n_o}^o \end{bmatrix} = g\left(\begin{bmatrix} a_1^o & \cdots & a_{n_o}^o \end{bmatrix}\right)$$

We can define our cost function based on the computational model accordingly. Here, we use cross-entropy as cost function.

$$L = y * \log\left(soft\max\left(y^{o}\right)\right)$$

Backward Propagation to update weights

$$output\ \ layer$$

$$\frac{\partial L}{\partial y^{o}} = \begin{pmatrix} \frac{\partial L}{\partial y_{1}^{0}} & \cdots & \frac{\partial L}{\partial y_{n_0}^{0}} \end{pmatrix}$$

$$\frac{\partial y^{o}}{\partial a^{o}} = g_{o}^{'}\left(a^{o}\right)$$

$$\frac{\partial a^{o}}{\partial w^{o}} = \left(\left(h^{L}\right)_{1}^{T} \quad \cdots \quad \left(h^{L}\right)_{n_o}^{T}\right), \frac{\partial a^{o}}{\partial b^{o}} = \left(1_{1} \quad \cdots \quad 1_{n_o}\right)$$

$$\frac{\partial a^{o}}{\partial h^{L}} = w^{o}$$

$$hidden\ \ layer$$

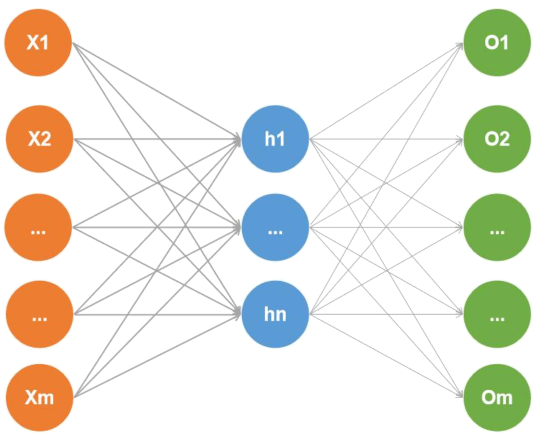
$$\frac{\partial h^{L}}{\partial a^{L}} = g_{L}^{'}\left(a^{L}\right)$$

$$\frac{\partial a^{L}}{\partial w^{L}} = \left(\left(h^{L-1}\right)_{1}^{T} \quad \cdots \quad \left(h^{L-1}\right)_{n_L}^{T}\right), \frac{\partial a^{L}}{\partial b^{L}} = \left(1_{1} \quad \cdots \quad 1_{n_L}\right)$$

$$\frac{\partial a^{L}}{\partial h^{L-1}} = w^{L}$$

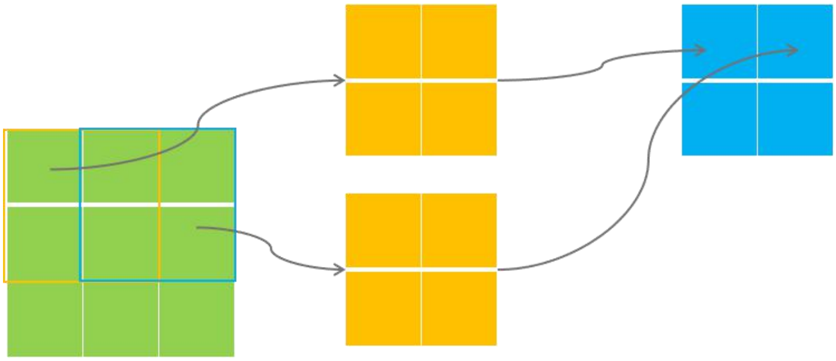
AutoEncoder

AutoEncoder is neural network that tries to reconstruct inputs. The number of AutoEncoder is quite small which means we use less nodes to capture as more features as possible.



Convolutional Neural Networks

It consists of Convolution and Pooling operations. These basic operations are the elements of deep convolutional neural network. The key idea is to design convolutional layers. In order to visualize the computational model, we use matrix multiplication. Here, we one-channel image as an example.



$$\begin{bmatrix} x_{11} & \cdots & x_{1d} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nd} \end{bmatrix}\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & \cdots & c_{1k} \\ \vdots & \ddots & \vdots \\ c_{m1} & \cdots & c_{mk} \end{bmatrix}$$

To better illustrate the operation, we use very simple matrix. For Convolutional operation,

$filter$

$$\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \rightarrow \begin{bmatrix} w_{11} \\ w_{12} \\ w_{21} \\ w_{22} \end{bmatrix}$$

$input$

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} \rightarrow \begin{bmatrix} x_{11} & x_{12} & x_{21} & x_{22} \\ x_{12} & x_{13} & x_{22} & x_{23} \\ x_{21} & x_{22} & x_{31} & x_{32} \\ x_{22} & x_{23} & x_{32} & x_{33} \end{bmatrix}$$

Then we can find the product of the two matrices.

$$\begin{bmatrix} x_{11} & x_{12} & x_{21} & x_{22} \\ x_{12} & x_{13} & x_{22} & x_{23} \\ x_{21} & x_{22} & x_{31} & x_{32} \\ x_{22} & x_{23} & x_{32} & x_{33} \end{bmatrix}\begin{bmatrix} w_{11} \\ w_{12} \\ w_{21} \\ w_{22} \end{bmatrix} = \begin{bmatrix} h_{11} \\ h_{12} \\ h_{21} \\ h_{22} \end{bmatrix}$$

$$\begin{bmatrix} h_{11} \\ h_{12} \\ h_{21} \\ h_{22} \end{bmatrix} \rightarrow \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$$

The gradient descent can be calculated.

$$\frac{\partial h}{\partial w} = \begin{bmatrix} x_{11} & x_{12} & x_{21} & x_{22} \\ x_{12} & x_{13} & x_{22} & x_{23} \\ x_{21} & x_{22} & x_{31} & x_{32} \\ x_{22} & x_{23} & x_{32} & x_{33} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Then we have

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$$

Through reshape the matrix, we can do the matrix operations easily. We can find the gradient and update weights easily.

For pooling operation,

$$\begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} \rightarrow \begin{bmatrix} c_{11} & c_{12} & c_{21} & c_{22} \\ c_{12} & c_{13} & c_{22} & c_{23} \\ c_{21} & c_{22} & c_{31} & c_{32} \\ c_{22} & c_{23} & c_{32} & c_{33} \end{bmatrix}$$

Find the maximum in each row

$$\begin{bmatrix} c_{11} & c_{12} & c_{21} & c_{22} \\ c_{12} & c_{13} & c_{22} & c_{23} \\ c_{21} & c_{22} & c_{31} & c_{32} \\ c_{22} & c_{23} & c_{32} & c_{33} \end{bmatrix} = \begin{bmatrix} p_{11} \\ p_{12} \\ p_{21} \\ p_{22} \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} \\ p_{21} & p_{22} \end{bmatrix}$$

Then, we get

$$pooling\left(\begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} \right) = \begin{bmatrix} p_{11} & p_{12} \\ p_{21} & p_{22} \end{bmatrix}$$

For a three-channel image, we can use the same method to do matrix multiplication.

filter

$$
\begin{cases}
\begin{bmatrix} w_{11}^1 & w_{12}^1 \\ w_{21}^1 & w_{22}^1 \end{bmatrix} \rightarrow \begin{bmatrix} w_{11}^1 \\ w_{12}^1 \\ w_{21}^1 \\ w_{22}^1 \end{bmatrix} \\
\begin{bmatrix} w_{11}^2 & w_{12}^2 \\ w_{21}^2 & w_{22}^2 \end{bmatrix} \rightarrow \begin{bmatrix} w_{11}^2 \\ w_{12}^2 \\ w_{21}^2 \\ w_{22}^2 \end{bmatrix} \\
\begin{bmatrix} w_{11}^3 & w_{12}^3 \\ w_{21}^3 & w_{22}^3 \end{bmatrix} \rightarrow \begin{bmatrix} w_{11}^3 \\ w_{12}^3 \\ w_{21}^3 \\ w_{22}^3 \end{bmatrix}
\end{cases}
\rightarrow
\begin{bmatrix} w_{11}^1 \\ w_{12}^1 \\ w_{21}^1 \\ w_{22}^1 \\ w_{11}^2 \\ w_{12}^2 \\ w_{21}^2 \\ w_{22}^2 \\ w_{11}^3 \\ w_{12}^3 \\ w_{21}^3 \\ w_{22}^3 \end{bmatrix}
$$

$$
\begin{cases}
\begin{bmatrix} x_{11}^1 & x_{12}^1 & x_{13}^1 \\ x_{21}^1 & x_{22}^1 & x_{23}^1 \\ x_{31}^1 & x_{32}^1 & x_{33}^1 \end{bmatrix} \rightarrow \begin{bmatrix} x_{11}^1 & x_{12}^1 & x_{21}^1 & x_{22}^1 \\ x_{12}^1 & x_{13}^1 & x_{22}^1 & x_{23}^1 \\ x_{21}^1 & x_{22}^1 & x_{31}^1 & x_{32}^1 \\ x_{22}^1 & x_{23}^1 & x_{32}^1 & x_{33}^1 \end{bmatrix} \\
\begin{bmatrix} x_{11}^2 & x_{12}^2 & x_{13}^2 \\ x_{21}^2 & x_{22}^2 & x_{23}^2 \\ x_{31}^2 & x_{32}^2 & x_{33}^2 \end{bmatrix} \rightarrow \begin{bmatrix} x_{11}^2 & x_{12}^2 & x_{21}^2 & x_{22}^2 \\ x_{12}^2 & x_{13}^2 & x_{22}^2 & x_{23}^2 \\ x_{21}^2 & x_{22}^2 & x_{31}^2 & x_{32}^2 \\ x_{22}^2 & x_{23}^2 & x_{32}^2 & x_{33}^2 \end{bmatrix} \\
\begin{bmatrix} x_{11}^3 & x_{12}^3 & x_{13}^3 \\ x_{21}^3 & x_{22}^3 & x_{23}^3 \\ x_{31}^3 & x_{32}^3 & x_{33}^3 \end{bmatrix} \rightarrow \begin{bmatrix} x_{11}^3 & x_{12}^3 & x_{21}^3 & x_{22}^3 \\ x_{12}^3 & x_{13}^3 & x_{22}^3 & x_{23}^3 \\ x_{21}^3 & x_{22}^3 & x_{31}^3 & x_{32}^3 \\ x_{22}^3 & x_{23}^3 & x_{32}^3 & x_{33}^3 \end{bmatrix}
\end{cases}
\rightarrow
\begin{bmatrix}
x_{11}^1 & x_{12}^1 & x_{21}^1 & x_{22}^1 & x_{11}^2 & x_{12}^2 & x_{21}^2 & x_{22}^2 & x_{11}^3 & x_{12}^3 & x_{21}^3 \\
x_{12}^1 & x_{13}^1 & x_{22}^1 & x_{23}^1 & x_{12}^2 & x_{13}^2 & x_{22}^2 & x_{23}^2 & x_{12}^3 & x_{13}^3 & x_{22}^3 \\
x_{21}^1 & x_{22}^1 & x_{31}^1 & x_{32}^1 & x_{21}^2 & x_{22}^2 & x_{31}^2 & x_{32}^2 & x_{21}^3 & x_{22}^3 & x_{31}^3 \\
x_{22}^1 & x_{23}^1 & x_{32}^1 & x_{33}^1 & x_{22}^2 & x_{23}^2 & x_{32}^2 & x_{33}^2 & x_{22}^3 & x_{23}^3 & x_{32}^3
\end{bmatrix}
$$

$$\begin{bmatrix} x_{11}^1 & x_{12}^1 & x_{21}^1 & x_{22}^1 & x_{11}^2 & x_{12}^2 & x_{21}^2 & x_{22}^2 & x_{11}^3 & x_{12}^3 & x_{21}^3 & x_{22}^3 \\ x_{12}^1 & x_{13}^1 & x_{22}^1 & x_{23}^1 & x_{12}^2 & x_{13}^2 & x_{22}^2 & x_{23}^2 & x_{12}^3 & x_{13}^3 & x_{22}^3 & x_{23}^3 \\ x_{21}^1 & x_{22}^1 & x_{31}^1 & x_{32}^1 & x_{21}^2 & x_{22}^2 & x_{31}^2 & x_{32}^2 & x_{21}^3 & x_{22}^3 & x_{31}^3 & x_{32}^3 \\ x_{22}^1 & x_{23}^1 & x_{32}^1 & x_{33}^1 & x_{22}^2 & x_{23}^2 & x_{32}^2 & x_{33}^2 & x_{22}^3 & x_{23}^3 & x_{32}^3 & x_{33}^3 \end{bmatrix} \begin{bmatrix} w_{11}^1 \\ w_{12}^1 \\ w_{21}^1 \\ w_{22}^1 \\ w_{11}^2 \\ w_{12}^2 \\ w_{21}^2 \\ w_{22}^2 \\ w_{11}^3 \\ w_{12}^3 \\ w_{21}^3 \\ w_{22}^3 \end{bmatrix} = \begin{bmatrix} h_{11} \\ h_{12} \\ h_{21} \\ h_{22} \end{bmatrix}$$

In the optimization process, the learning rate has an impact on the accuracy, which means it influences the convergence. In practice, the learning rate should be quite small to avoid fluctuation. The filters required need to be designed carefully to achieve the high accuracy results.

This project is to revisit neural network. We focus on the neural network design. The theories mentioned above are the guideline for designing neural networks. We build a computational model as follows. There are several .py files.

<Lib>

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
import numpy as np
import matplotlib.pyplot as plt
import os


default_mode = 'SNN'


""" Parameters of Simple Neural Network """
snn_param = {
    'learning_rate': 0.01,
    'num_steps': 1000,
    'batch_size': 128,
    'display_step': 100,
    'num_inputs': 784,
    'num_classes': 10,
    'hidden1': 256,
```

```
    'hidden2': 64
}

weights_snn = {
    'h1': tf.Variable(tf.random_normal([snn_param['num_inputs'],
snn_param['hidden1']])),
    'h2': tf.Variable(tf.random_normal([snn_param['hidden1'],
snn_param['hidden2']])),
    'out': tf.Variable(tf.random_normal([snn_param['hidden2'],
snn_param['num_classes']]))
}

biases_snn = {
    'b1': tf.Variable(tf.random_normal([snn_param['hidden1']])),
    'b2': tf.Variable(tf.random_normal([snn_param['hidden2']])),
    'out': tf.Variable(tf.random_normal([snn_param['num_classes']]))
}


""" Parameters of Convolutional Neural Network """
cnn_param = {
    'learning_rate': 0.01,
    'num_steps': 1000,
    'batch_size': 128,
    'display_step': 100,
    'num_inputs': 784,
    'channel': 1,
    'num_classes': 10,
    'filter1': 5,
    'hidden1': 32,
    'filter2': 5,
    'hidden2': 64,
    'fc1': 3136,
    'fc2': 1024,
    'dropout': 0.75
}

weights_cnn = {
    'wc1': tf.Variable(tf.random_normal([cnn_param['filter1'],
cnn_param['filter1'],
                                  cnn_param['channel'],
cnn_param['hidden1']])),
    'wc2': tf.Variable(tf.random_normal([cnn_param['filter2'],
cnn_param['filter2'],
```

```
                                   cnn_param['hidden1'],
cnn_param['hidden2']])),
    'wd1': tf.Variable(tf.random_normal([cnn_param['fc1'], cnn_param['fc2']])),
    'out': tf.Variable(tf.random_normal([cnn_param['fc2'],
cnn_param['num_classes']]))
}

biases_cnn = {
    'bc1': tf.Variable(tf.random_normal([cnn_param['hidden1']])),
    'bc2': tf.Variable(tf.random_normal([cnn_param['hidden2']])),
    'bd1': tf.Variable(tf.random_normal([cnn_param['fc2']])),
    'out': tf.Variable(tf.random_normal([cnn_param['num_classes']]))
}

""" Parameters of AutoEncoder """
ae_param = {
    'learning_rate': 0.04,
    'num_steps': 3000,
    'batch_size': 256,
    'display_step': 100,
    'num_inputs': 784,
    'examples_to_show': 10,
    'hidden1': 256,
    'hidden2': 128,
}

weights_ae = {
    'encoder_h1': tf.Variable(tf.random_normal([ae_param['num_inputs'],
ae_param['hidden1']])),
    'encoder_h2': tf.Variable(tf.random_normal([ae_param['hidden1'],
ae_param['hidden2']])),
    'decoder_h1': tf.Variable(tf.random_normal([ae_param['hidden2'],
ae_param['hidden1']])),
    'decoder_h2': tf.Variable(tf.random_normal([ae_param['hidden1'],
ae_param['num_inputs']])),
}

biases_ae = {
    'encoder_b1': tf.Variable(tf.random_normal([ae_param['hidden1']])),
    'encoder_b2': tf.Variable(tf.random_normal([ae_param['hidden2']])),
    'decoder_b1': tf.Variable(tf.random_normal([ae_param['hidden1']])),
    'decoder_b2': tf.Variable(tf.random_normal([ae_param['num_inputs']])),
}
```

<main>

```
from Load_Data import load_data
from Training import train


def run(mnist, mode):
    train(mnist, mode)


def app():
    print('Loading Data ......')
    mnist = load_data()
    print('Training Neural Network ......')
    print('Mode Selection')
    print("""There are a few neural network we can choose.
          Mode 1: Simple Neural Network  -> SNN
          Mode 2: Convolutional Neural Network -> CNN
          Mode 3: AutoEncoder -> AE""")
    # input_mode = input('Enter the mode:(ex -> SNN)\n')

    run(mnist, 'AE')


if __name__ == '__main__':
    app()
```

<Load_Data>

```
from Lib import *


def load_data():

    filename = "DataSet/"
    mnist = input_data.read_data_sets(filename, one_hot=True)

    return mnist
```

<Training>

```
from Lib import *
from SimpleNeuralNetwork import snn_model_fn
from ConvolutionalNeuralNetwork import cnn_model_fn
from AutoEncoder import ae_model_fn
from Storage import get_path


def train(mnist, mode):
```

```python
    model_path = get_path(mode)

    if mode == 'SNN':
        train_snn(mnist, model_path)
    elif mode == 'CNN':
        train_cnn(mnist, model_path)
    elif mode == 'AE':
        train_ae(mnist, model_path)


def train_snn(mnist, model_path):

    x = tf.placeholder("float", [None, snn_param['num_inputs']])
    y = tf.placeholder("float", [None, snn_param['num_classes']])

    train_op, loss_op, accuracy = snn_model_fn(x, y)

    init = tf.global_variables_initializer()
    saver = tf.train.Saver()

    with tf.Session() as sess:
        sess.run(init)

        saver.restore(sess, model_path)
        print("Model restored from file: %s" % model_path)

        for step in range(1, snn_param['num_steps'] + 1):
            batch_x, batch_y = mnist.train.next_batch(snn_param['batch_size'])
            sess.run(train_op, feed_dict={x: batch_x, y: batch_y})

            if step % snn_param['display_step'] == 0 or step == 1:
                loss, acc = sess.run([loss_op, accuracy], feed_dict={x: batch_x,
y: batch_y})
                print("Step " + str(step) + ", Minibatch Loss= " +
                      "{:.4f}".format(loss) + ", Training Accuracy= " +
                      "{:.3f}".format(acc))

        print("Training Finished!")

        print("Testing Accuracy:",
              sess.run(accuracy, feed_dict={x: mnist.test.images,
                                            y: mnist.test.labels}))

        # save_path = saver.save(sess, model_path)
```

```
        # print("Model saved in file: %s" % save_path)


def train_cnn(mnist, model_path):

    x = tf.placeholder("float", [None, cnn_param['num_inputs']])
    y = tf.placeholder("float", [None, cnn_param['num_classes']])
    keep_prob = tf.placeholder(tf.float32)

    train_op, loss_op, accuracy = cnn_model_fn(x, y, keep_prob)

    init = tf.global_variables_initializer()
    saver = tf.train.Saver()

    with tf.Session() as sess:
        sess.run(init)

        # saver.restore(sess, model_path)
        # print("Model restored from file: %s" % model_path)

        for step in range(1, cnn_param['num_steps'] + 1):

            batch_x, batch_y = mnist.train.next_batch(cnn_param['batch_size'])
            sess.run(train_op, feed_dict={x: batch_x, y: batch_y, keep_prob: 0.8})

            if step % cnn_param['display_step'] == 0 or step == 1:
                loss, acc = sess.run([loss_op, accuracy], feed_dict={x: batch_x,
y: batch_y,
                                                                     keep_prob: 1.0})

                print("Step " + str(step) + ", Minibatch Loss= " +
                      "{:.4f}".format(loss) + ", Training Accuracy= " +
                      "{:.3f}".format(acc))

        print("Training Finished!")
        print("Testing Accuracy:", sess.run(accuracy, feed_dict={x:
mnist.test.images[:256],
                                                                 y:
mnist.test.labels[:256],
                                                                 keep_prob: 1.0}))

        save_path = saver.save(sess, model_path)
        print("Model saved in file: %s" % save_path)
```

```
def train_ae(mnist, model_path):

    x = tf.placeholder("float", [None, ae_param['num_inputs']])

    train_op, loss_op,  decoder_op = ae_model_fn(x)

    init = tf.global_variables_initializer()
    saver = tf.train.Saver()

    with tf.Session() as sess:
        sess.run(init)

        # saver.restore(sess, model_path)
        # print("Model restored from file: %s" % model_path)

        for step in range(1, ae_param['num_steps'] + 1):
            batch_x, _ = mnist.train.next_batch(ae_param['batch_size'])
            sess.run(train_op, feed_dict={x: batch_x})

            if step % ae_param['display_step'] == 0 or step == 1:
                loss = sess.run(loss_op, feed_dict={x: batch_x})
                print("Step " + str(step) + ", Minibatch Loss= " +
                      "{:.4f}".format(loss))

        print("Training Finished!")

        save_path = saver.save(sess, model_path)
        print("Model saved in file: %s" % save_path)

        n = 4
        canvas_orig = np.empty((28 * n, 28 * n))
        canvas_recon = np.empty((28 * n, 28 * n))

        for i in range(n):

            batch_x, _ = mnist.test.next_batch(n)
            g = sess.run(decoder_op, feed_dict={x: batch_x})

            for j in range(n):
                canvas_orig[i * 28:(i + 1) * 28, j * 28:(j + 1) * 28] =
batch_x[j].reshape([28, 28])

            for j in range(n):
```

```
            canvas_recon[i * 28:(i + 1) * 28, j * 28:(j + 1) * 28] =
g[j].reshape([28, 28])

      print("Original Images")
      plt.figure(figsize=(n, n))
      plt.imshow(canvas_orig, origin="upper", cmap="gray")
      # plt.show()

      print("Reconstructed Images")
      plt.figure(figsize=(n, n))
      plt.imshow(canvas_recon, origin="upper", cmap="gray")
      plt.show()
```

<Evaluation>

```
from Lib import *


def snn_evaluation_fn(prediction, y):

   correct_pred = tf.equal(tf.argmax(prediction, 1), tf.argmax(y, 1))
   accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

   return accuracy


def cnn_evaluation_fn(prediction, y):

   correct_pred = tf.equal(tf.argmax(prediction, 1), tf.argmax(y, 1))
   accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

   return accuracy
```

<Storage>

```
def get_path(mode):

   return 'Model/' + str(mode) + '/' + str(mode) + '.ckpt'
```

<SimpleNeuralNetwork>

```
from Lib import *
from Evaluation import snn_evaluation_fn


def simple_neural_net_naf(x):

   layer_1 = tf.add(tf.matmul(x, weights_snn['h1']), biases_snn['b1'])
   layer_2 = tf.add(tf.matmul(layer_1, weights_snn['h2']), biases_snn['b2'])
   out_layer = tf.matmul(layer_2, weights_snn['out']) + biases_snn['out']
```

```
    return out_layer


def simple_neural_net_af(x):

    layer_1 = tf.nn.relu(tf.add(tf.matmul(x, weights_snn['h1']), biases_snn['b1']))
    layer_2 = tf.add(tf.matmul(layer_1, weights_snn['h2']), biases_snn['b2'])
    out_layer = tf.matmul(layer_2, weights_snn['out']) + biases_snn['out']

    return out_layer


def snn_cost_fn(logits, y):

    loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=y))
    optimizer = tf.train.AdamOptimizer(learning_rate=snn_param['learning_rate'])
    train_op = optimizer.minimize(loss_op)

    return train_op, loss_op


def snn_model_fn(x, y):

    logits = simple_neural_net_af(x)
    prediction = tf.nn.softmax(logits)
    train_op, loss_op = snn_cost_fn(logits, y)
    accuracy = snn_evaluation_fn(prediction, y)

    return train_op, loss_op, accuracy
```

<ConvolutionalNeuralNetwork>

```
from Lib import *
from Evaluation import cnn_evaluation_fn


def conv2d(x, w, b, strides=1):

    x = tf.nn.conv2d(x, w, strides=[1, strides, strides, 1], padding='SAME')
    x = tf.nn.bias_add(x, b)

```

```
    return tf.nn.relu(x)


def maxpool2d(x, k=2):

    return tf.nn.max_pool(x, ksize=[1, k, k, 1], strides=[1, k, k, 1],
padding='SAME')


def conv_net(x, dropout):

        # Tensor input become 4-D: [Batch Size, Height, Width, Channel]
        x = tf.reshape(x, shape=[-1, 28, 28, 1])

        conv1 = conv2d(x, weights_cnn['wc1'], biases_cnn['bc1'])
        conv1 = maxpool2d(conv1, k=2)

        conv2 = conv2d(conv1, weights_cnn['wc2'], biases_cnn['bc2'])
        conv2 = maxpool2d(conv2, k=2)

        fc1 = tf.reshape(conv2, [-1,
weights_cnn['wd1'].get_shape().as_list()[0]])
        fc1 = tf.add(tf.matmul(fc1, weights_cnn['wd1']), biases_cnn['bd1'])
        fc1 = tf.nn.relu(fc1)

        fc1 = tf.nn.dropout(fc1, dropout)

        out = tf.add(tf.matmul(fc1, weights_cnn['out']), biases_cnn['out'])

        return out


def cnn_cost_fn(logits, y):

    loss_op =
tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits,
labels=y))
    optimizer =
tf.train.AdamOptimizer(learning_rate=cnn_param['learning_rate'])
    train_op = optimizer.minimize(loss_op)

    return train_op, loss_op
```

```
def cnn_model_fn(x, y, dropout):

    logits = conv_net(x, dropout)
    prediction = tf.nn.softmax(logits)
    train_op, loss_op = cnn_cost_fn(logits, y)
    accuracy = cnn_evaluation_fn(prediction, y)

    return train_op, loss_op, accuracy
```

<AutoEncoder>

```
from Lib import *


def encoder(x):

    layer_1 = tf.nn.sigmoid(tf.add(tf.matmul(x, weights_ae['encoder_h1']),
                                   biases_ae['encoder_b1']))
    layer_2 = tf.nn.sigmoid(tf.add(tf.matmul(layer_1, weights_ae['encoder_h2']),
                                   biases_ae['encoder_b2']))
    return layer_2


def decoder(x):

    layer_1 = tf.nn.sigmoid(tf.add(tf.matmul(x, weights_ae['decoder_h1']),
                                   biases_ae['decoder_b1']))
    layer_2 = tf.nn.sigmoid(tf.add(tf.matmul(layer_1, weights_ae['decoder_h2']),
                                   biases_ae['decoder_b2']))

    return layer_2


def ae_cost_fn(y_true, y_pred):

    loss_op = tf.reduce_mean(tf.pow(y_true - y_pred, 2))
    optimizer = tf.train.RMSPropOptimizer(ae_param['learning_rate'])
    train_op = optimizer.minimize(loss_op)

    return train_op, loss_op


def ae_model_fn(x):

    encoder_op = encoder(x)
    y_true, y_pred = x, decoder(encoder_op)
```

```
    train_op, loss_op = ae_cost_fn(y_true, y_pred)

    return train_op, loss_op, y_pred
```

We also save the model for the future training. Each model store in different folder. Some results are shown below.

<SNN>

Model restored from file: Model/SNN/SNN.ckpt
Step 1, Minibatch Loss= 2.0194, Training Accuracy= 0.953
Step 100, Minibatch Loss= 3.7523, Training Accuracy= 0.961
Step 200, Minibatch Loss= 2.9967, Training Accuracy= 0.977
Step 300, Minibatch Loss= 0.6676, Training Accuracy= 0.984
Step 400, Minibatch Loss= 6.8569, Training Accuracy= 0.938
Step 500, Minibatch Loss= 0.7962, Training Accuracy= 0.977
Step 600, Minibatch Loss= 1.6282, Training Accuracy= 0.977
Step 700, Minibatch Loss= 0.0035, Training Accuracy= 1.000
Step 800, Minibatch Loss= 0.6187, Training Accuracy= 0.992
Step 900, Minibatch Loss= 1.7825, Training Accuracy= 0.977
Step 1000, Minibatch Loss= 0.0368, Training Accuracy= 0.992
Training Finished!
Testing Accuracy: 0.9447

<CNN>

Model restored from file: Model/CNN/CNN.ckpt
Step 1, Minibatch Loss= 0.2174, Training Accuracy= 0.992
Step 100, Minibatch Loss= 4.4564, Training Accuracy= 0.984
Step 200, Minibatch Loss= 6.0015, Training Accuracy= 0.992
Step 300, Minibatch Loss= 11.2433, Training Accuracy= 0.984
Step 400, Minibatch Loss= 30.3554, Training Accuracy= 0.977
Step 500, Minibatch Loss= 13.1715, Training Accuracy= 0.977
Step 600, Minibatch Loss= 0.0000, Training Accuracy= 1.000
Step 700, Minibatch Loss= 7.5031, Training Accuracy= 0.992
Step 800, Minibatch Loss= 9.0606, Training Accuracy= 0.992
Step 900, Minibatch Loss= 12.8153, Training Accuracy= 0.992
Step 1000, Minibatch Loss= 11.6827, Training Accuracy= 0.977
Training Finished!
Testing Accuracy: 0.98046875

<AE>

Model restored from file: Model/AE/AE.ckpt

Step 1, Minibatch Loss= 0.0406

Step 100, Minibatch Loss= 0.0368

Step 200, Minibatch Loss= 0.0391

Step 300, Minibatch Loss= 0.0395

Step 400, Minibatch Loss= 0.0388

Step 500, Minibatch Loss= 0.0367

Step 600, Minibatch Loss= 0.0386

Step 700, Minibatch Loss= 0.0372

Step 800, Minibatch Loss= 0.0362

Step 900, Minibatch Loss= 0.0365

Step 1000, Minibatch Loss= 0.0389

Step 1100, Minibatch Loss= 0.0370

Step 1200, Minibatch Loss= 0.0381

Step 1300, Minibatch Loss= 0.0342

Step 1400, Minibatch Loss= 0.0394

Step 1500, Minibatch Loss= 0.0374

Step 1600, Minibatch Loss= 0.0357

Step 1700, Minibatch Loss= 0.0342

Step 1800, Minibatch Loss= 0.0335

Step 1900, Minibatch Loss= 0.0371

Step 2000, Minibatch Loss= 0.0345

Step 2100, Minibatch Loss= 0.0344

Step 2200, Minibatch Loss= 0.0333

Step 2300, Minibatch Loss= 0.0345

Step 2400, Minibatch Loss= 0.0330

Step 2500, Minibatch Loss= 0.0322

Step 2600, Minibatch Loss= 0.0341

Step 2700, Minibatch Loss= 0.0318

Step 2800, Minibatch Loss= 0.0297

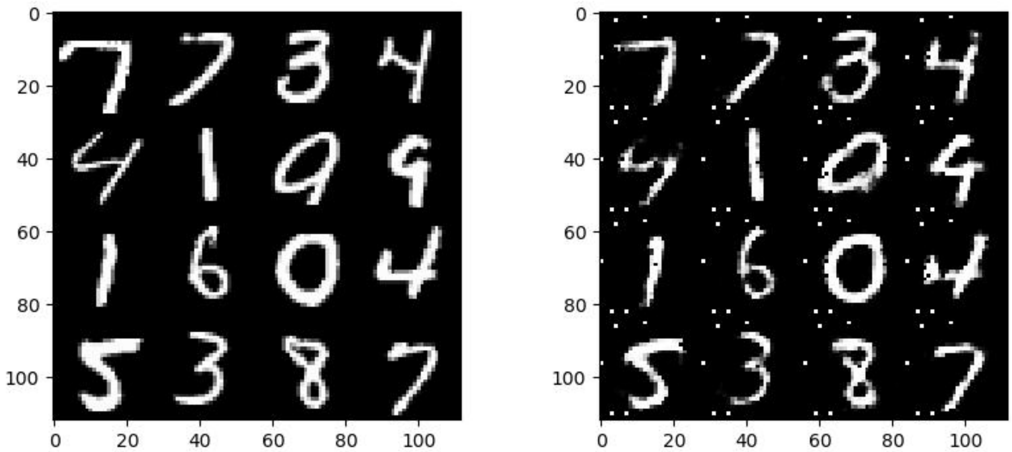
Step 2900, Minibatch Loss= 0.0312

Step 3000, Minibatch Loss= 0.0321

Training Finished!

Original Images

Reconstructed Images


0
20
40
60
80
100
0 20 40 60 80 100
0
20
40
60
80
100
0 20 40 60 80 100