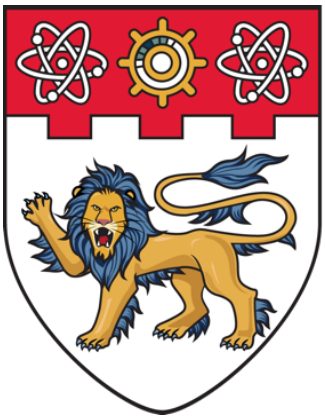


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

# CZ042 NEURAL NETWORKS
# ASSIGNMENT 2


ZHANG WANLU U1420638G
ZHU YIMIN U1420701C

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

# 1 Part A

## 1.1 Introduction

Part A of this project aims to implement a convolutional neural networks (CNN) for object recognition in images. A CNN contains an input layer, some convolutional layers, some pooling layers, some fully connected hidden layers and an output layer.

## 1.2 Notes

For Part 1, the main libraries we use are Theano, numpy and matplotlib. All the training and performance measurements are done on a notebook with 4.0 GHz Intel Core i7-6700k and 32GB memory.
There is a comparison for the training cost and test accuracy of each question in the discussion part.

## 1.3 Question 1

The first question is to design a convolutional neural network to recognize hand-written digits from MNIST dataset.

- An input layer of 28x28 dimensions.
- A convolution layer C1 of 15 feature maps and filters of window size 9x9. A max pooling layer S1 with a pooling window of size 2x2.
- A convolution layer C2 of 20 feature maps and filters of window size 5x5. A max pooling layer S2 with a pooling window of size 2x2.
- A fully connected layer F3 of size 100.
- A softmax layer F4 of size 10.

ReLu activation function and mini batch gradient descent learning are applied here. Parameters: *batch_size*=128 learning rate $\alpha = 0.05$ decay $\beta = 10^{-4}$.

### 1.3.1 Method

First of all, we use the following functions to initialize the weights and biases randomly. The first function is for convolutional layers and the second one is for fully connected layer the softmax layer.

```
def init_weights_bias4(filter_shape, d_type):
  fan_in = np.prod(filter_shape[1:])
  fan_out = filter_shape[0] * np.prod(filter_shape[2:])

  bound = np.sqrt(6. / (fan_in + fan_out))
  w_values = np.asarray(
      np.random.uniform(low=-bound, high=bound, size=filter_shape),
      dtype=d_type)
  b_values = np.zeros((filter_shape[0],), dtype=d_type)
  return theano.shared(w_values,borrow=True), theano.shared(b_values, borrow=True)

def init_weights_bias2(filter_shape, d_type):
  fan_in = filter_shape[1]
  fan_out = filter_shape[0]

  bound = np.sqrt(6. / (fan_in + fan_out))
  w_values = np.asarray(
      np.random.uniform(low=-bound, high=bound, size=filter_shape),
```

```
        dtype=d_type)
    b_values = np.zeros((filter_shape[1],), dtype=d_type)
    return theano.shared(w_values,borrow=True), theano.shared(b_values, borrow=True)
```

Then we use the following function to define the CNN model. The function takes in the original input and weight and bias of each layer then return the output and feature maps of each layer. The activation function is ReLu, which is defined as:$f(x) = max(0, x)$. The activation is simply thresholded at zero. Compared to *tanh/sigmoid* functions, it can greatly accelerate the convergence of SGD. However, the downside of ReLU is that it outputs zero and has zero derivatives for all negative inputs. So, if the weights in the network always lead to negative inputs into a ReLU neuron, that neuron is effectively not contributing to the network's training since mathematically, the gradient contribution to the weight updates coming from that neuron is always zero.

```
def model(X, w1, b1, w2, b2, w3, b3, w4, b4):
    pool_dim = (2, 2)
    # First convolution layer C1 and pooling layer S1
    y1 = T.nnet.relu(conv2d(X, w1) + b1.dimshuffle('x', 0, 'x', 'x'))
    o1 = pool.pool_2d(y1, pool_dim, ignore_border=True)

    # Second convolution layer C2 and pooling layer S2
    y2 = T.nnet.relu(conv2d(o1, w2) + b2.dimshuffle('x', 0, 'x', 'x'))
    o2 = pool.pool_2d(y2, pool_dim, ignore_border=True)

    # Input for fully connected layer
    fcinput = T.flatten(o2, outdim=2)
    # Fully connected layer F3 of size 100
    fcoutput = T.nnet.relu(T.dot(fcinput, w3) + b3)

    # Softmax layer F4 of size 10
    pyx = T.nnet.softmax(T.dot(fcoutput, w4) + b4)
    return y1, o1, y2, o2, pyx
```

Before training, the data is imported from function *mnist()* from *load.py*. Then the parameters are defined according to the requirements of question 1. First question we use SGD to calculate gradients for updates. Notice that, the defined Theano functions *test1* and *test2* are used to plot the feature maps from each convolution layer and pooling layer.

```
trX, teX, trY, teY = mnist(onehot=True)
trX = trX.reshape(-1, 1, 28, 28)
teX = teX.reshape(-1, 1, 28, 28)
trX, trY = trX[:12000], trY[:12000]
teX, teY = teX[:2000], teY[:2000]

X = T.tensor4('X')
Y = T.matrix('Y')

batch_size = 128
learning_rate = 0.05
decay = 1e-4
epochs = 100
num_filters_1 = 15
num_filters_2 = 20
fcsize = 100
smsize = 10

w1, b1 = init_weights_bias4((num_filters_1, 1, 9, 9), X.dtype)
w2, b2 = init_weights_bias4((num_filters_2, num_filters_1, 5, 5), X.dtype)
w3, b3 = init_weights_bias2((num_filters_2*3*3, fcsize), X.dtype)
w4, b4 = init_weights_bias2((fcsize, smsize), X.dtype)

y1, o1, y2, o2, py_x = model(X, w1, b1, w2, b2, w3, b3, w4, b4)
y_x = T.argmax(py_x, axis=1)

cost = T.mean(T.nnet.categorical_crossentropy(py_x, Y))
```

```
params = [w1, b1, w2, b2, w3, b3, w4, b4]
updates = sgd(cost, params, lr=learning_rate, decay=decay)

train = theano.function(inputs=[X, Y], outputs=cost, updates=updates,
    allow_input_downcast=True)
predict = theano.function(inputs=[X], outputs=y_x, allow_input_downcast=True)
test1 = theano.function(inputs = [X], outputs=[y1, o1], allow_input_downcast=True)
test2 = theano.function(inputs = [X], outputs=[y2, o2], allow_input_downcast=True)
```

The following part is trainning process. Before training the data is shuffled. The list *costA* is for ploting use.

```
for i in range(epochs):
  trX, trY = shuffle_data (trX, trY)
  teX, teY = shuffle_data (teX, teY)
  cost = 0.0
  for start, end in zip(range(0, len(trX), batch_size), range(batch_size, len(trX),
    batch_size)):
    cost = np.append(cost, train(trX[start:end], trY[start:end]))
  costA = np.append(costA, np.mean(cost))
  a = np.append(a, np.mean(np.argmax(teY, axis=1) == predict(teX)))
  print('No.%2d Accuracy = %f'%(i+1,a[i]))
```

### 1.3.2 Result

The test accuracy and training cost are shown as follow. As we can see the accuracy reach 96% after about 20 epochs. However, there are still some oscillations after 60 iterations. Since we trained the network 100 iterations, the training cost decreased to around 0.001.

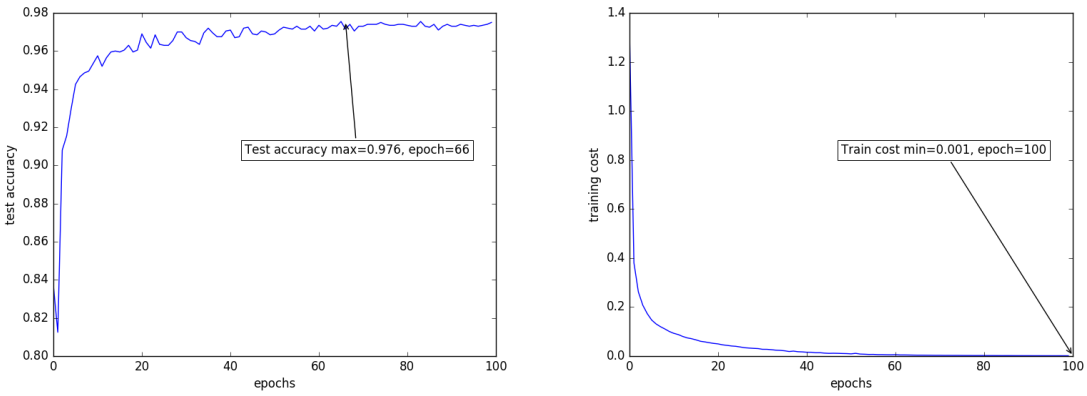
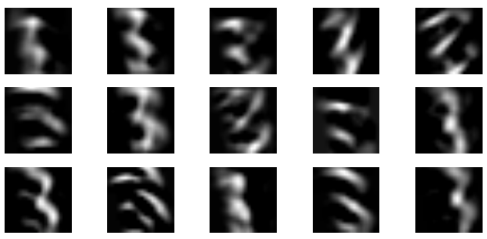


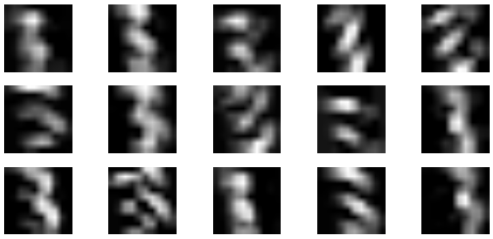

Figure 1: Test accuracy and training cost for Q1

The figure below is one of the test patterns, followed by its corresponding feature maps from each layer.

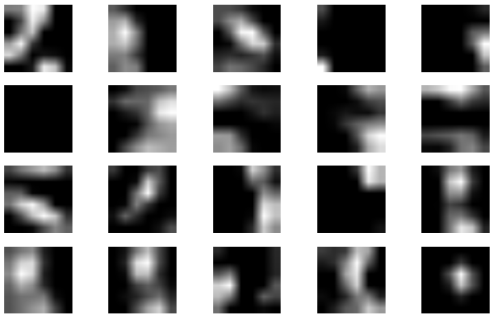
Figure 2: Input 1 for Q1



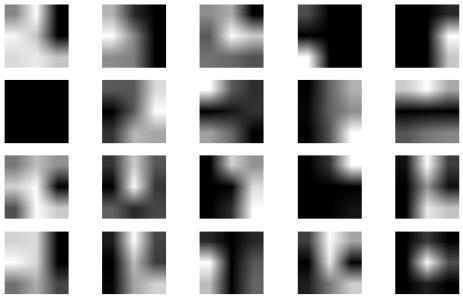
(a) First convolution layer



(b) First pooling layer



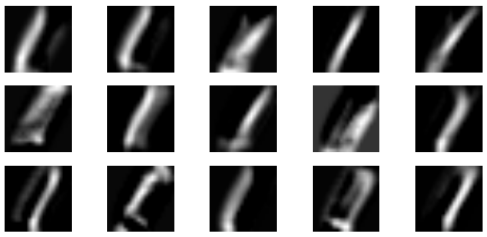
(c) Second convolution layer



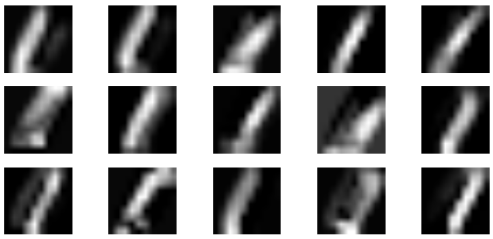
(d) Second pooling layer

The figure below is another test pattern followed by its corresponding feature maps from each layer.

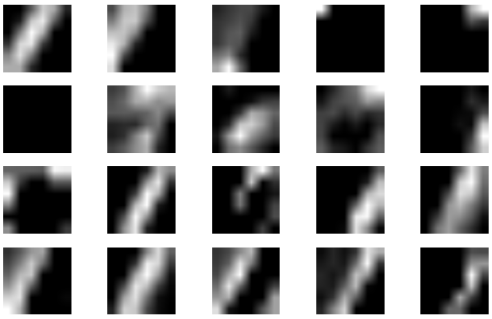
Figure 4: Input 2 for Q1



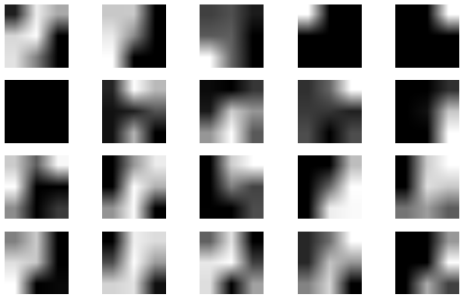
(a) First convolution layer



(b) First pooling layer



(c) Second convolution layer



(d) Second pooling layer

After flowing through the first convolutional layer, 15 feature maps are produced by the filters. Then we use a max pooling layer with a 2x2 pooling window to do down-sampling. The pooling layers reduce the spatial size of the feature maps. By doing so, the amount of parameters and computational effort needed in the network are also reduced. The output of pooling layer has translation invariant, which means the features remain even some pixels move a little from original position. The second convolutional and pooling layers are similar. However, we can find that there is one feature map from second convolutional layer is totally black. It remains black after pooling. This kernel is deactivated on the input.

## 1.4 Question 2

The question 2 is to repeat part 1 by adding momentum term to mini batch gradient descent learning. The momentum parameter is set as $\gamma = 0.1$.

### 1.4.1 Method

For question 2, most part of code is same as the code for question 1. We use momentum update instead of normal SGD update. The formula of momentum update:

$$V = \gamma V - \alpha \frac{\partial J}{\partial W}$$

$$W = W + V$$

V is the velocity term and has same dimension as the weight vector W. The momentum term $\gamma \in [0, 1]$ indicates how many iterations the previous gradients are incorporated into current state. It often initially set to 0.1 until the learning is stable and increased to 0.9 then.

```
def sgd_momentum(cost, params, lr=0.05, decay=0.0001, momentum=0.5):
  grads = T.grad(cost=cost, wrt=params)
  updates = []
  for p, g in zip(params, grads):
    v = theano.shared(p.get_value()*0.)
    v_new = momentum*v - (g + decay*p) * lr
    updates.append([p, p + v_new])
    updates.append([v, v_new])
  return updates
```

In ideal condition, it's guaranteed to find the global minimum. However, the real error surface is more complex and has many local minimum. To avoid this, we can add a momentum term to the function. Momentum term is a simulation of real world. It's like a ball falling downhill. The ball starts by following the gradient, but it no longer does the steepest descent once it has velocity. The momentum of it will keep it in the previous direction. By combining the previous directions and gradients, it damps the oscillations toward minimum. At the beginning of learning, there may be very large gradients so the momentum is small. Once the large gradients disappear and weights get stuck, the momentum should be increased to 0.9, its final value.

### 1.4.2 Result

The test accuracy and training cost are shown as follow. As we can see, test accuracy is higher than normal SGD and the convergence comes earlier. Training cost seems no difference.

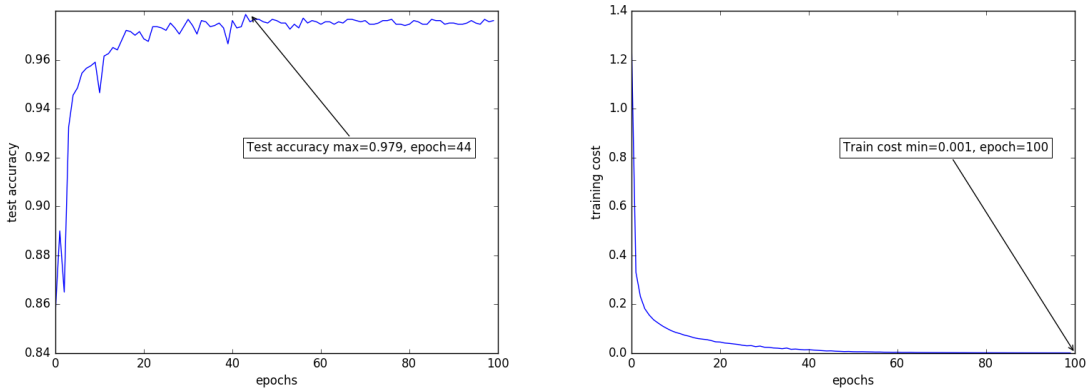



Figure 6: Test accuracy and training cost for Q2

The figure below is one of the test patterns, followed by the feature maps from each layer.

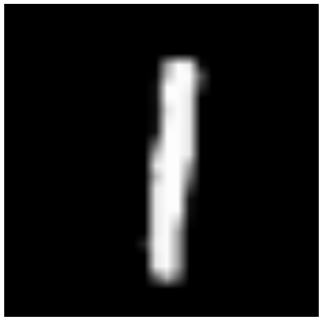
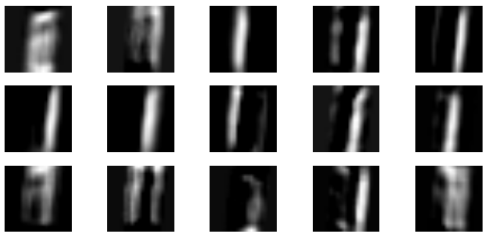
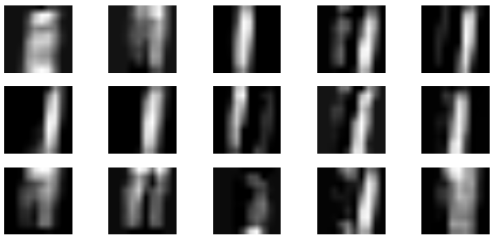


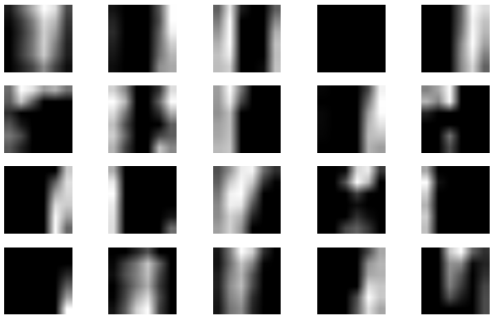
Figure 7: Input 1 for Q2



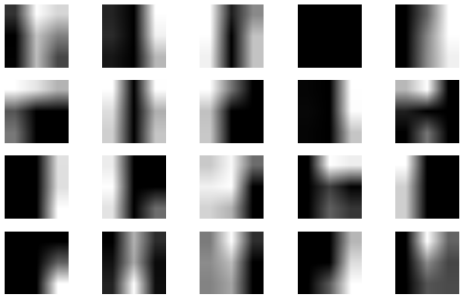
(a) First convolution layer



(b) First pooling layer



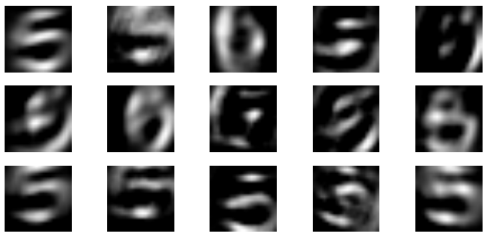
(c) Second convolution layer



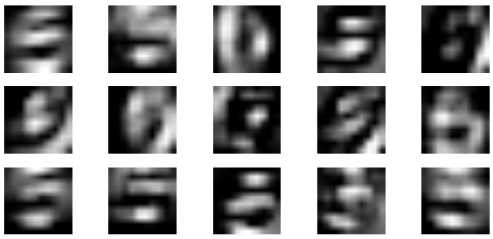
(d) Second pooling layer

This figure is another test pattern, followed by the feature maps from each layer.

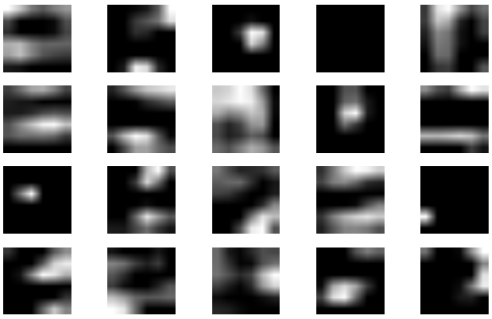
Figure 9: Input 2 for Q2



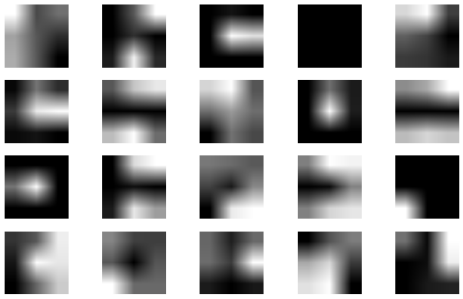
(a) First convolution layer



(b) First pooling layer



(c) Second convolution layer



(d) Second pooling layer

## 1.5 Question 3

The question 3 uses RMSProp algorithm for learning. Parameters: $\alpha = 0.001$, $\beta = 10^{-4}$, $\rho = 0.9$, $\epsilon = 10^{-6}$.

### 1.5.1 Method

For question 3, we use RMSProp instead of momentum update. In deep neural networks, the appropriate learning rates differs widely among the weights. So use a global learning rate multiplied by a local parameter is helpful to get better convergence performance. RMSProp is such a adaptive learning rate method. It keeps a moving average of squared gradient for each weight. RMSProp uses an exponentially decaying average to discard the history from extreme past which leads to a rapid convergence after finding a convex region.

$$r = \rho r + (1 - \rho)(\frac{\partial J}{\partial W})^2$$

$$W = W - \frac{\alpha}{\sqrt{\epsilon + r}} \cdot \frac{\partial J}{\partial W}$$

```
def RMSprop(cost, params, lr=0.001, decay=0.0001, rho=0.9, epsilon=1e-6):
  grads = T.grad(cost=cost, wrt=params)
  updates = []
  for p, g in zip(params, grads):
    acc = theano.shared(p.get_value() * 0.)
    acc_new = rho * acc + (1 - rho) * g ** 2
    gradient_scaling = T.sqrt(acc_new + epsilon)
    g = g / gradient_scaling
    updates.append((acc, acc_new))
    updates.append((p, p - lr * (g+ decay*p)))
  return updates
```

### 1.5.2 Result

The test accuracy and training cost are shown as follow. We can see that the test accuracy and convergence of RMSProp is best among the implemented three methods.

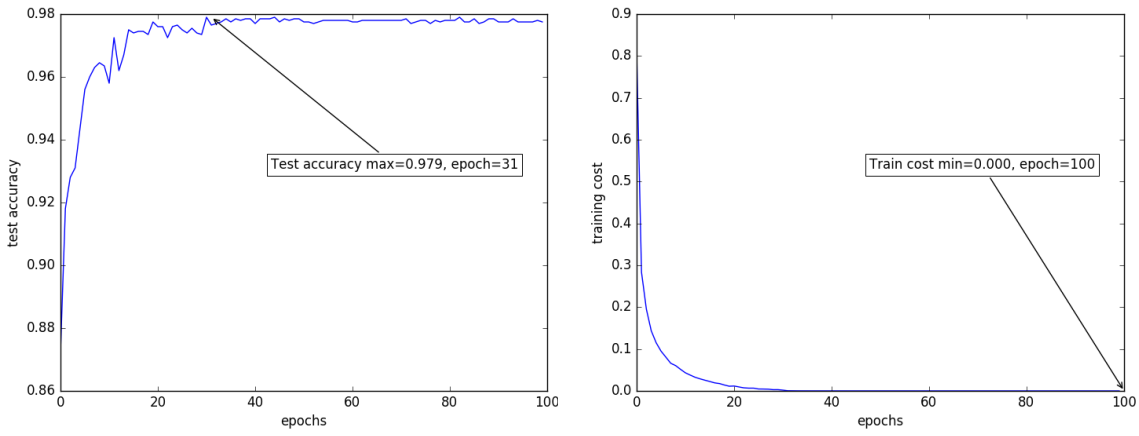
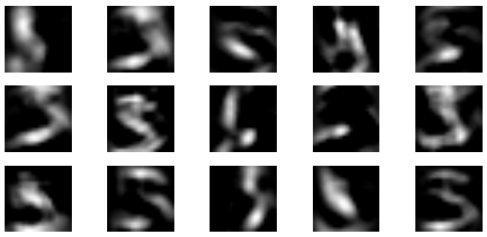


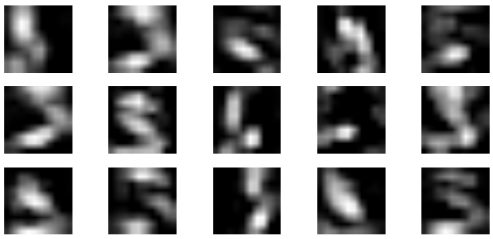

Figure 11: Test accuracy and training cost for Q3

The figure below is one of the test patterns, followed by the feature maps from each layer.

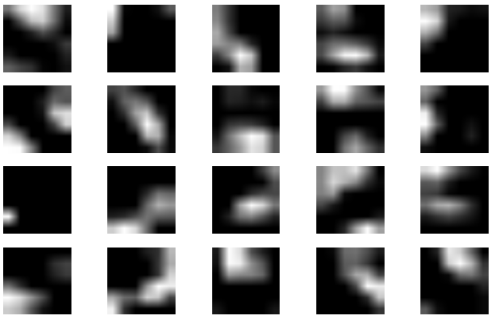
Figure 12: Input 1 for Q3



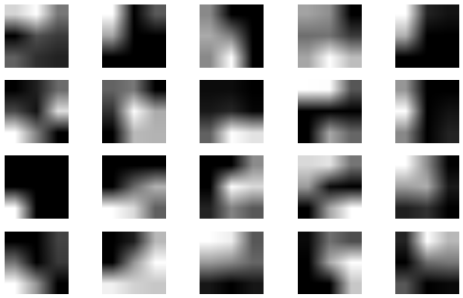
(a) First convolution layer



(b) First pooling layer



(c) Second convolution layer



(d) Second pooling layer

The figure below is another test pattern, followed by the feature maps from each layer.

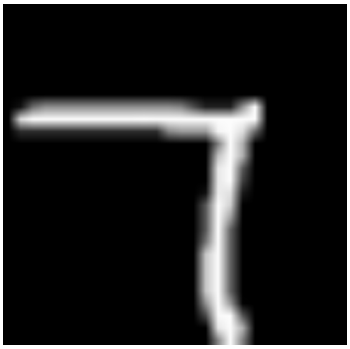
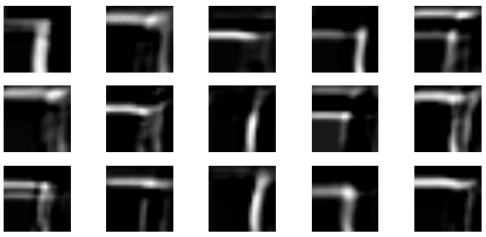
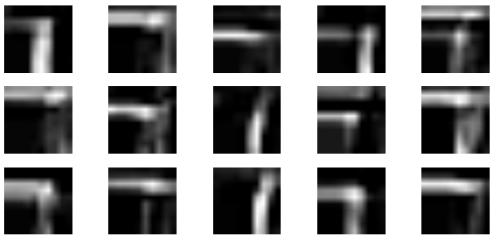


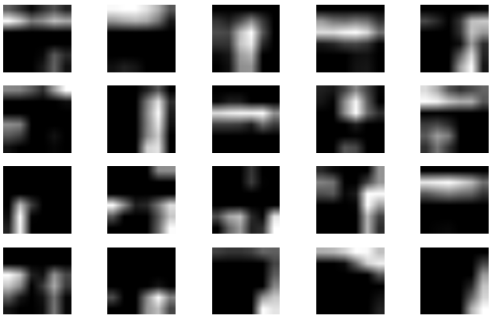
Figure 14: Input 2 for Q3



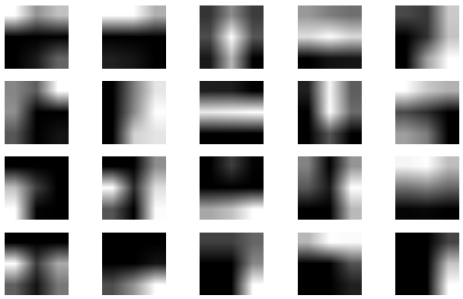
(a) First convolution layer



(b) First pooling layer



(c) Second convolution layer



(d) Second pooling layer

We can see that there is no deactivated kernel.

## 1.6 Discussion

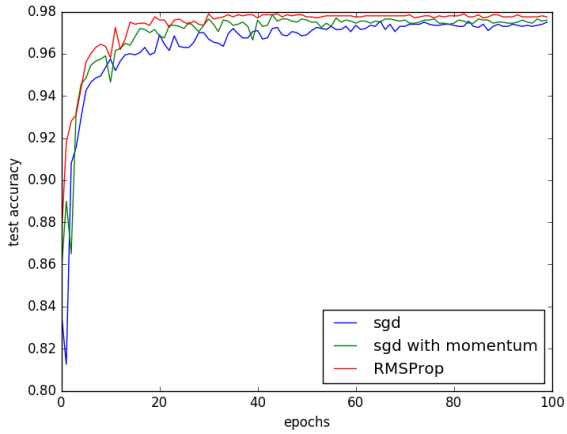



Figure 16: Comparison of test accuracy

According to the figure, we can easily know the performance of three methods. RMSProp outperforms, SGD with momentum second and normal SGD last. The reasons have been stated in previous parts. However, they are quite close to each other probably due to small dataset since we only use part of MNIST dataset to train the network.

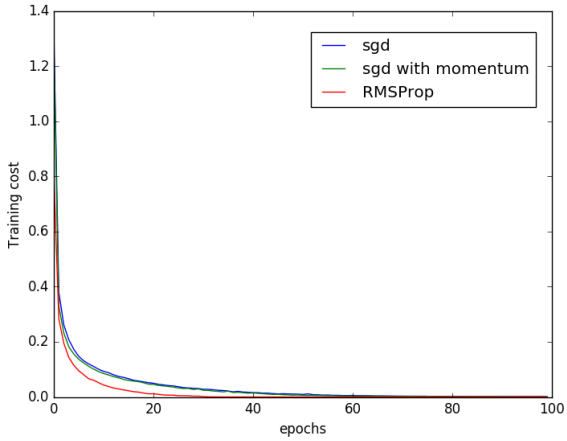



Figure 17: Comparison of training cost

The cost of normal SGD and momentum method are similar while the RMSProp is better than them. This is because the RMSProp algorithm converging faster when a convex region is found.
Momentum and RMSProp are designed to solve two problems of normal SGD. one problem is SGD tends to have unstable update step in each iteration which leads to more converging time and stuck in local minimum. Momentum is introduced to solve this by combining the history direction and next gradient. Another problem is that a outlier instance will give a large opposite gradient which will remove some gradient information before. To solve it, RMSProp introduces adaptive learning rate for different weight parameters. The idea is to divide the learning rate for a weight by a running average of the magnitudes of recent gradients for that weight.
Furthermore, we can combine these two methods together to get better performance.

# 2 Part B

## 2.1 Introduction

Part B of the project is mainly to understand the implementation and usage of autoencoders. An autoencoder neural network applies unsupervised learning and tries to set the target to be equal to its inputs. A simple autoencoder contains an input layer, a hidden layer and an output layer. The first two layers make up an encoder which is used to generate a code from the input. The last two layers is called a decoder which form the output based on the code. The loss of an autoencoder is defined as the similarities between the real input and the generated one, which is the output. The figure below is a sample autoencoder.

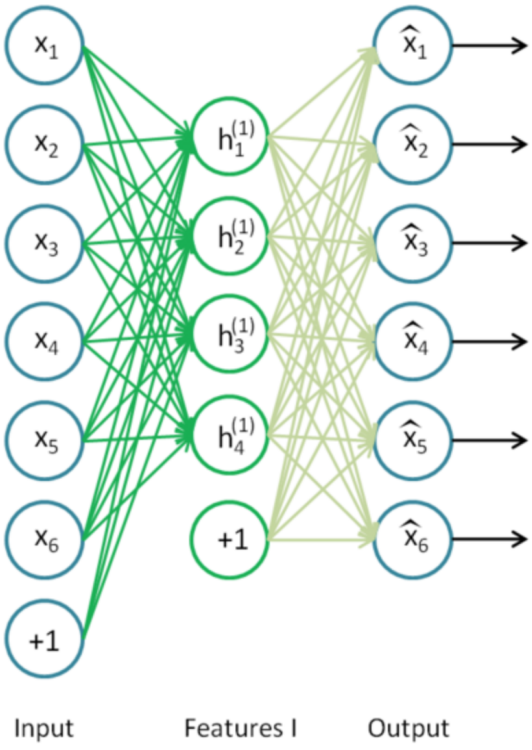



Figure 18: Autoencoder

## 2.2 Notes

For part B, the main libraries used are Theano, numpy and pylab. All the training and performance measurements are done on a macbook pro with 2.6 GHz Intel Core i5 and 8GB memory. The dataset used in this part is MNIST and the parameters are given the following values,
corruption level = 0.1 training epochs = 25, learning rate = 0.1, batch size = 128

## 2.3 Question 1

In this section, we designed a stacked denoising autoencoder with three hidden layers, each consists of 900, 625, 400 neurons respectively.

### 2.3.1 Method

Initialize the weights and biases for creating three autoencoders. Each of them is used for training a specific hidden layer. Constrain the reverse mapping by setting the weight of decoder to be the transpose of the weight of encoder. Note that the input data from MNIST has a shape of 28*28.

```
# 3 hidden layers
# stacked autoencoder
W1 = init_weights(28*28, 900)
b1 = init_bias(900)
b1_prime = init_bias(28*28)
W1_prime = W1.transpose()

W2 = init_weights(900, 625)
b2 = init_bias(625)
b2_prime = init_bias(900)
W2_prime = W2.transpose()

W3 = init_weights(625, 400)
b3 = init_bias(400)
b3_prime = init_bias(625)
W3_prime = W3.transpose()
```

Corrupt the input data using a binomial distribution. Then pass the tuned value to calculate the synaptic input. As shown in the following code, y1 is the encoded result and z1 is the decoded or generated one. We wish the reconstructed z1 and the original input, x, to be as similar as possible. Here we use the cross-entropy as cost function and gradient descent as optimizer.

```
# train first hidden layer
# corrupting inputs -- binomial distribution
tilde_x = theano_rng.binomial(size=x.shape, n=1, p=1 - corruption_level,
                              dtype=theano.config.floatX)*x
y1 = T.nnet.sigmoid(T.dot(tilde_x, W1) + b1)
z1 = T.nnet.sigmoid(T.dot(y1, W1_prime) + b1_prime)
# cross-entropy
cost1 = - T.mean(T.sum(x * T.log(z1) + (1 - x) * T.log(1 - z1), axis=1))

params1 = [W1, b1, b1_prime]
grads1 = T.grad(cost1, params1)
updates1 = [(param1, param1 - learning_rate * grad1)
            for param1, grad1 in zip(params1, grads1)]
train_da1 = theano.function(inputs=[x], outputs = cost1, updates = updates1,
    allow_input_downcast = True)
```

The training procedure is shown as follow. The parameter *train_da* is the training model and value passed to the model is in a form of mini batches.

```
def plot_learning_curves(layer_no, train_da):
    reconstruction_err = []
    for epoch in range(training_epochs):
        # go through training set
        cost = []
        # mini-batch
        for start, end in zip(range(0, len(trX), batch_size), range(batch_size, len(
    trX), batch_size)):
            cost.append(train_da(trX[start:end]))
        reconstruction_err.append(np.mean(cost, dtype='float64'))
        print(reconstruction_err[-1])

    # plot the learning curves for layer
    pylab.figure()
    pylab.plot(range(training_epochs), reconstruction_err)
    annotate_min(pylab, reconstruction_err)
    pylab.xlabel('iterations')
    pylab.ylabel('cross-entropy')
```

```
    pylab.savefig('learning_curve_'+str(layer_no)+'.png')
    pylab.show()
```

After training the first hidden layer, use the primary feature obtained by this layer as the raw input of the second autoencoder, as shown in the figure below. Then compare the decoded output with the primary feature to compute the cost. Similar approach is conducted for training the third hidden layer.

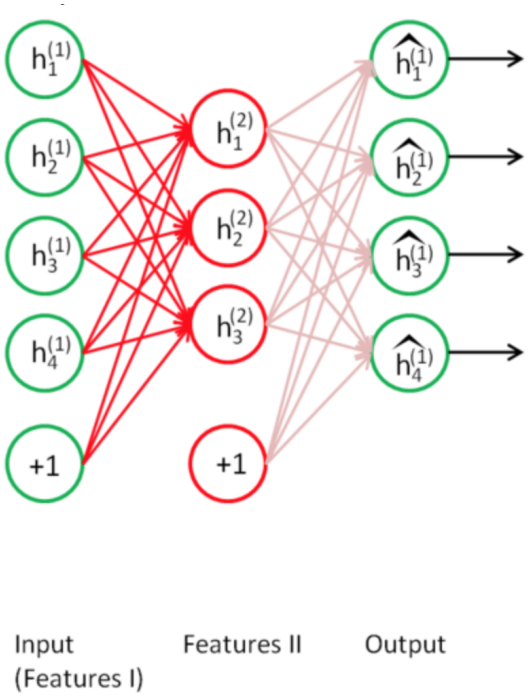



Figure 19: Learn secondary features

When finish training each of the hidden layer, a stacked autoencoder is completed. We random selecte 100 sample images from test dataset and pass to the autoencoder. The *regenerated* output could be seen in the result section.

```
encoder1 = theano.function(inputs=[x], outputs = y1, allow_input_downcast=True)
encoder2 = theano.function(inputs=[y1], outputs = y2, allow_input_downcast=True)
encoder3 = theano.function(inputs=[y2], outputs = y3, allow_input_downcast=True)

decoder3 = theano.function(inputs=[y3], outputs = z3, allow_input_downcast=True)
decoder2 = theano.function(inputs=[y2], outputs = z2, allow_input_downcast=True)
decoder1 = theano.function(inputs=[y1], outputs = z1, allow_input_downcast=True)

# reconstructed images
regenerated = decoder1(decoder2(decoder3(encoder3(encoder2(encoder1(teX[idx,:]))))))
```

### 2.3.2 Result and Discussion

For each hidden layer, we implement a corresponding autoencoder and the learning curves for each hidden layer is shown as follow.

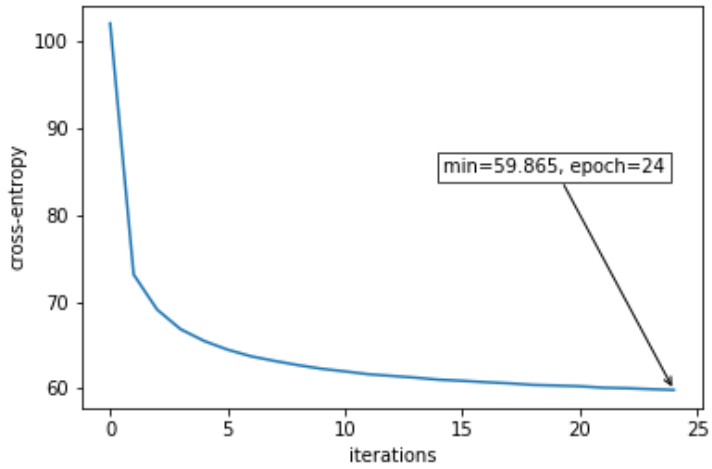



Figure 20: Learning Curve of Hidden Layer 1

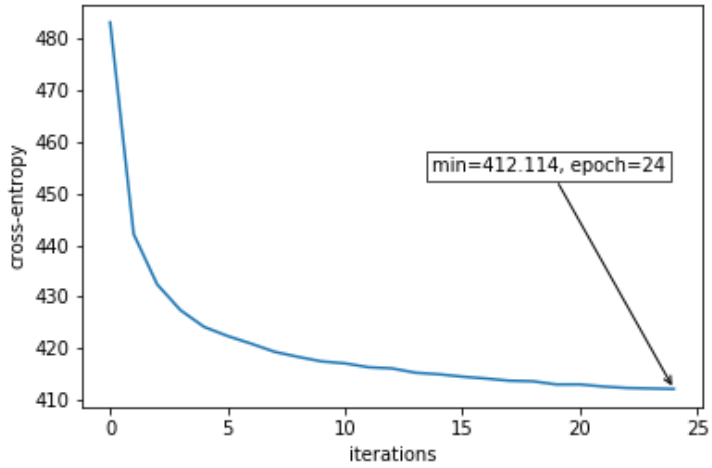



Figure 21: Learning Curve of Hidden Layer 2

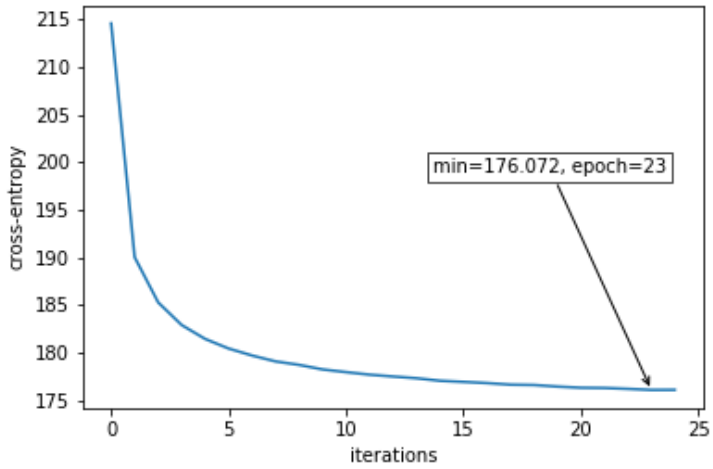



Figure 22: Learning Curve of Hidden Layer 3

As we can see, the cross-entropy loss of the first hidden layer reduces to around 60 while the loss for the second layer suddenly becomes above 400 and the third layer drops a lot but still quite high. After analyzing the network, we think the relatively low cost for the first one might because the first one is an **overcomplete autoencoder** since the hidden layer, 900 neurons, has a higher dimension than the input, which is 28*28. However, the second one is an **undercomplete autoencoder** with an input dimension of 900 and hidden layer of 600. Thus the learning of decoding or reconstructing back to the original input may not as good as the first one since it might learn to capture all the significant features while ignore some trivial ones. With the similar explanation, the third one is also an **undercomplete autoencoder** but with a relatively smaller difference between the input(625) and hidden layer(400).

Sometimes looking at the learned coefficients of a neural network can provide insight into the learning behavior. Thus we pick 100 sample weights from each layer and plot the learned wights as images shown as follow,

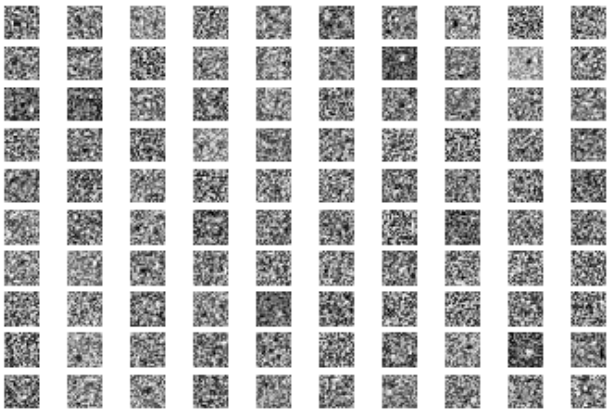


Figure 23: Learned Weights of Hidden Layer 1

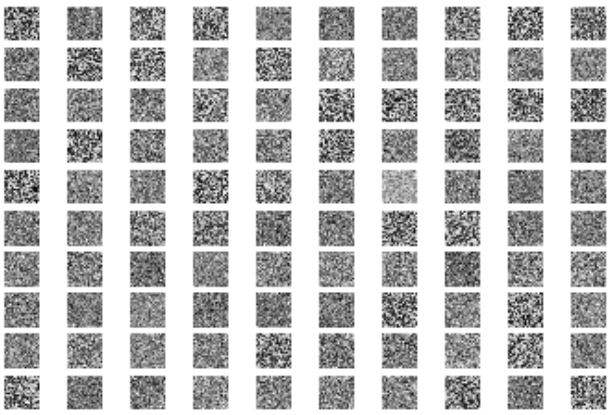


Figure 24: Learned Weights of Hidden Layer 2

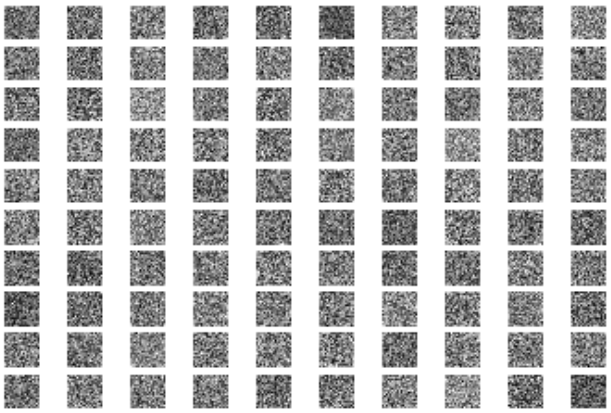
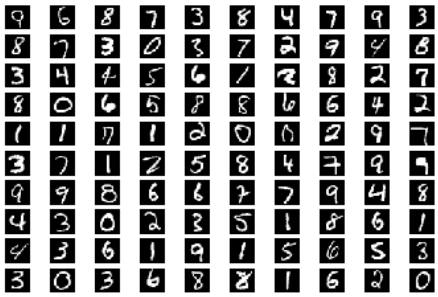
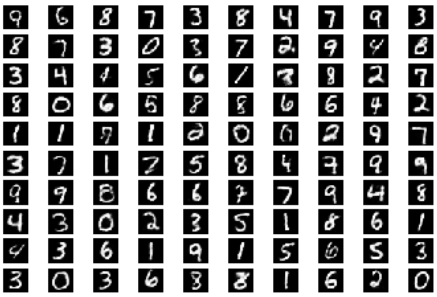


Figure 25: Learned Weights of Hidden Layer 3

In order to test the functionality of our stacked autoencoder, we randomly pick 100 images from the test set and feed them to the network, to see how is the quality of the reconstructed images. Meanwhile, we also try to plot the activation of each hidden layer to analyze the network performance. The regenerated figure and the activations are shown below. We can see that the stacked autoencoder actually performs quite well. Although the reconstructed images are a bit blur, it preserves all the major components for digit recognition.



(a) Original Figure



(b) Reconstructed Figure

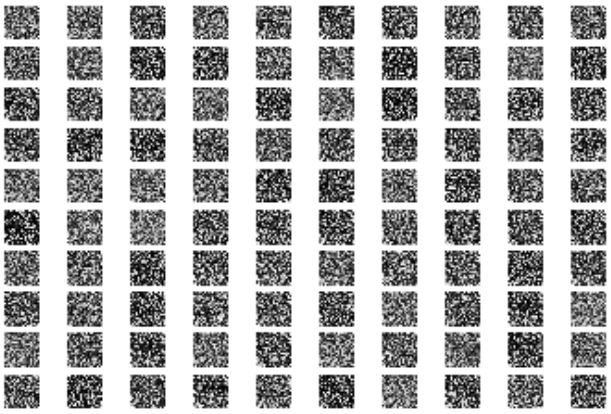


Figure 27: Activation of Hidden Layer 1

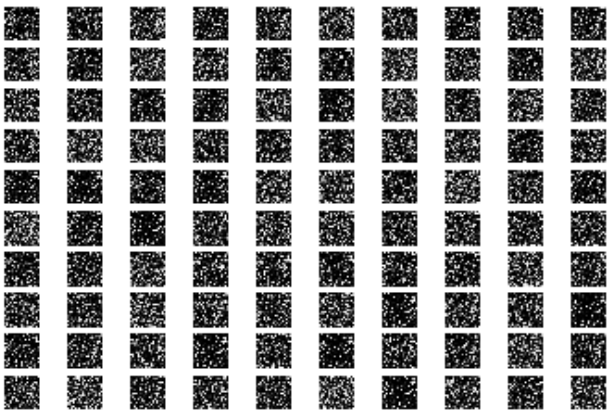


Figure 28: Activation of Hidden Layer 2

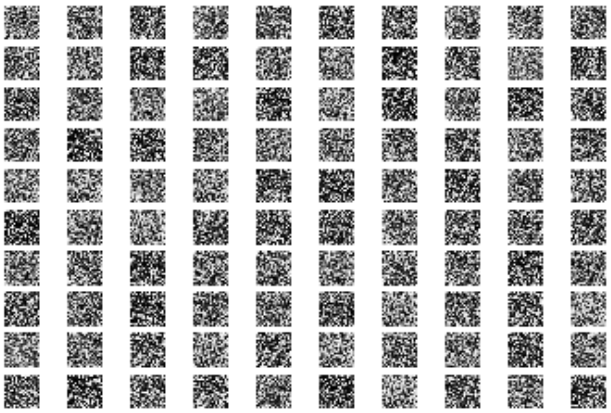


Figure 29: Activation of Hidden Layer 3

## 2.4 Question 2

In this section, we add a classification layer behind the previously trained hidden layers, and thus, form a five layer feedforward neural network.

### 2.4.1 Method

Add in a softmax layer as output layer into the network. Initialize the weights and biases for the output. As for the hidden layers, use the pretrained parameters obtained in the previous section. Since the predicted value should be a one hot vector, we use categorical_crossentropy to calculate the cost since we wish the final classification result to be in a one-hot manner. While training the classifier, all the parameters are updated using gradient descent for finding an optima.

```
W4 = init_weights(400, 10)
b4 = init_bias(10)

p_y = T.nnet.softmax(T.dot(y3, W4)+b4)
y4 = T.argmax(p_y, axis=1)
cost4 = T.mean(T.nnet.categorical_crossentropy(p_y, d))

params4 = [W1, b1, W2, b2, W3, b3, W4, b4]
grads4 = T.grad(cost4, params4)
updates4 = [(param4, param4 - learning_rate * grad4)
            for param4, grad4 in zip(params4, grads4)]
train_ffn = theano.function(inputs=[x, d], outputs = cost4, updates = updates4,
    allow_input_downcast = True)
test_ffn = theano.function(inputs=[x], outputs = y4, allow_input_downcast=True)
```

### 2.4.2 Result and Discussion

We train the network with MNIST-train dataset, then followed by a test on the model using MNIST-test dataset. The learning curve as well as the test accracy are shown as follow,

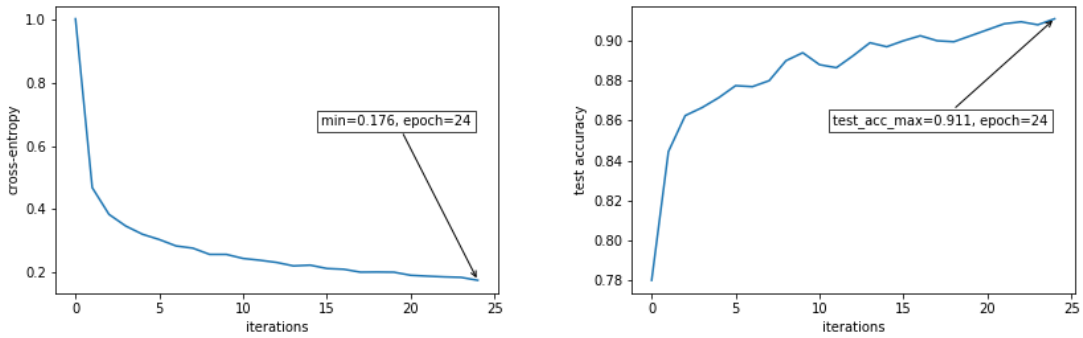



Figure 30: Training Cost and Test Accuracy for Q2

From the figure, we can see that although it didn't converge in the end, the overall trend is decreasing. Probably train for some more epochs would lead to a convergence in the end. The test result is also impressive. It keeps increasing and reaches an accuracy of 91.1%.

## 2.5 Question 3

Stochastic Gradient Descent has trouble navigating rains, which is an area with a much steeper dimension than the other one. And this is really common near local optima. Under this case, SGD will move in an inefficient zig-zag manner. The momentum algorithm is introduced to accelerate this procedure. The main idea is to accumulate an exponentially decaying moving average of past gradients and continue to move in their direction. The momentum formula is shown below. In this section, we introduce momentum term for gradient descent.

$$\boldsymbol{V} = \gamma \boldsymbol{V} - \alpha \frac{\partial J}{\partial \boldsymbol{W}}$$
$$\boldsymbol{W} = \boldsymbol{W} + \boldsymbol{V}$$

Normally for an autoencoder, if the hidden layer has higher dimension than the input, the hidden neurons could simply do nothing and just preserve all the input information trivially. Under this case, adding sparsity constraint on the hidden layer will force compression to happen so that the hidden layer will learn something interesting and capture the significant part of the data. In this section, we implement this feature by adding sparsity constraint to the cost function.
We use the previous section as the base line to see how these modifications could make the learning better. The parameter settings are given as follow,
momentum parameter gamma = 0.1
penalty parameter beta = 0.5
sparsity parameter rho = 0.05

### 2.5.1 Method

We define a *sgd_momentum* function which takes in the cost function and the parameters, returns a list of updates. The function is being called during training time when the weights and bias are being updated. We also modify each cost function by adding in sparsity constraint, which is to constraint the neurons at hidden layer to be inactive for most of the time.

```
1 def sgd_momentum(cost, params, lr=0.1,momentum=0.1):
2     grads = T.grad(cost=cost, wrt=params)
3     updates = []
```

```
    for p, g in zip(params, grads):
        v = theano.shared(p.get_value()*0.)
        v_new = momentum*v - g * lr
        updates.append([p, p + v_new])
        updates.append([v, v_new])
    return updates

# train first hidden layer
# corrupting inputs -- binomial distribution
tilde_x = theano_rng.binomial(size=x.shape, n=1, p=1 - corruption_level,
                              dtype=theano.config.floatX)*x
y1 = T.nnet.sigmoid(T.dot(tilde_x, W1) + b1)
z1 = T.nnet.sigmoid(T.dot(y1, W1_prime) + b1_prime)
# cross-entropy & sparsity constraint
cost1 = - T.mean(T.sum(x * T.log(z1) + (1 - x) * T.log(1 - z1), axis=1))\
            + beta*T.shape(y1)[1]*(rho*T.log(rho) + (1-rho)*T.log(1-rho)) \
            - beta*rho*T.sum(T.log(T.mean(y1, axis=0)+1e-6)) \
            - beta*(1-rho)*T.sum(T.log(1-T.mean(y1, axis=0)+1e-6))

params1 = [W1, b1, b1_prime]
grads1 = T.grad(cost1, params1)
train_da1 = theano.function(inputs=[x], outputs = cost1, updates = sgd_momentum(
    cost1, params1, learning_rate, gamma), allow_input_downcast = True)
```

### 2.5.2 Result and Discussion

The learning curves for each hidden layer is shown below. From the figures we realize that as for the second and third hidden layer, the overall loss decreases comparing with the ones in previous section, while the loss for the first layer increases a bit. This is because before implementing sparsity constraint, a overcomplete autoencoder could reconstruct the input fairly well by capturing all the necessary information, in this section, since we add in sparsity constraint, number of neurons that are in active state is reduced, and thus, regenerate the input becomes harder although for each neuron, the capability of learning improved. Despite this, the overall trend is still decreasing due to the function of momentum. The converging procedure is much faster comparing to before.

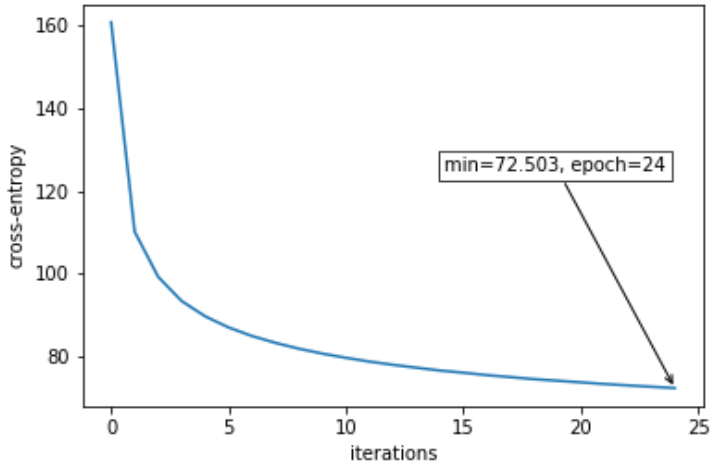



Figure 31: Learning Curve of Hidden Layer 1

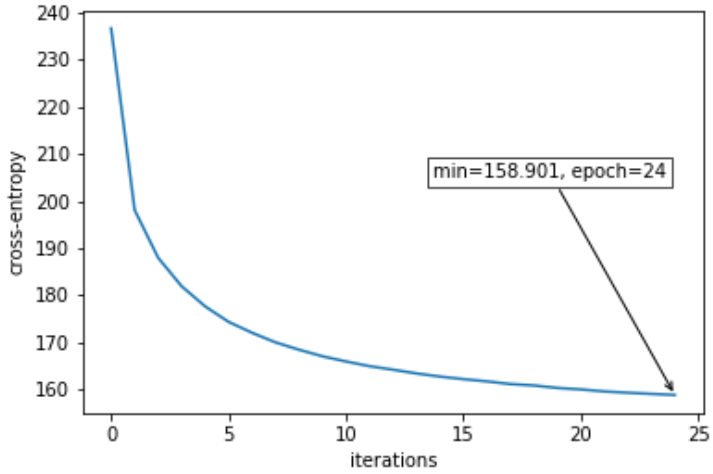



Figure 32: Learning Curve of Hidden Layer 2

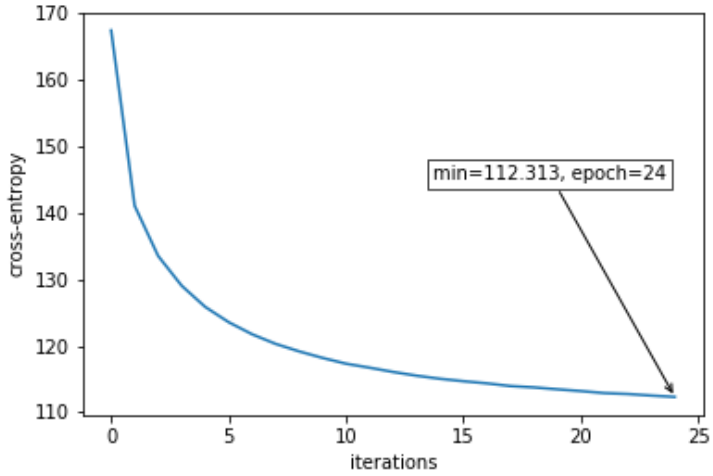



Figure 33: Learning Curve of Hidden Layer 3

We pick 100 sample weights from each layer and the learned wights are plotted as images shown as follow. From Figure 18 we could see some rough outlines of digits which indicating that the neurons are trying to learn the image representations.

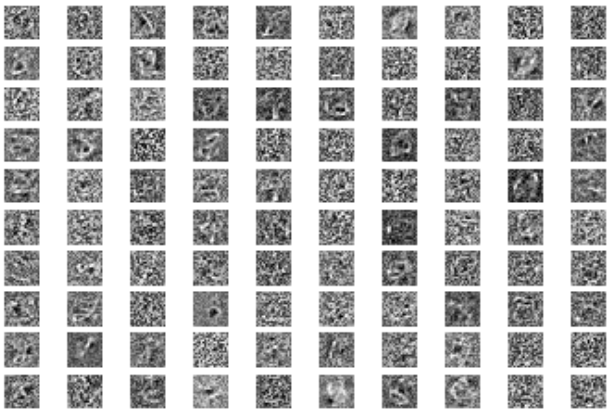


Figure 34: Learned Weights of Hidden Layer 1

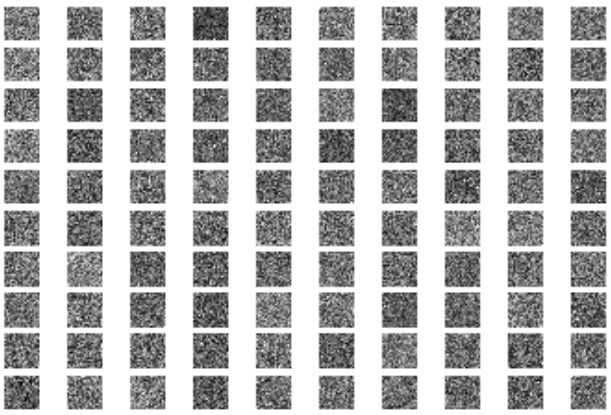


Figure 35: Learned Weights of Hidden Layer 2

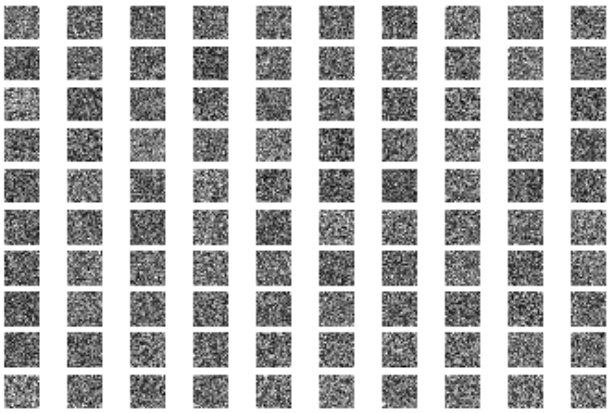
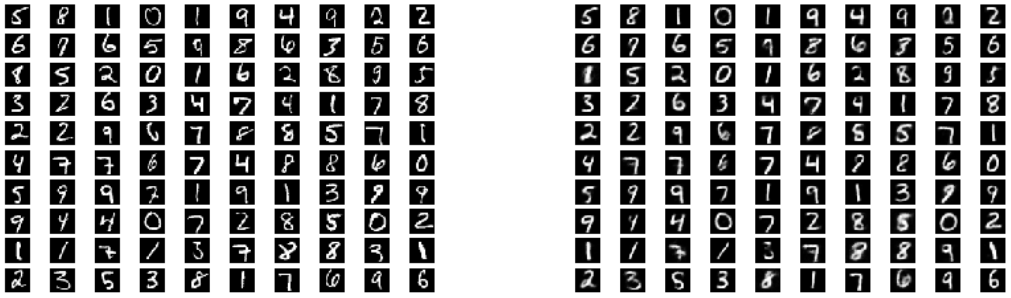


Figure 36: Learned Weights of Hidden Layer 3

We randomly pick 100 images from the test set and pass them to the stacked autoencoder. The original figure as well as the reconstructed on are shown as follow. It seems that the reconstructed images are a bit blue but carry enough information for digit recognition.



(a) Original Figure

(b) Reconstructed Figure

The flowing figures show the activation of each hidden layer after implementing sparsity constraint. As we can see that comparing with previous activation figures, this set of plots are relatively darker. This is because some neurons in the hidden layer are disabled due to the constraint applied on.

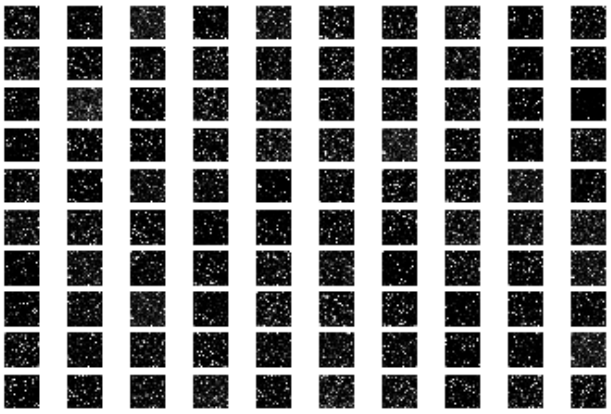


Figure 38: Activation of Hidden Layer 1

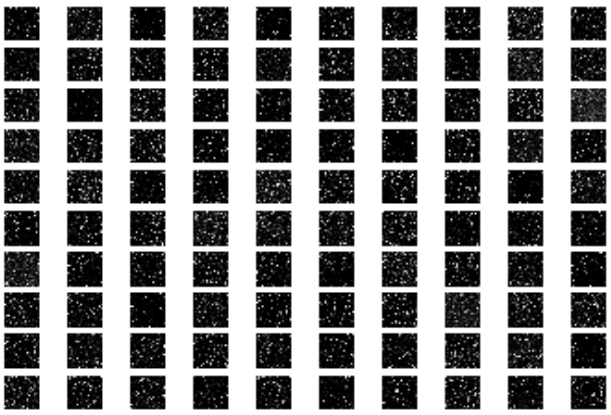


Figure 39: Activation of Hidden Layer 2

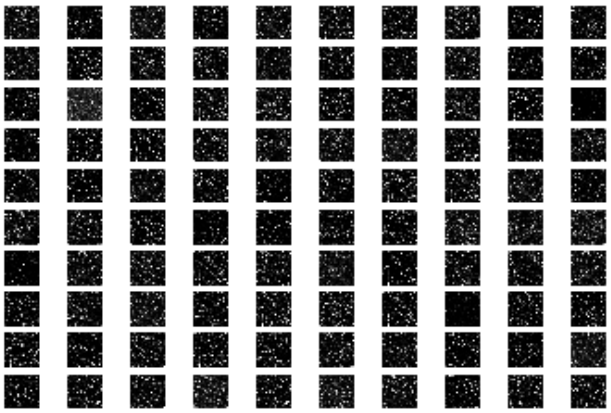


Figure 40: Activation of Hidden Layer 3

After getting the pretrained hidden layers, with similar implementation as in question 2, we form a five layer feedforward network by adding a softmax layer with 10 neurons at the end for classification purpose. Initialize the weights for the last layer and keep the others as what they are after pretrain step. During training, we back propagate to the very first layer and update each corresponding parameters. The following curves showing the training loss as well as test accuracy.

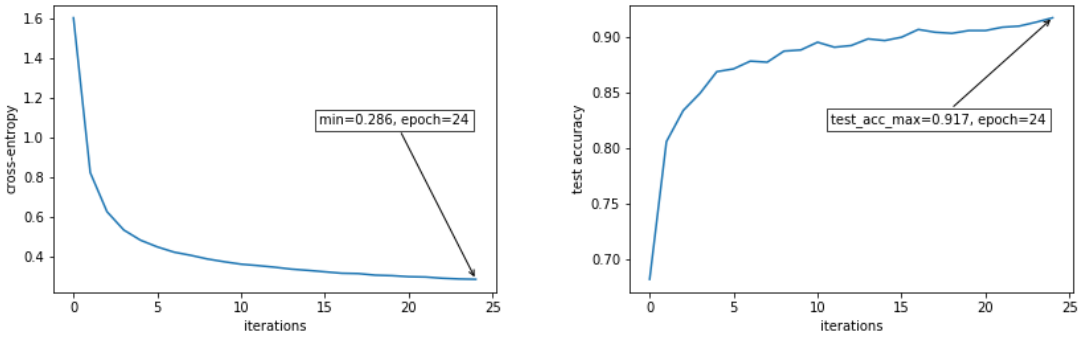



Figure 41: Training Cost and Test Accuracy for Q3

Comparing with the plots obtained from question 2, we realize that the curve for test accuracy is much smoother due to the implementation of gradient descent with momentum. Meanwhile, the training cost is relatively higher because we introduce sparsity constraint in this section which causes some neurons to be inactive. Since the training has not converged yet, the loss could be decreasing further if adding more iterations.