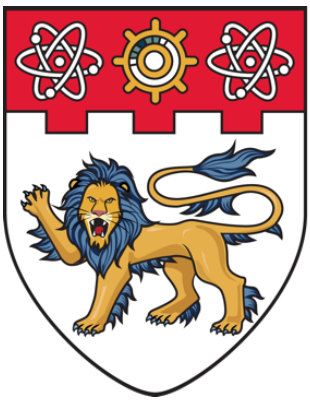


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ4042 NEURAL NETWORK PROJECT 2

SHAO YIYANG (U1420722G)
ZHAO QINGMEI (U1422368A)

School of Computer Science and Engineering

AY 17/18, semester 1

Contents

1 Part A: Deep convolutional neural network

In this part, I used convolutional neural networks (CNN) for object recognition in images, using MNIST dataset. I designed and implemented a convolutional neural network, and tested out differences between SGD, SGD with momentum and RMSprop algorithms for CNN.

1.1 Basic CNN network with SGD

In this section, I implemented a convolutional neural network with a input layer of 28 x 28 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 2 x 2, a convolution layer C1 of 20 feature maps and filters of window size 5 x 5. A max pooling layer S1 with a pooling window of size 2 x 2, a fully connected layer F3 of size 100 and finally a softmax layer F4 of size 10.

The architecture of the convolutional neural network is briefly shown in Figure 1.

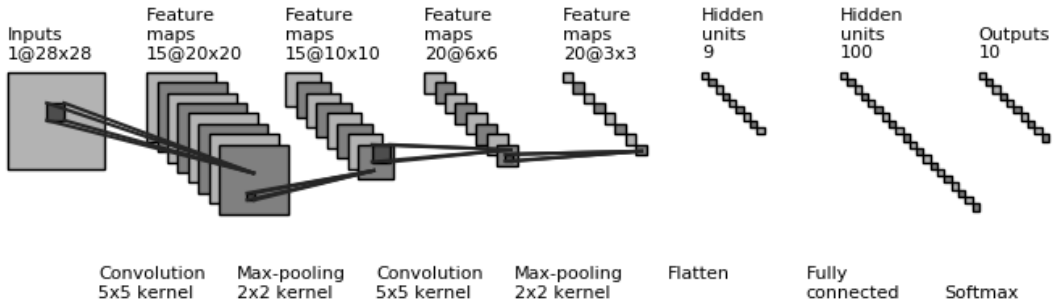


Figure 1: Convolutional Neural Network Architecture

1.1.1 Implementation

I implemented the CNN model using Theano. The model includes two convolutional layers, two pooling layers, a fully connect layer and a softmax layer of size 10, corresponding to the 10 digits classes desired. The detailed code is listed below.

```
def model(X, w1, b1, w2, b2, w3, b3, w4, b4):
        # First convolution and pooling layer
        y1 = T.nnet.relu(conv2d(X, w1) + b1.dimshuffle('x', 0, 'x', 'x'))
        pool_dim1 = (2, 2)
        o1 = pool.pool_2d(y1, pool_dim1, ignore_border=True)

        # Second convolution and pooling layer
        y2 = T.nnet.relu(conv2d(o1, w2) + b2.dimshuffle('x', 0, 'x', 'x'))
        pool_dim2 = (2, 2)
        o2 = pool.pool_2d(y2, pool_dim2, ignore_border=True)

        # FC layer
        fc = T.nnet.relu(T.dot(T.flatten(o2, outdim=2), w3) + b3)

```

```
        # Softmax
        pyx = T.nnet.softmax(T.dot(fc, w4) + b4)

        return y1, o1, y2, o2, pyx
```

In addition, a normal SGD is implemented and prepared for further comparison.

```
def sgd(cost, params, lr=0.05, decay=0.0001):
    grads = T.grad(cost=cost, wrt=params)
    updates = []
    for p, g in zip(params, grads):
        updates.append([p, p - (g + decay*p) * lr])
    return updates
```

The parameters for training and sizes for each layer are listed below.

```
batch_size = 128
learning_rate = 0.05
decay = 1e-4
epochs = 100

num_filters_1 = 15
num_filters_2 = 20
fc_size = 100
softmax_size = 10
```

1.1.2 Result

The plot of training cost and test accuracy against epochs are in Figure 2 and Figure 3. The training achieved the best accuracy 97.2% at iteration number 29.

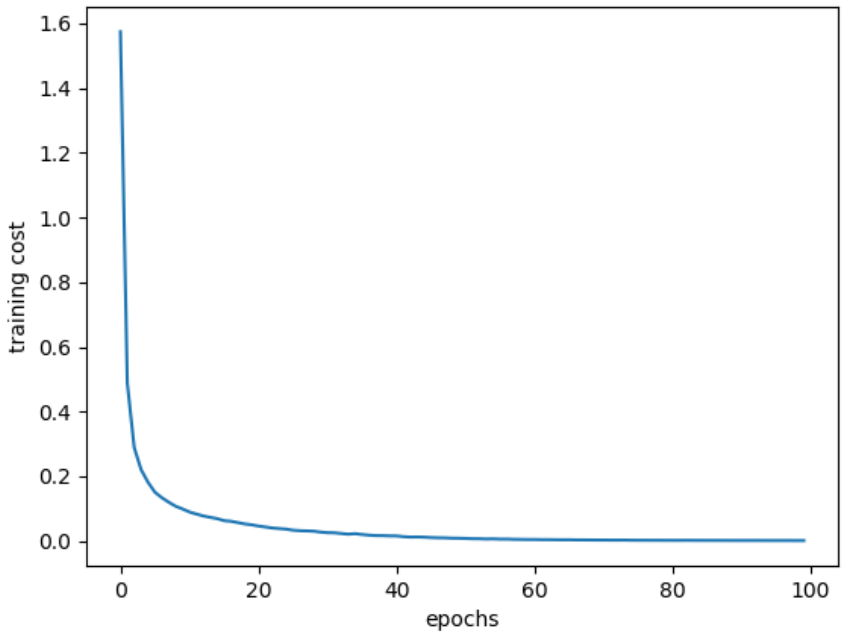


Figure 2: SGD Training Cost

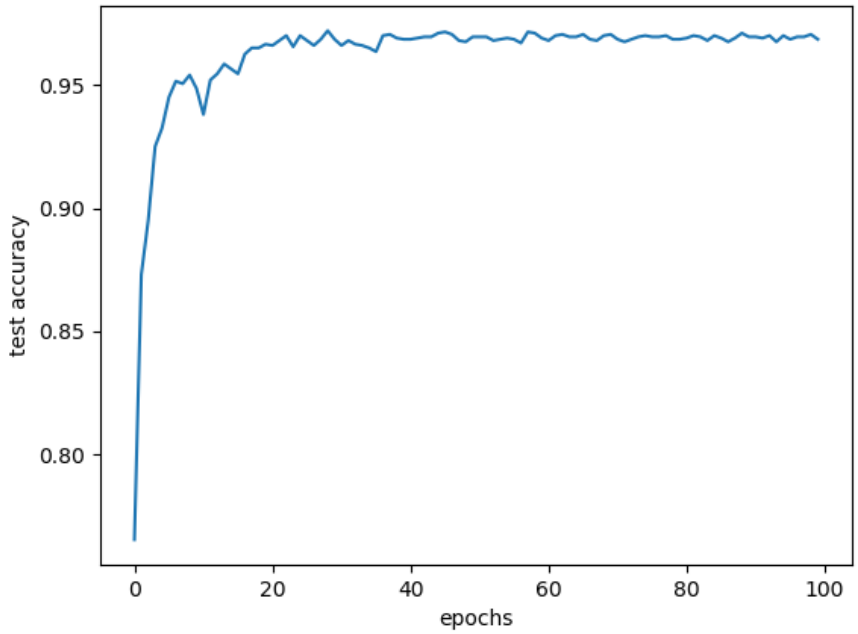
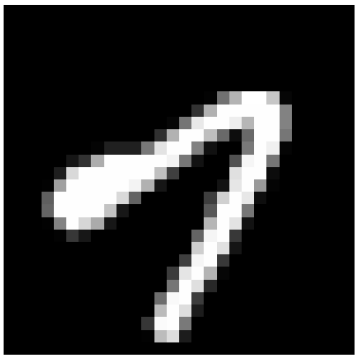
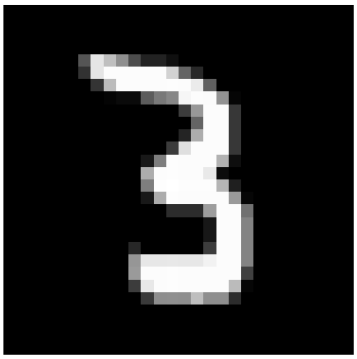


Figure 3: SGD Test Accuracy

The feature maps after each convolutional layer and pooling layer are also visualized.
Figure 4 shows the two representative test inputs and the input images were reshaped to a size of 28 by 28.
Figure 5 and Figure 7 show the feature maps extracted after the two convolutional layers.
Figure 6 and Figure 8 show the feature maps extracted after the two pooling layers.

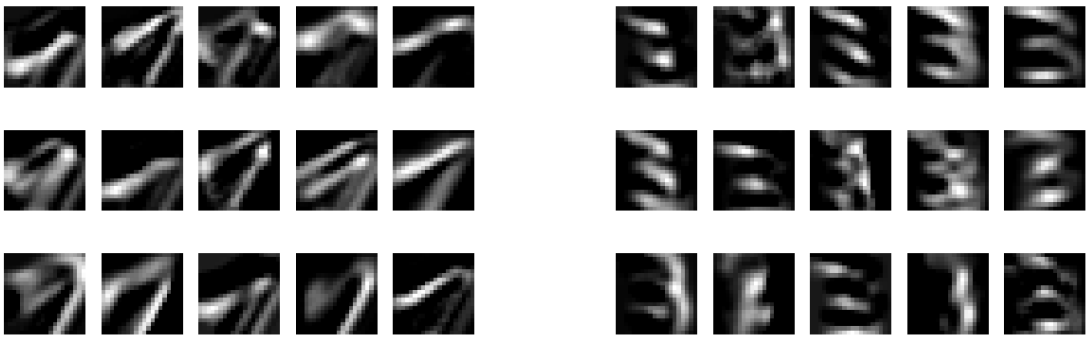


(a) Test Input 1



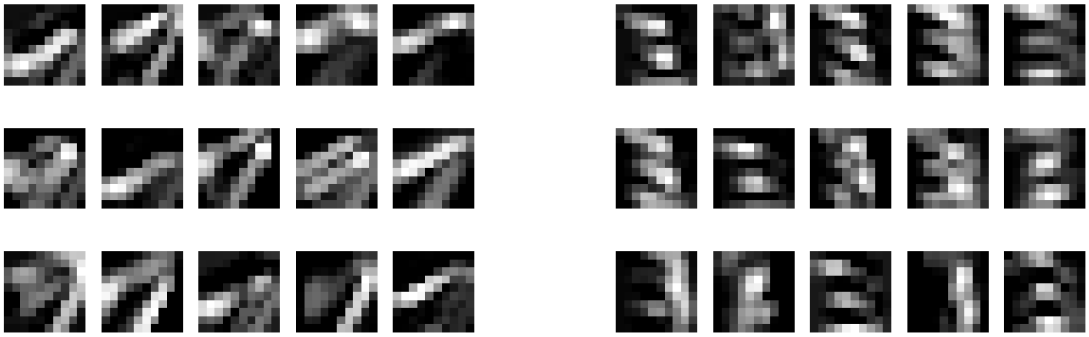
(b) Test Input 2

Figure 4: Test Inputs (28 x 28)



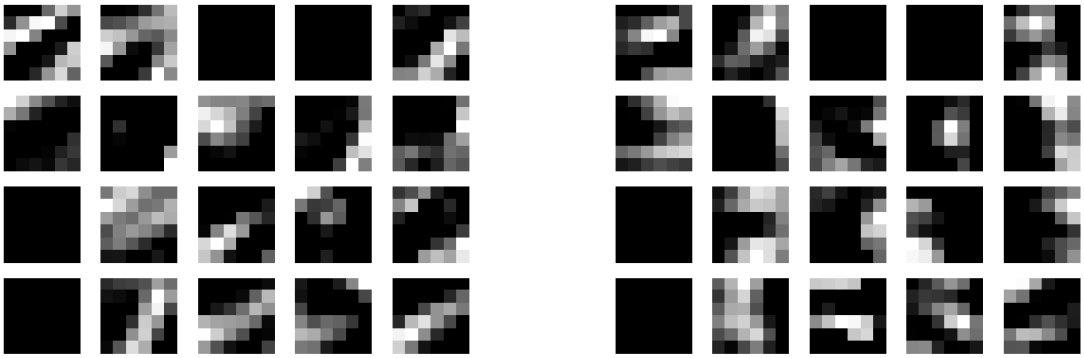
(a) Test Input 1 (b) Test Input 2

Figure 5: Feature Maps after SGD Convolutional Layer 1 (20 x 20)



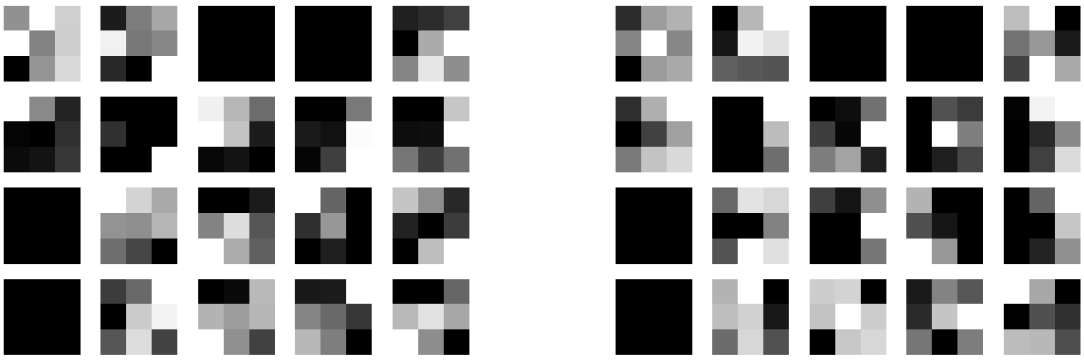
(a) Test Input 1 (b) Test Input 2

Figure 6: Feature Maps after SGD Pooling Layer 1 (10 x 10)



(a) Test Input1 (b) Test Input2

Figure 7: Feature Maps after SGD Convolutional Layer 2 (6 x 6)



(a) Test Input 1 (b) Test Input 2

Figure 8: Feature Maps after SGD Pooling Layer 2 (3 x 3)

1.2 SGD with momentum

1.2.1 Implementation

In this secion, I modified the normal SGD by adding the momentum term. The implementation is listed below.

```
def sgd_momentum(cost, params, lr=0.05, decay=0.0001, momentum=0.1):
        grads = T.grad(cost=cost, wrt=params)
        updates = []
        for p, g in zip(params, grads):
                v = theano.shared(p.get_value()*0.0)
                v_new = momentum*v - (g + decay*p) * lr
                updates.append([p, p + v_new])
                updates.append([v, v_new])
        return updates
```

The parameters for training and sizes for each layer are listed below.

```
batch_size = 128
learning_rate = 0.05
decay = 1e-4
epochs = 100
momentum = 0.1  # Provide with momentum

num_filters_1 = 15
num_filters_2 = 20
fc_size = 100
softmax_size = 10
```

1.2.2 Result

The plot of training cost and test accuracy against epochs are in Figure 9 and Figure 10. The training achieved the best accuracy 97.25% at iteration number 49.

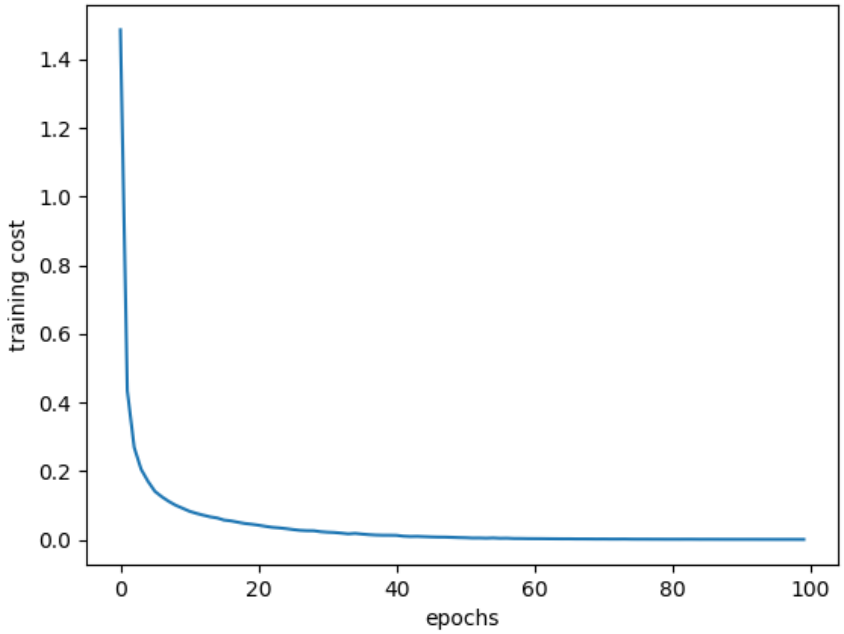


Figure 9: SGD with Momentum Training Cost

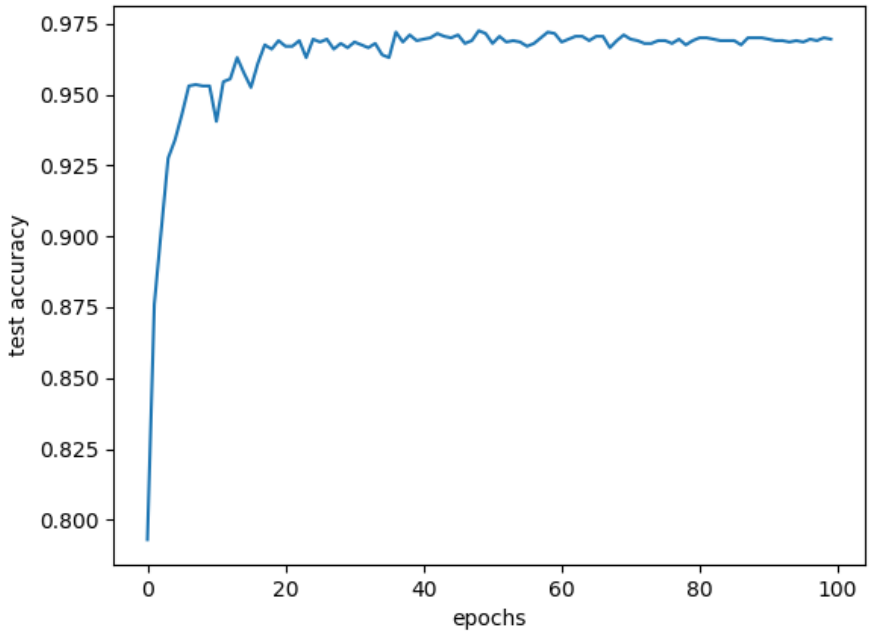
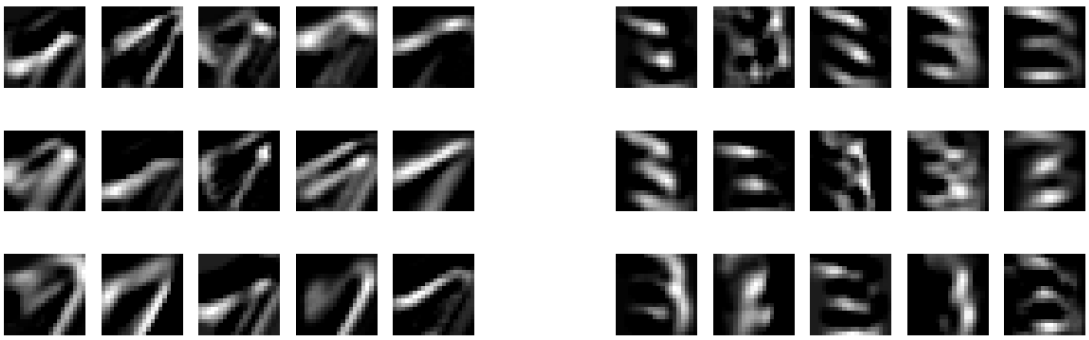


Figure 10: SGD with Momentum Test Accuracy

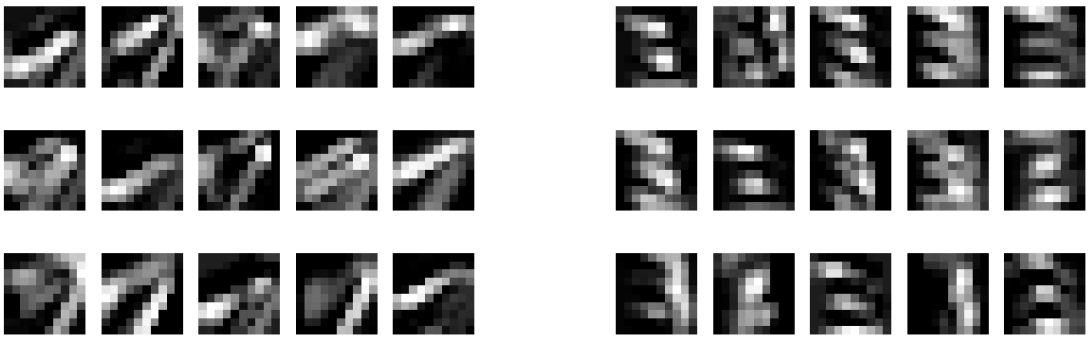
The feature maps after each convolutional layer and pooling layer are also visualized.
Figure 4 shows the two representative test inputs and the input images were reshaped to a size of 28 by 28.
Figure 11 and Figure 13 show the feature maps extracted after the two convolutional layers.
Figure 12 and Figure 14 show the feature maps extracted after the two pooling layers.



(a) Test Input 1

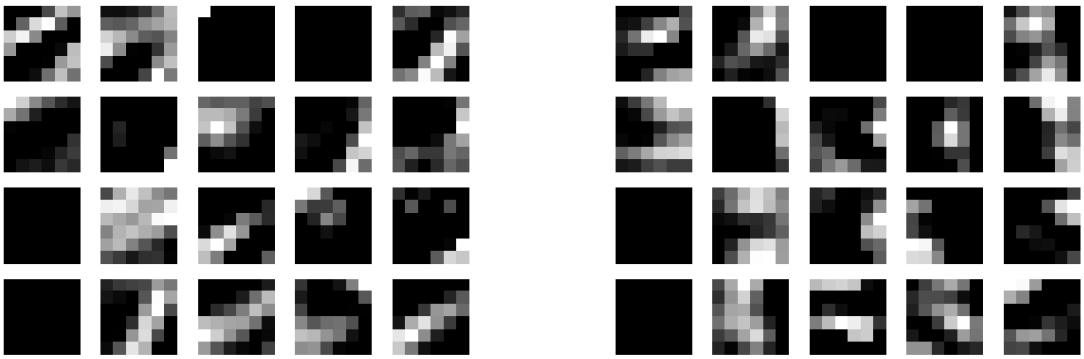
(b) Test Input 2

Figure 11: Feature Maps after SGD with Momentum Convolutional Layer 1 (20 x 20)



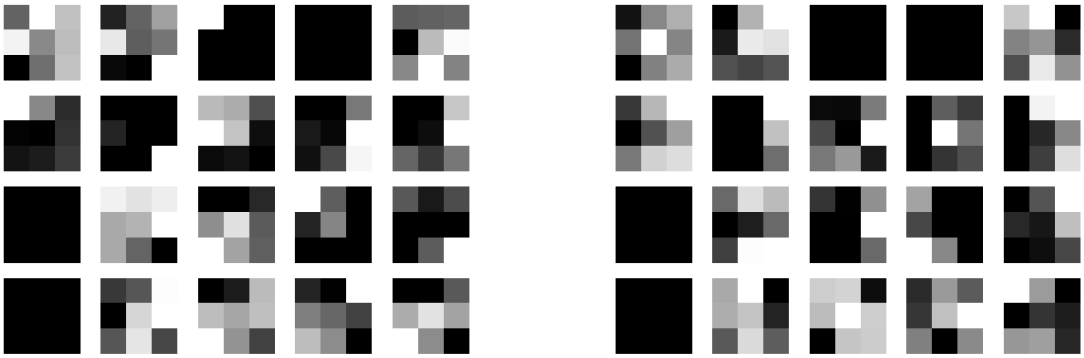
(a) Test Input 1 (b) Test Input 2

Figure 12: Feature Maps after SGD with Momentum Pooling Layer 1 (10 x 10)



(a) Test Input1 (b) Test Input2

Figure 13: Feature Maps after SGD with Momentum Convolutional Layer 2 (6 x 6)



(a) Test Input 1 (b) Test Input 2

Figure 14: Feature Maps after SGD with Momentum Pooling Layer 2 (3 x 3)

1.3 RMSProp

1.3.1 Implementation

In this secion, I used RMSProp algorithm. The implementation is listed below.

```
def RMSprop(cost, params, lr=0.001, decay=0.0001, rho=0.9, eps=1e-6):
        grads = T.grad(cost=cost, wrt=params)
        updates = []
        for p, g in zip(params, grads):
                r = theano.shared(p.get_value() * 0.0)
                r_new = rho * r + (1 - rho) * (g ** 2)
                updates.append([p, p - lr * (g / T.sqrt(r_new + eps)+ decay*p)])
                updates.append([r, r_new])
        return updates
```

And the parameters are adjusted for RMSProp. A smaller learning rate is used.

```
batch_size = 128
learning_rate = 0.001
decay = 1e-4
epochs = 100

# rho and epsilon are added for RMSProp
rho=0.9
eps=1e-6

num_filters_1 = 15
num_filters_2 = 20
fc_size = 100
softmax_size = 10
```

1.3.2 Result

The plot of training cost and test accuracy against epochs are in Figure 15 and Figure 16. The training achieved the best accuracy 97.75% at iteration number 39.

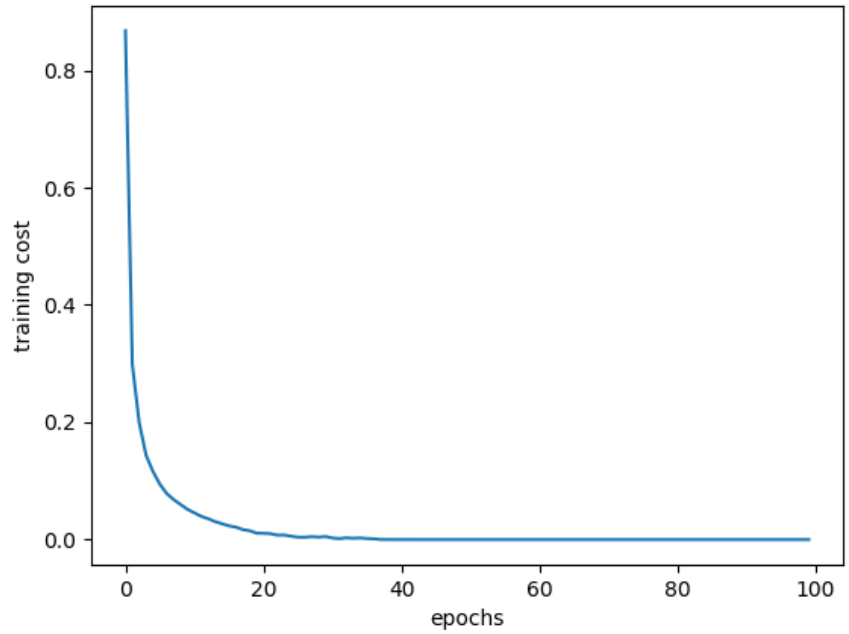


Figure 15: RMSProp Training Cost

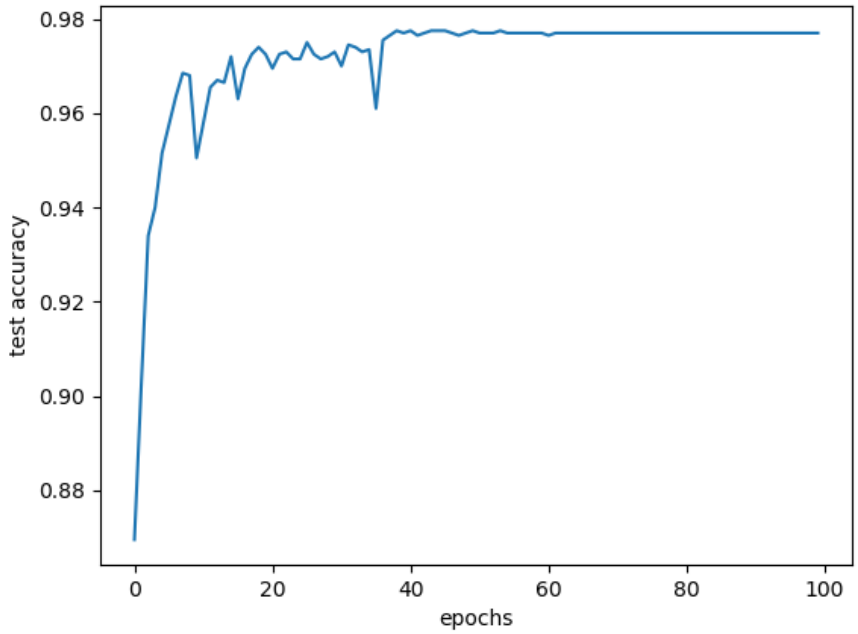
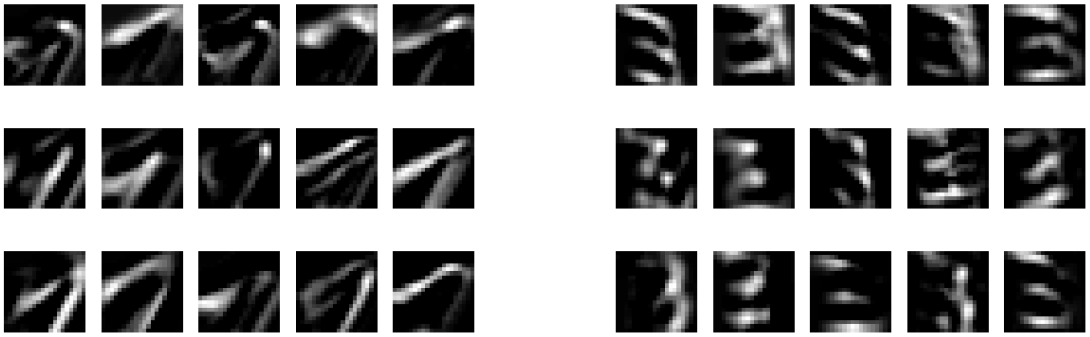


Figure 16: RMSProp Test Accuracy

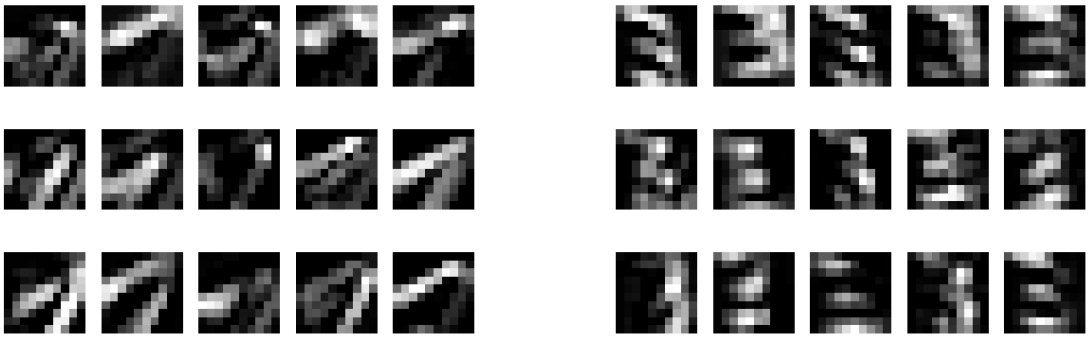
The feature maps after each convolutional layer and pooling layer are also visualized.
Figure 4 shows the two representative test inputs and the input images were reshaped to a size of 28 by 28.

Figure 17 and Figure 19 show the feature maps extracted after the two convolutional layers. Figure 18 and Figure 20 show the feature maps extracted after the two pooling layers.



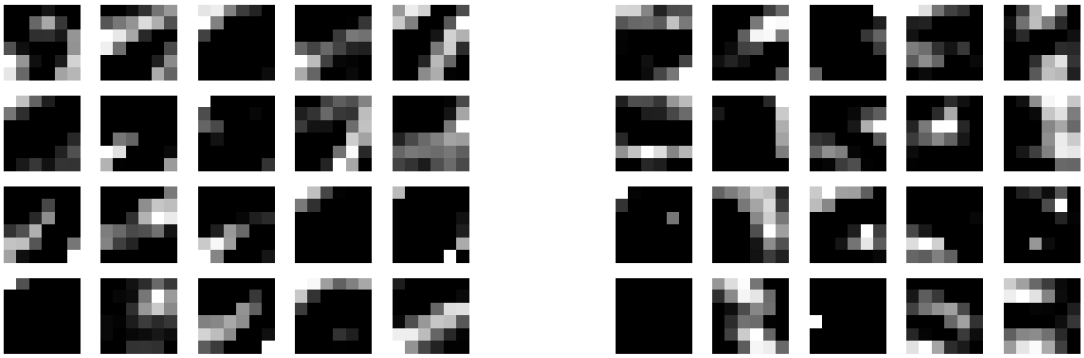
(a) Test Input 1 (b) Test Input 2

Figure 17: Feature Maps after RMSProp Convolutional Layer 1 (20 x 20)



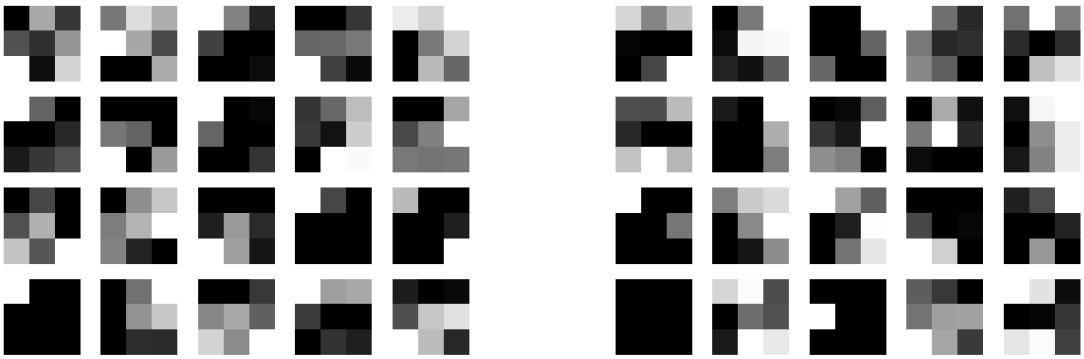
(a) Test Input 1 (b) Test Input 2

Figure 18: Feature Maps after RMSProp Pooling Layer 1 (10 x 10)



(a) Test Input1 (b) Test Input2

Figure 19: Feature Maps after RMSProp Convolutional Layer 2 (6 x 6)



(a) Test Input 1 (b) Test Input 2

Figure 20: Feature Maps after RMSProp Pooling Layer 2 (3 x 3)

1.4 Discussion for CNN

Overall, after training 100 epoches, three schemes all could achieve an accuracy around 97%. From the visualization of feature maps of different stages of the network. As seen from the graphs above, higher level of features could be extracted after each convolutional layer and pooling layer.

In addition, different gradient descent optimization algorithms have obvious effects on the training and convergence of the neural network. Original SGD tends to have unstable update steps per iteration and convergence takes relatively more time. Sometimes it will make the network stuck in a poor local minima. However, adding momentum could reduce the wildness of original SGD. Momentum is able to keep a record of the previous update steps and utilize this information for the next gradient step to keep the updates more stable, therefore the test accuracy graph looks more smooth when the model is close to convergence point. RMSProp uses an exponentially decaying average to discard the history from extreme past so that it can

converge rapidly after finding a convex region. It keeps running average of its recent gradient magnitudes and divides the next gradient by this average so that loosely gradient values are normalized.

2 Part B: Autoencoders

2.1 3-Layer Stacked Denoising Autoencoder

2.1.1 Implementation

The denoising auto-encoder receives corrupted data as inputs and its trained to predict the original uncorrupted data as its output. For this question, we corrupt the input data using a binomial distribution at 10% corruption level, achieved by applying following code:

```
tilde_x = theano_rng.binomial(size=x.shape, n=1, p=1 - corruption_level, dtype=
    theano.config.floatX)*x
```

The each layer in Stacked auto-encoder should be trained separately. Hence, the stacked denoising autoe-ncoder is constructed by following steps:

Firstly, apply binomial distribution at 10% corruption level on raw input x to obtain corrupted data ˜x. Use the ˜x as input to train the 1st denoising auto-encoder and learn primary features y2. The cost is the calculated based on the cross-entropy between the original data X and decoder output of y1 which is z1.

Secondly, we should obtrain the primary feature activations y1 as the raw input. Then, apply binomial distribution at 10% corruption level on raw input y1 to obtain corrupted data ˜y1. Use the ˜y1 as input to train the 2nd denoising auto-encoder and learn primary features y2. The cost is the calculated based on the cross-entropy between the original data y1 and decoder output of y2 which is z2.

Secondly, we should obtrain the primary feature activations y2 as the raw input. Then, apply binomial distribution at 10% corruption level on raw input y2 to obtain corrupted data ˜y2. Use the ˜y2 as input to train the 3rd denoising auto-encoder and learn primary features y3. The cost is the calculated based on the cross-entropy between the original data y2 and decoder output of y3 which is z4.

Lastly, since we already have trained the 3 layers y1, y2 and y3. The 3-Layer Stacked Denoising Autoencoder can be constructed by raw original input layer x, ˜x, y1, ˜y1, y2, ˜y2, y3, z3, z2, z1, as shown in the Figure 21

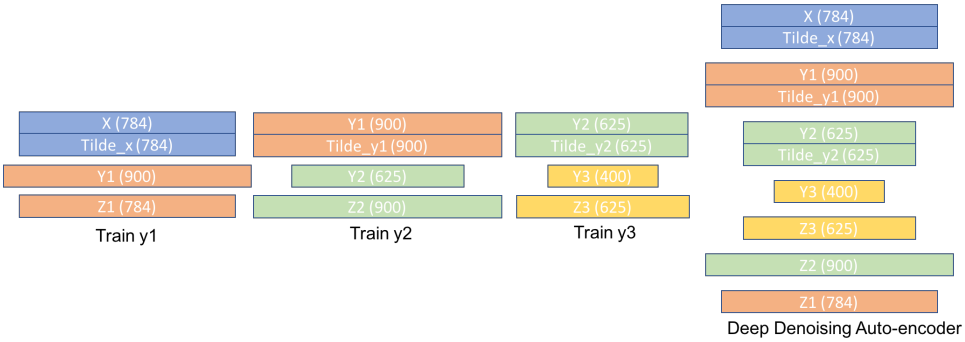


Figure 21: 3-Layer Stacked Denoising Autoencoder

2.1.2 Learning Curves for Training Each Layer

Figure 22 is the training error (cross-entropy) for training the 1st layer. Figure 23 is the training error (cross-entropy) for training the 2nd layer. Figure 24 is the training error (cross-entropy) for training the 3rd layer. By comparing the figures, they basically have the similar curve about how the training error is been decreasing. However, the ranges of the curves are obviously different. The first layer with 900 neurons has the training error ranged from 60 to 100. The second layer with 625 neurons has the training error ranged from 410 to 480. The third layer with 400 neurons has the training error ranged from 180 to 215.

The first layer has the least training error, as shown in Figure 22. The training of first layer is a training process of an over-complete auto-encoder. The input dimension is 28*28 = 784 and the hidden-layer has the dimension 900, as shown in the Figure 21 Training y1. The hidden-layer has a higher dimension than the input. Hence, it is an **over-complete auto-encoder**.

The second layer and third layer has the larger training errors, as shown in Figure 23 and Figure 24. The training of second and third layer is a training process of an under-complete auto-encoder. The hidden-layer has a lower dimension than the input as shown in Figure 21 Training y2 and Training y3. Hence, They are **under-complete auto-encoders**.

As the hidden layer of under-complete auto-encoder compresses the input. Hidden units will be good features for training distribution. We guess that it is more difficult to train the under-complete auto-encoder model than over-complete auto-encoder model, as the under-complete auto-encoder needs to catch the interesting structures, but there is no guarantee that over-complete auto-encoder can extract meaningful structure. Hence, the training errors of training the second layer and the third layer are larger than the training errors of training the first layer.

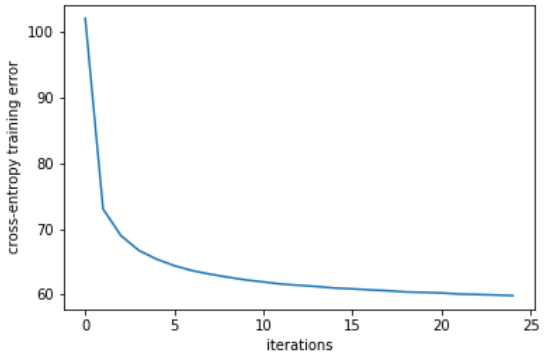


Figure 22: Training Error Against Epoch for Training 1st Layer

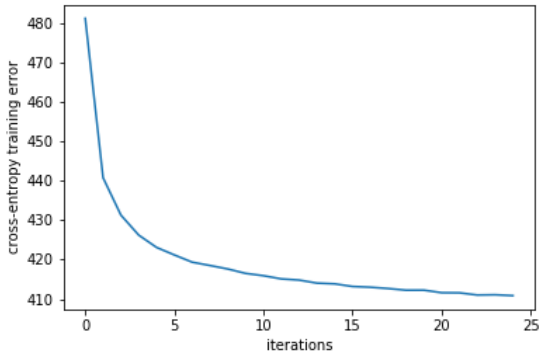


Figure 23: Training Error Against Epoch for Training 2rd Layer

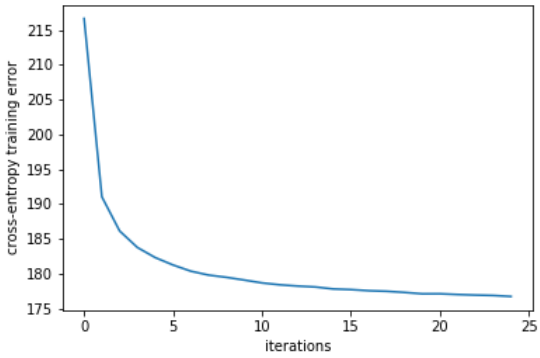


Figure 24: Training Error Against Epoch for Training 3rd Layer

2.1.3 100 Samples of Weights (as images) Learned at Each Layer

Some obvious Light dots and black dots can be spotted in the weight images. We believe that the dots are representing the important features which can differentiate the input images.

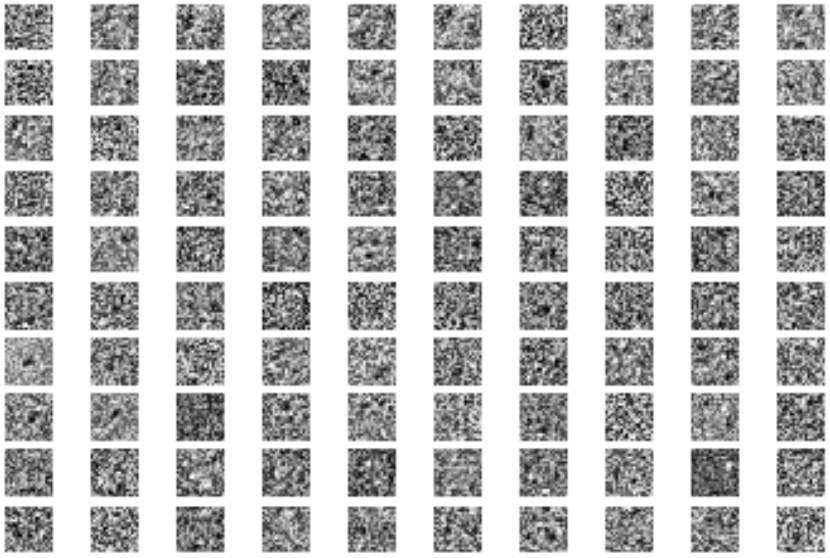


Figure 25: 100 Samples of Weights Learned at 1st Layer

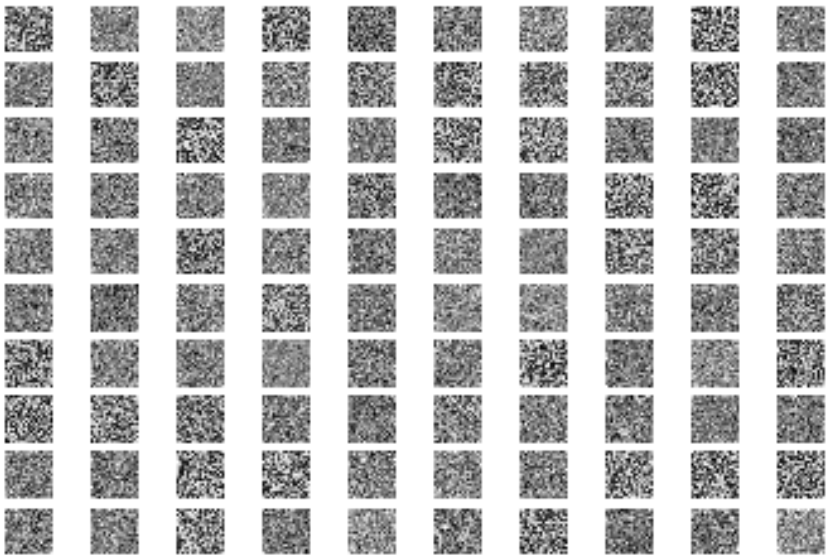


Figure 26: 100 Samples of Weights Learned at 2nd Layer

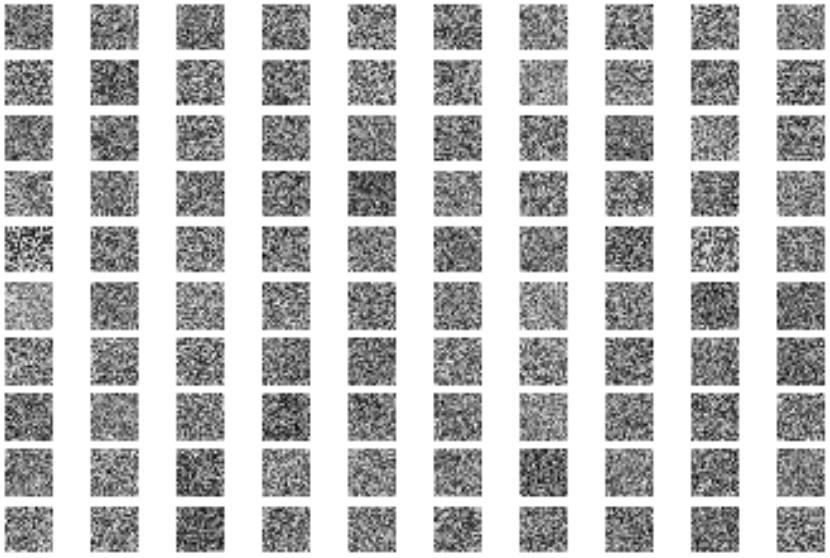


Figure 27: 100 Samples of Weights Learned at 3rd Layer

2.1.4 100 Representative Test Images

The following section shows the 100 Representative Test Images (Figure 28), The 100 Reconstructed images from 100 Representative Test Images (Figure 29), and the Hidden Layer Activation on the 3 layers (Figure 30, 31, 32). The Hidden Layer Activation is actually the encoded image of the original image.

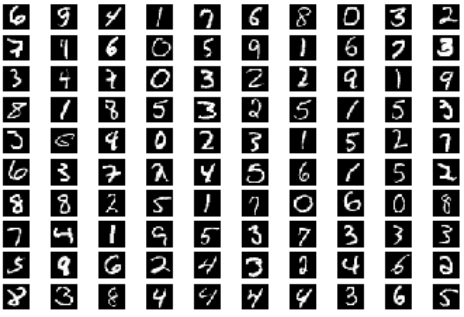


Figure 28: 100 Representative Test Images

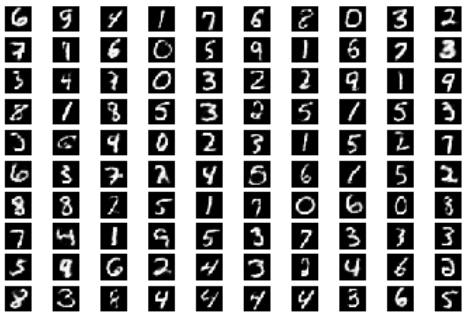


Figure 29: The 100 Reconstructed images from 100 Representative Test Images

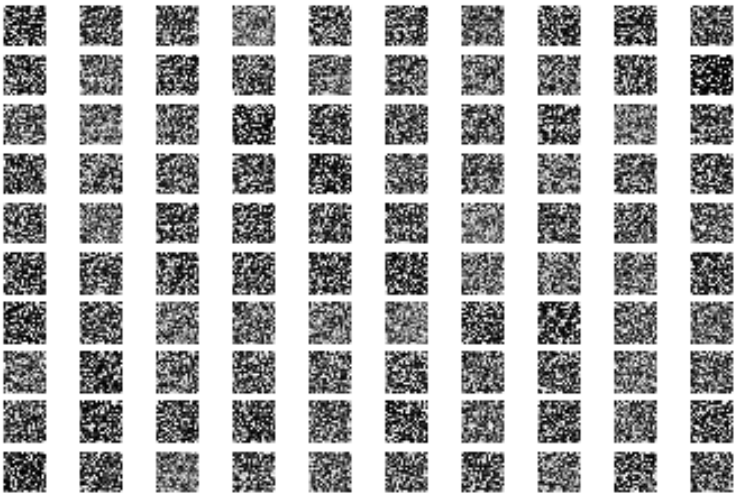


Figure 30: Hidden Layer Activation from 100 Representative Test Images on 1st Layer

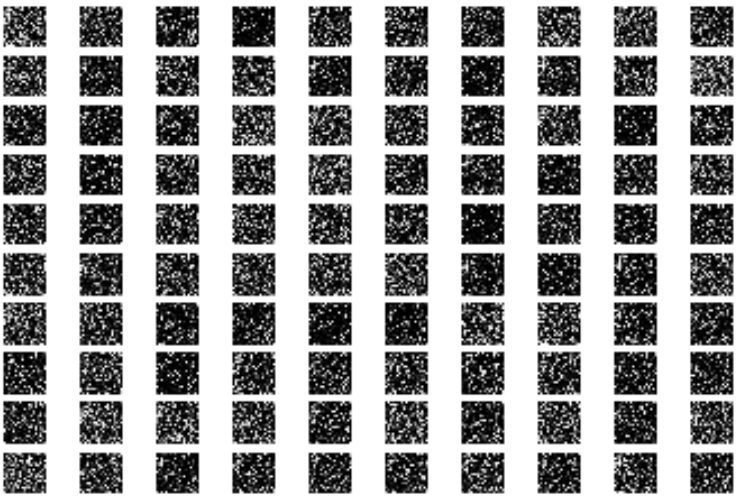


Figure 31: Hidden Layer Activation from 100 Representative Test Images on 2nd Layer

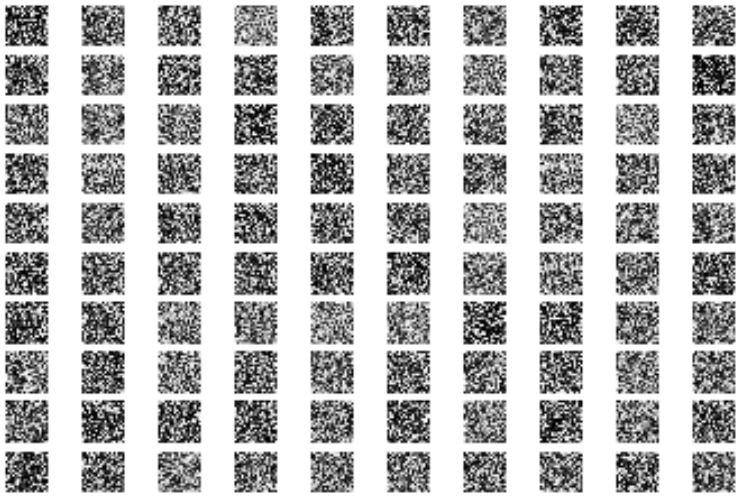


Figure 32: Hidden Layer Activation from 100 Representative Test Images on 3rd Layer

2.1.5 Code for Building 3-Layer Stacked Denoising Autoencoder

```
#  train on the inputs tilde_x to learn primary features y1
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)
cost_da1 = - T.mean(T.sum(x * T.log(z1) + (1 - x) * T.log(1 - z1), axis=1))
params_da1 = [W1, b1, b1_prime]
grads_da1 = T.grad(cost_da1, params_da1)
updates_da1 = [(param_da, param_da - learning_rate * grad_da)
           for param_da, grad_da in zip(params_da1, grads_da1)]
train_da1 = theano.function(inputs=[x], outputs = cost_da1, updates =
    updates_da1, allow_input_downcast = True)

#  train on the inputs y1 to learn features y2
tilde_y1 = theano_rng.binomial(size=y1.shape, n=1, p=1 - corruption_level,
                       dtype=theano.config.floatX)*y1
y2 = T.nnet.sigmoid(T.dot(tilde_y1, W2) + b2)
z2 = T.nnet.sigmoid(T.dot(y2, W2_prime) + b2_prime)
cost_da2 = - T.mean(T.sum(y1 * T.log(z2) + (1 - y1) * T.log(1 - z2), axis=1))
params_da2 = [W2, b2, b2_prime]
grads_da2 = T.grad(cost_da2, params_da2)
updates_da2 = [(param_da, param_da - learning_rate * grad_da)
           for param_da, grad_da in zip(params_da2, grads_da2)]
train_da2 = theano.function(inputs=[x], outputs = cost_da2, updates =
    updates_da2, allow_input_downcast = True)

#  train on the inputs y2 to learn features y3
tilde_y2 = theano_rng.binomial(size=y2.shape, n=1, p=1 - corruption_level,
                       dtype=theano.config.floatX)*y2
```

```
y3 = T.nnet.sigmoid(T.dot(tilde_y2, W3) + b3)
z3 = T.nnet.sigmoid(T.dot(y3, W3_prime) + b3_prime)
cost_da3 = - T.mean(T.sum(y2 * T.log(z3) + (1 - y2) * T.log(1 - z3), axis=1))
params_da3 = [W3, b3, b3_prime]
grads_da3 = T.grad(cost_da3, params_da3)
updates_da3 = [(param_da, param_da - learning_rate * grad_da)
           for param_da, grad_da in zip(params_da3, grads_da3)]
train_da3 = theano.function(inputs=[x], outputs = cost_da3, updates =
    updates_da3, allow_input_downcast = True)

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)
    # 625
decoder2 = theano.function(inputs=[y2],outputs = z2, allow_input_downcast=True)
    # 900
decoder1 = theano.function(inputs=[y1], outputs = z1, allow_input_downcast=True)
     # 784
```

2.2 Five-layer Feed-forward Neural Network

2.2.1 Implementation

Five-layer Feed-forward Neural Network initialized by the three hidden layers learned in the previous section and adding a softmax layer as the output layer. Categorical_crossentropy is used as the cost function. The parameters of the whole system are fine tuned to minimize the error in predicting the supervised target by supervised gradient descent learning. The struction of the Five-layer Feed-forward Neural Network is shown as the Figure 33

The input of the Five-layer Feed-forward Neural Network is the 28*28 pixels images of handwriting number, and the output of the model is the prediction about what the number is in the images.

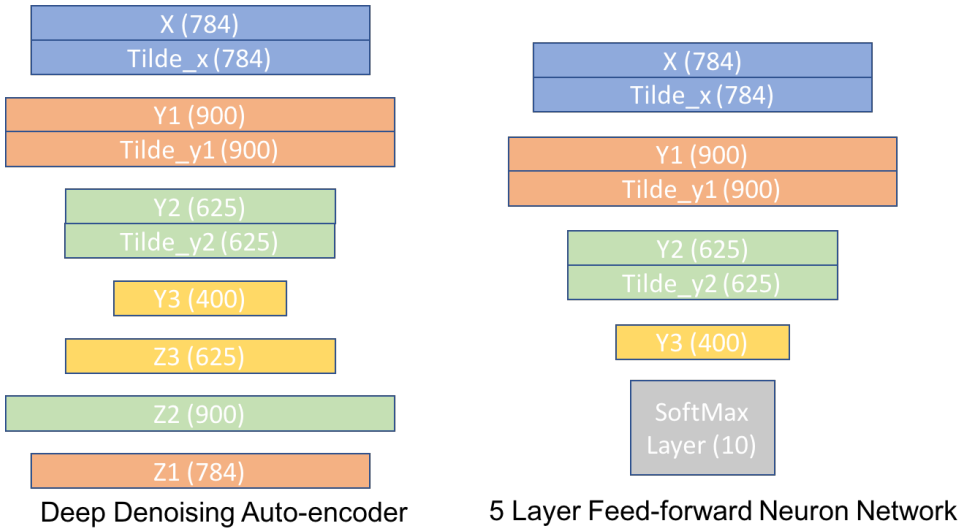


Figure 33: Structure Comparison between 3 Layer Auto-encoder and 5-Layer Feedforward Neural Network

2.2.2 Training Errors and Test Accuracies During Training

The model has been trained for 25 epochs. After 25 epochs, it can achieve the accuracy about 0.90 as shown in the Figure 35. The training error (Categorical_crossentropy) is decreased from 1.1 at epoch 0 to 0.2 at epoch 24 as shown in the Figure 34.

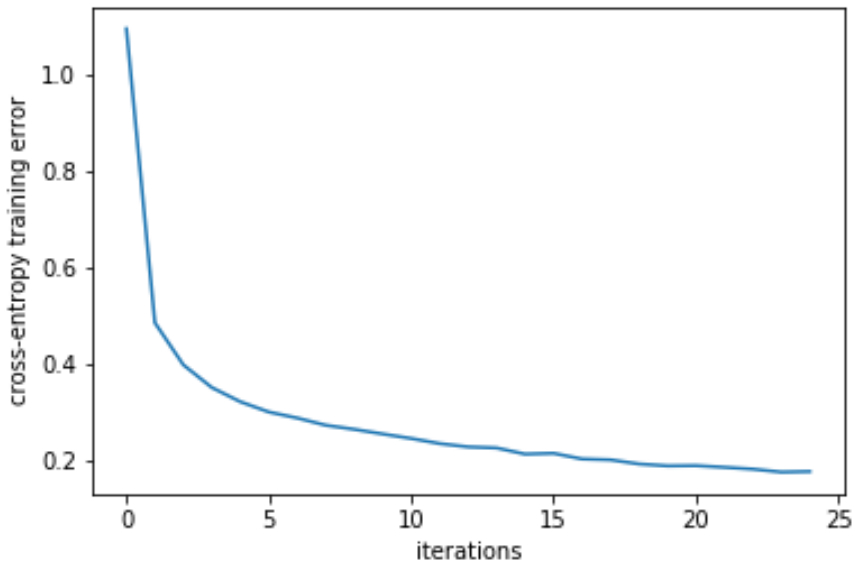


Figure 34: Training Errors During Training 5-Layer Feedforward Neural Network

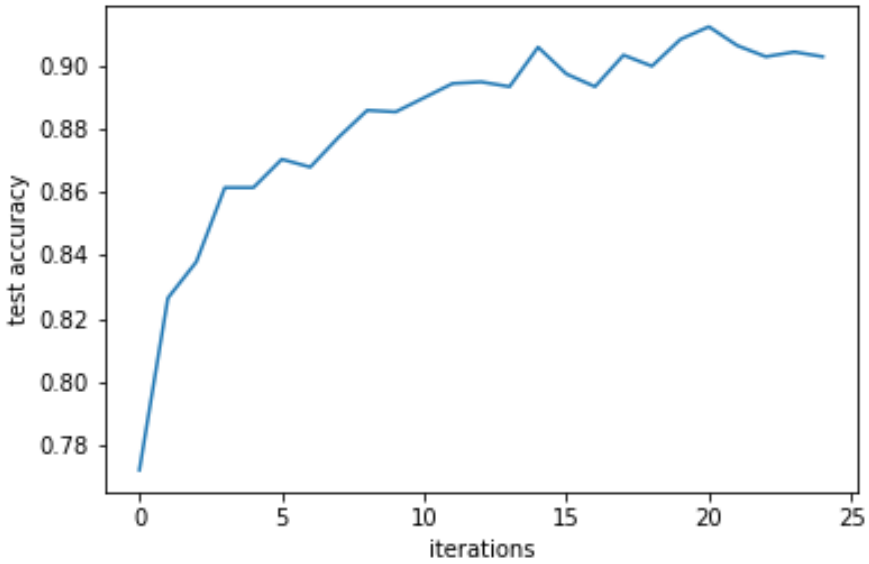


Figure 35: Test Accuracies During Training 5-Layer Feedforward Neural Network

2.2.3 Code for Building 5-Layer Feedforward Neural Network

```
# five-layer feedforward neuron network
output_ff = T.nnet.softmax(T.dot(y3, W4)+b4)
predicted_result_ff = T.argmax(output_ff, axis=1)
cost_ff = T.mean(T.nnet.categorical_crossentropy(output_ff, d))

params_ff = [W1, b1, W2, b2, W3, b3, W4, b4]
grads_ff = T.grad(cost_ff, params_ff)
updates_ff = [(param_ff, param_ff - learning_rate * grad_ff)
           for param_ff, grad_ff in zip(params_ff, grads_ff)]
train_ffn = theano.function(inputs=[x, d], outputs = cost_ff, updates =
    updates_ff, allow_input_downcast = True)
test_ffn = theano.function(inputs=[x], outputs = predicted_result_ff,
    allow_input_downcast=True)
```

2.3 Repeated the Previous 2 Parts by Introducing the Momentum Term for Gradient Descent Learning and the Sparsity Constraint to the Cost Function

2.3.1 Implementation

To implement the momentum term for Gradient Descent, the velocity needs to be calculated which is $v\_new = momentum * v - g * lr$. The code for implementing the momentum term for Gradient Descent is shown as below.

```
def sgd_momentum(cost, params, lr, momentum):
    grads = T.grad(cost=cost, wrt=params)
    updates = []
    for p, g in zip(params, grads):
        v = theano.shared(p.get_value())
```

```
#          v_new = momentum*v - (g + decay*p) * lr
         v_new = momentum*v - g * lr
         updates.append([p, p + v_new])
         updates.append([v, v_new])
         return updates
```

By theory, the Sparsity Constraint can help over-complete auto-encoder to explore interesting structures. It is implemented by adding penalty function to the cost function.

```
cost_da1 = - 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))
```

2.3.2 Learning Curves for Training Each Layer in 3-Layer Auto-encoder

Figure 36 is the training error (cross-entropy) for training the 1st layer. Figure 37 is the training error (cross-entropy) for training the 2nd layer. Figure 38 is the training error (cross-entropy) for training the 3rd layer. The shape of the training error curve is basically same as the previous result. However, the value of training error is different from the previous section. The first layer with 900 neurons has the training error ranged from 60 to 100, which is same as the previous result (Figure 22). The second layer with 625 neurons has the training error ranged from 350 to 410, which is significantly lower than the previous result 410 to 480 (Figure 23). The third layer with 400 neurons has the training error ranged from 195 to 235, which is also different from the previous result 180 to 215 (Figure 24).

According to the theory, the sparsity constraint can help over-complete auto-encoder to explore interesting structures. However, for our case, the training of first layer y1 is an over-complete auto-encoder. The value of training error with sparsity constraint almost has no different from the previous result. We believe that, after more epochs, it may help the over-complete auto-encoder to explore more interesting structures.

According the training error curve for training the second layer y2, the training error decreased from 410 - 480 to 350-410 with sparsity constraint and momentum SGD. It is because that the sparsity constraint can help auto-encoder to explore interesting structure so that the training error decreased.

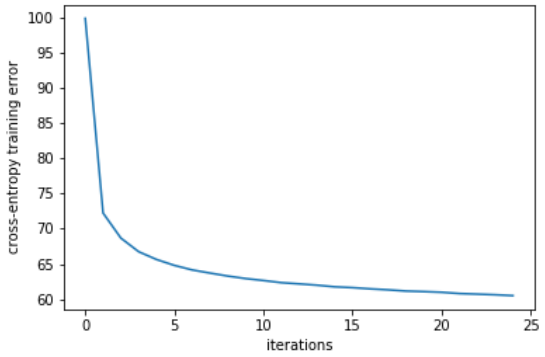


Figure 36: Training Error Against Epoch for Training 1st Layer

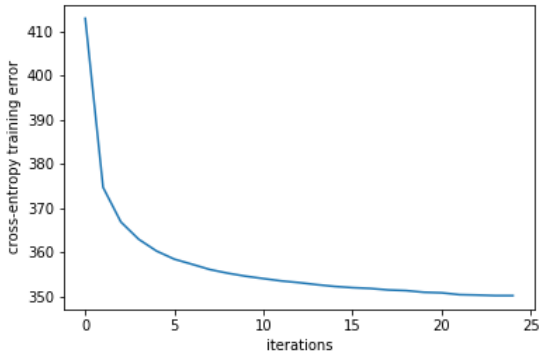


Figure 37: Training Error Against Epoch for Training 2rd Layer

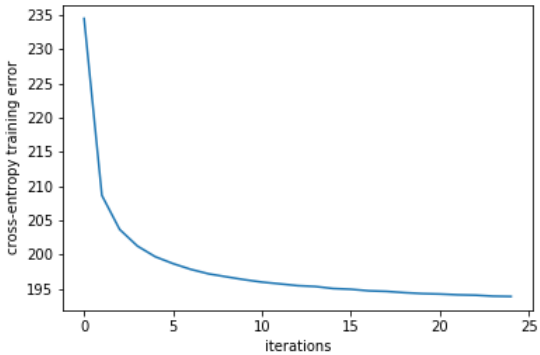


Figure 38: Training Error Against Epoch for Training 3rd Layer

2.3.3 100 Samples of Weights (as images) Learned at Each Layer in 3-Layer Auto-encoder

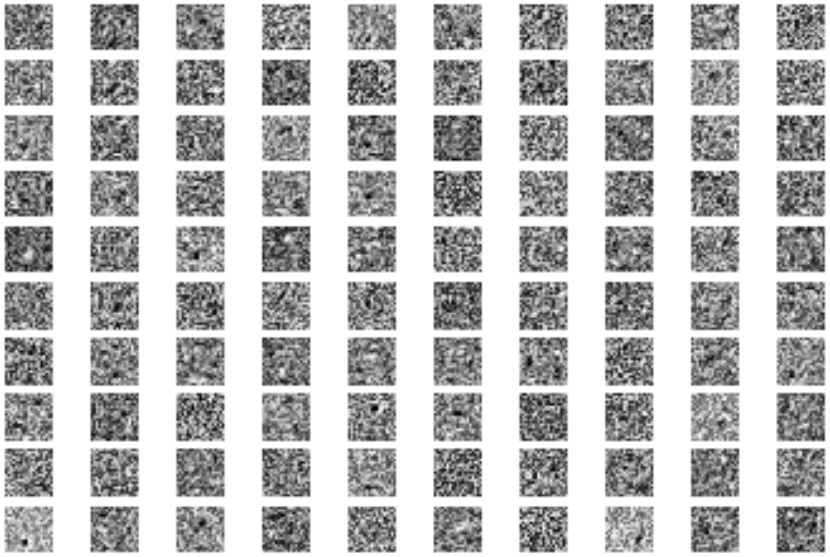


Figure 39: 100 Samples of Weights Learned at 1st Layer

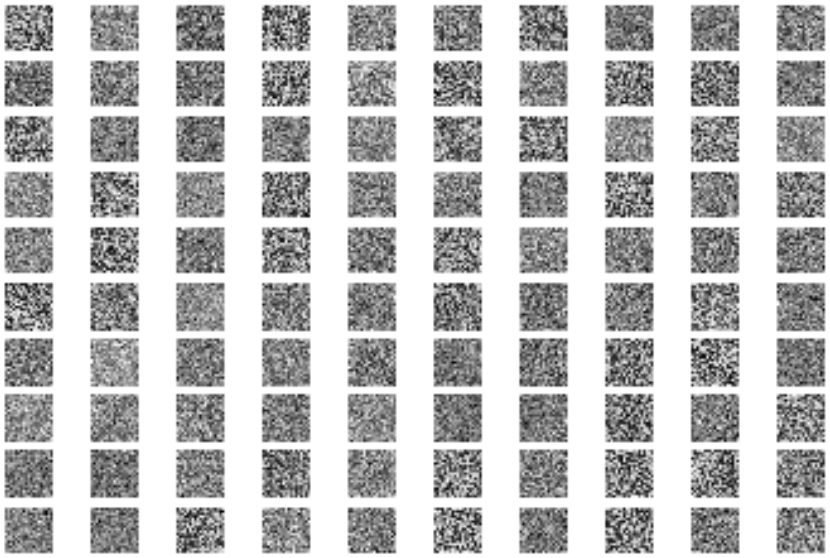


Figure 40: 100 Samples of Weights Learned at 2nd Layer

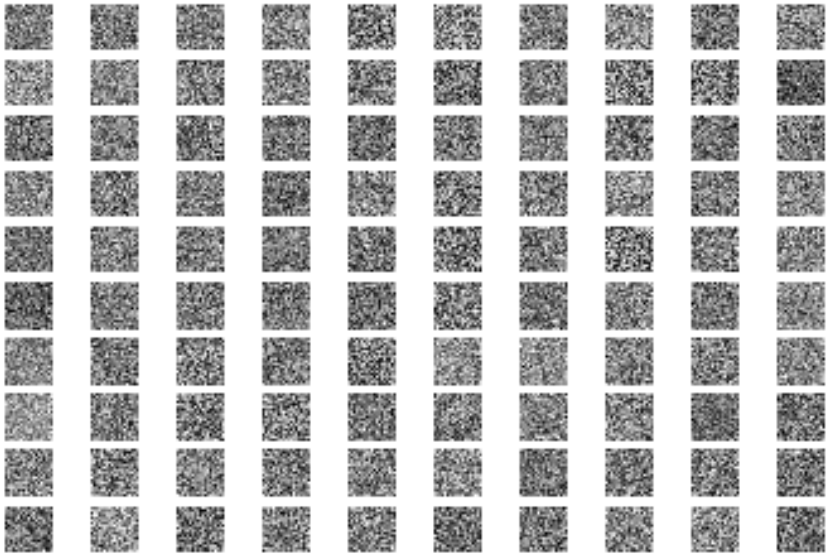


Figure 41: 100 Samples of Weights Learned at 3rd Layer

2.3.4 100 Representative Test Images in 3-Layer Auto-encoder

The following section shows the 100 Representative Test Images (Figure 42), The 100 Reconstructed images from 100 Representative Test Images (Figure 43), and the Hidden Layer Activation on the 3 layers (Figure 44, 45, 46). The Hidden Layer Activation is actually the encoded image of the original image.

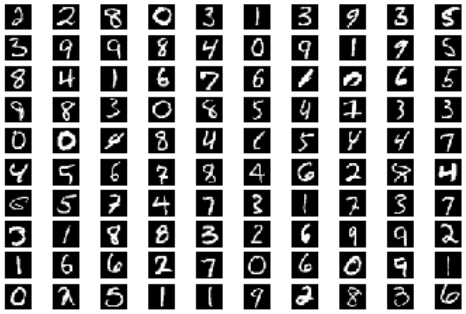


Figure 42: 100 Representative Test Images

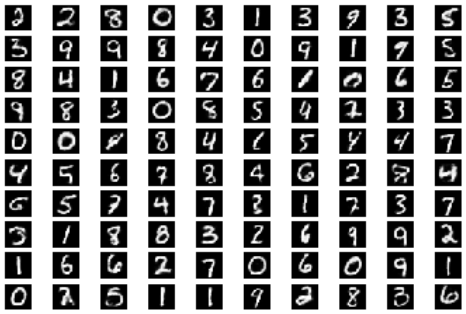


Figure 43: The 100 Reconstructed images from 100 Representative Test Images

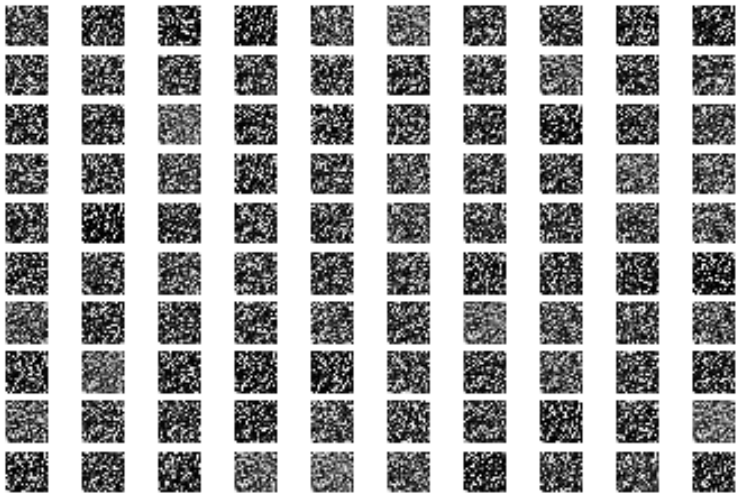


Figure 44: Hidden Layer Activation from 100 Representative Test Images on 1st Layer

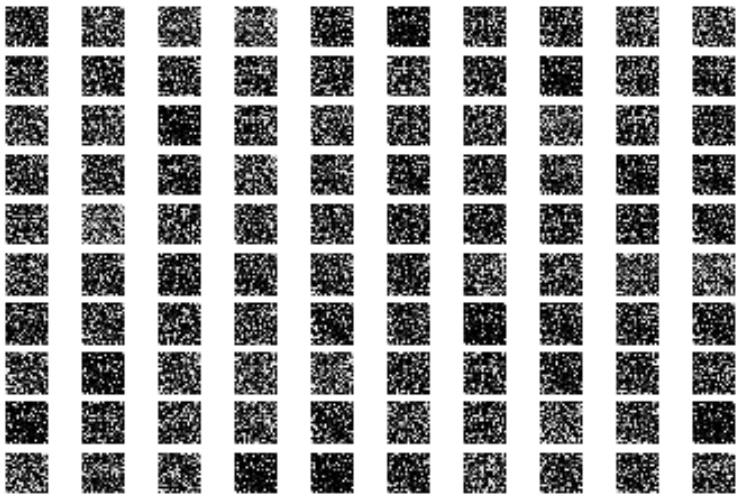


Figure 45: Hidden Layer Activation from 100 Representative Test Images on 2nd Layer

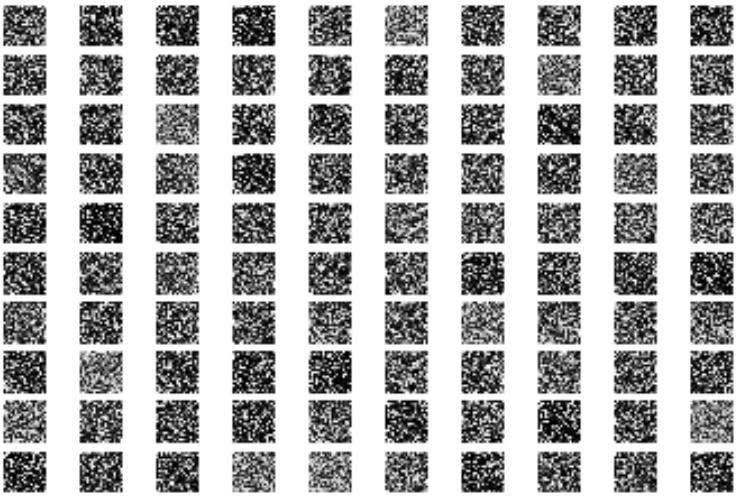


Figure 46: Hidden Layer Activation from 100 Representative Test Images on 3rd Layer

2.3.5 Training Errors and Test Accuracies During Training Five-layer Feed-forward Neural Network

The model has been trained for 25 epochs. After 25 epochs, it can achieve the accuracy about 0.75 as shown in the Figure 48. It is far from the previous result 0.90 as shown in the Figure 35. The training error (Categorical_crossentropy) is decreased from 4.4 at epoch 0 to 0.6 at epoch 24 as shown in the Figure 47. It is also far from the previous result which is decreased from 1.1 at epoch 0 to 0.2 at epoch 24 as shown in the Figure 34.

As momentum is added to the SGD, it takes more epochs for the result to converge. Hence, for the same 25 epochs, the model without momentum has the better performance and higher test accuracy than the model with momentum. However, I believe that if the model can be trained for larger number of epochs, the model with momentum can converge better and have better performance than the current result.

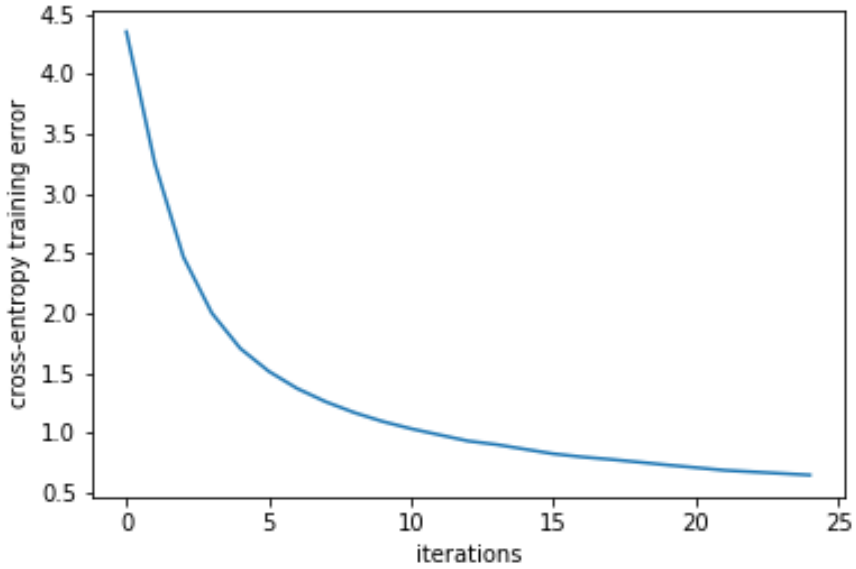


Figure 47: Training Errors During Training 5-Layer Feedforward Neural Network

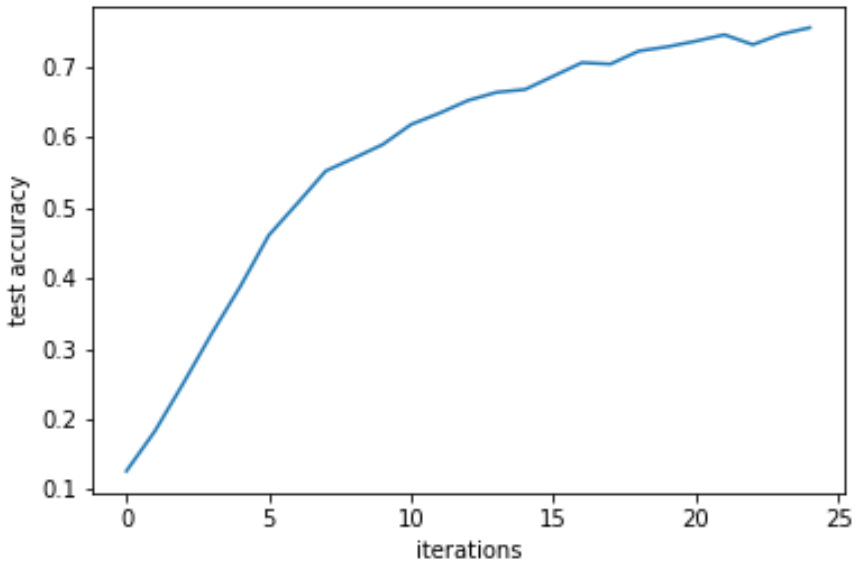


Figure 48: Test Accuracies During Training 5-Layer Feedforward Neural Network