# assignment3

November 22, 2017

Name: Qin Qianqian Student Number: 601098

# 1 Optimization of Fully Convolutional Neural Networks

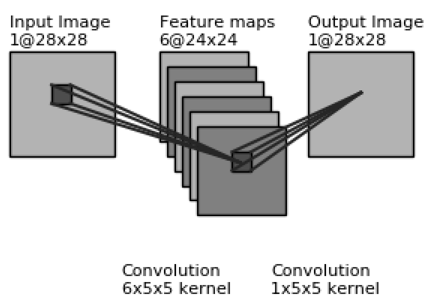
In this exercise, you will use the the model explained in the demonstration and apply some optimization techniques for empirical risk minimization by:

1. Tuning the bias initializations using grid search
2. Implementing the model of momentum and Adam to accelerate learning

Remember the model structure:

```
In [1]: from IPython.display import Image
        Image(filename='images/convnetStructureforAssignment.png',height=800,width=800)
```

Out[1]:




We will begin the assignment by importing necessary python libraries:

```
In [2]: import six.moves.cPickle as pickle
        import matplotlib.pyplot as plt
        import gzip
        import os, sys
        import numpy as np
        import theano
        import theano.tensor as T
        from theano.tensor.nnet import conv2d
        from exercise_helper import load_data, conv_layer
        print('***** Import complete *****')
```

```
***** Import complete *****
```

**Mini-Batch Gradient Descent** was already implemented in both this week and last week's demos. It is also used as an optimization technique. It updates weights incrementally after each iteration and calculates the cost over mini batches. In the below function, *updates* variable shows the update operation for each parameter (weights and biases) based on the calculated cost with the learning rate. You can use this model as a hint for the rest of the homework.

```
In [3]: def gradient_updates_sgd(cost, params, learning_rate):
            # Function to return an update list for the parameters to be updated

            # Inputs:
            # cost: MSE cost Theano variable
            # params :  parameters coming from hidden and output layers
            # learning rate: learning rate defined as hyperparameter

            # Outputs:
            # updates : updates to be made and to be defined in the train_model function.
            updates = []
            # just gradient descent on cost
            for param in params:
                # use append function to make a list for which Theano variables will be
                # updated with which value
                updates.append((param, param - learning_rate*T.grad(cost, param)))
            return updates
```

### 1.0.1 1. Parameter Initialization

Now we will create a function *run_convnet()* to run the experiments. The inside of the function is a little bit different than the demo. First, there is two different ways to initialize the output layer bias outside the *conv_layer()* operation. **Note that you have to write the non-shared bias initialization by yourself. You can return the demo for more explanation of the shared and non-shared bias initialization types.** Secondly, the function updates weights and biases based on the *momentum_type* parameter. Check the *gradient_updates_sgd* code above to get a hint for other parameter update functions. In the non-shared initialization mode, the biases will start from the same value; however now we will have 28x28 matrix, so there will be a different bias for every neuron in the output layer.

```
In [4]: def run_convnet(learning_rate, num_epochs,
                        train_set_x,
                        num_filters, batch_size,
                        momentum_type, bias_type, bias_init=None):
            # Function to create the convolutional neural network, train and
            # evaluate it.

            # Inputs:
            # learning_rate - Learning rate for Stochastic Gradient Descent
            # num_epochs - Number of training epochs
            # train_set_x - training set
            # num_filters - Number of channels for each convolution layer
            #                for e.g. 2 layers - [20, 50].
            #.               layer1 = 20, layer2 = 50
            # batch_size - Mini-batch size to be used
            # momentum_type - Parameter update algorithm to be used
            # bias type - bias initialization type to be used
            #                 shared or non-shared
```

```
# bias_init - initial value for bias

# Outputs:
# Training MSE for each iteration

# random seed to initialize the pseudo-random number generator.
rng = np.random.RandomState(23455)

# compute number of minibatches for training and testing
n_train_batches = train_set_x.get_value(borrow=True).shape[0] // batch_size
n_test_batches = test_set_x.get_value(borrow=True).shape[0] // batch_size

# get the dimensions for input images
import math
D = train_set_x.get_value(borrow=True).shape[1]
L = train_set_x.get_value(borrow=True).shape[0]
W = int(math.sqrt(D))
assert W * W == D

# allocate symbolic variables for the data
# minibatch index
index = T.lscalar()
x = T.matrix('x')


# reshape matrix of rasterized images of shape (batch_size x W x W), W=28
# to a 4D tensor to produce MNIST images with a size of
# (mini_batch_size x 1 x 28 x 28)
input_layer = x.reshape((batch_size, 1, W, W))

# binarize the hidden layer 4D tensor with uniform distribution
input_layer_binarized = ((input_layer +
                                np.random.rand(batch_size,1,W,W)) >
                                1.0).astype(theano.config.floatX)


# construct the first convolutional layer:
# filtering reduces the image size to (24, 24)
# no pooling
# 4D output tensor is thus of shape (mini_batch_size, num_filters[0], 24, 24)
[hidden_layer_output,
hidden_layer_params] = conv_layer(
                                rng, input=input_layer_binarized,
                                image_shape=(batch_size, 1, 28, 28),
                                filter_shape=(num_filters[0], 1, 5, 5),
                                border_mode='valid',
                                activation = T.tanh, bias=None)
```

```
# check the bias type (shared, or non-shared)
# if it is shared, create bias of shape (num_filters[1])
# if not shared, you need to create bias of shape (image_size)
# this bias initialization type will only applied to the output layer
if (bias_type == 'shared') and (bias_init is not None):
    bias_init = np.ones((num_filters[1],),
                        dtype=theano.config.floatX)*bias_init
elif (bias_type == 'non-shared') and (bias_init is not None):

    ################# Insert Your Code Here ###################
    bias_init = np.ones((W,W), dtype=theano.config.floatX)*bias_init
    ################# Insert Your Code Here ###################

# construct the second convolutional layer for output
# filtering increases the image size to (28, 28)
# no pooling
# 4D output tensor is thus of shape (mini_batch_size, num_filters[1], 28, 28)
[output,
output_layer_params] = conv_layer(
                                rng,
                                input=hidden_layer_output,
                                image_shape=(batch_size, num_filters[0], 24, 24),
                                filter_shape=(num_filters[1],
                                num_filters[0], 5, 5),
                                border_mode='full',
                                activation = T.nnet.sigmoid, bias=bias_init)

# compute the cost (Mean Square Error) to be optimized
cost = T.mean((x.flatten(2) - output.flatten(2)) ** 2)
# create a list of all model parameters to be fit by gradient descent
params = output_layer_params + hidden_layer_params

# check the parameter update techniques
# and run the related function to update parameters
if momentum_type == 'sgd':
        updates = gradient_updates_sgd(cost, params, learning_rate)
elif momentum_type == 'momentum':
        updates = gradient_updates_momentum(cost, params, learning_rate)
elif momentum_type == 'Adam':
        updates = gradient_updates_Adam(cost, params, learning_rate)


train_model = theano.function(
                [index],
                cost,
                updates=updates,
                givens={x: train_set_x[index * batch_size: (index + 1) * batch_size
```

```
        # test_ model function is initialized and called for plotting the reconstructed ima
        ind = np.random.randint(n_test_batches)
        test_model = theano.function([],
            [input_layer,
            input_layer_binarized, output],
            givens={x: test_set_x[ind * batch_size: (ind + 1) * batch_size]})


        print('...training model...')

        epoch = 0
        train_mse = []
        while (epoch < num_epochs):
            epoch = epoch + 1
            for minibatch_index in range(n_train_batches):
                iter = (epoch - 1) * n_train_batches + minibatch_index
                train_mse = np.append(train_mse, train_model(minibatch_index))

        [original_input, binarized_input, predicted_output] = test_model()

        print('***** Training Complete *****')
        return train_mse,[original_input, binarized_input, predicted_output]
```

Now we will load a subset of the full dataset. Please check that hyperparameters, such as number of filters for each layer and the learning rate are different from the one in the demo so that you can easily visualize and interpret the differences between optimization algotihms. **Now define an array for the *bias_init_search_array* parameter.** It could be a type of *numpy.asarray()* or *numpy.linspace()* functions or a simple list. Start from negative values and continue to positive values. Define at least 5 different values (5-10 values) for this array.

```
In [5]: # load the dataset and only get the training data
        dataset = 'mnist.pkl.gz'
        datasets = load_data(dataset)

        train_set_x = datasets[0]
        test_set_x = datasets[2]
        print('Training set: %d samples'
          %(train_set_x.get_value(borrow=True).shape[0]))
        print('Test set: %d samples'
          %(test_set_x.get_value(borrow=True).shape[0]))

        # Define hyperparameters
        num_filters = [6, 1]
        batch_size = 64
        learning_rate = 0.01
        num_epochs = 3

        ################ Insert Your Code Here ##################
```

```
bias_init_search_array = np.linspace(-2.5,2,num=10)
################# Insert Your Code Here ###################
```

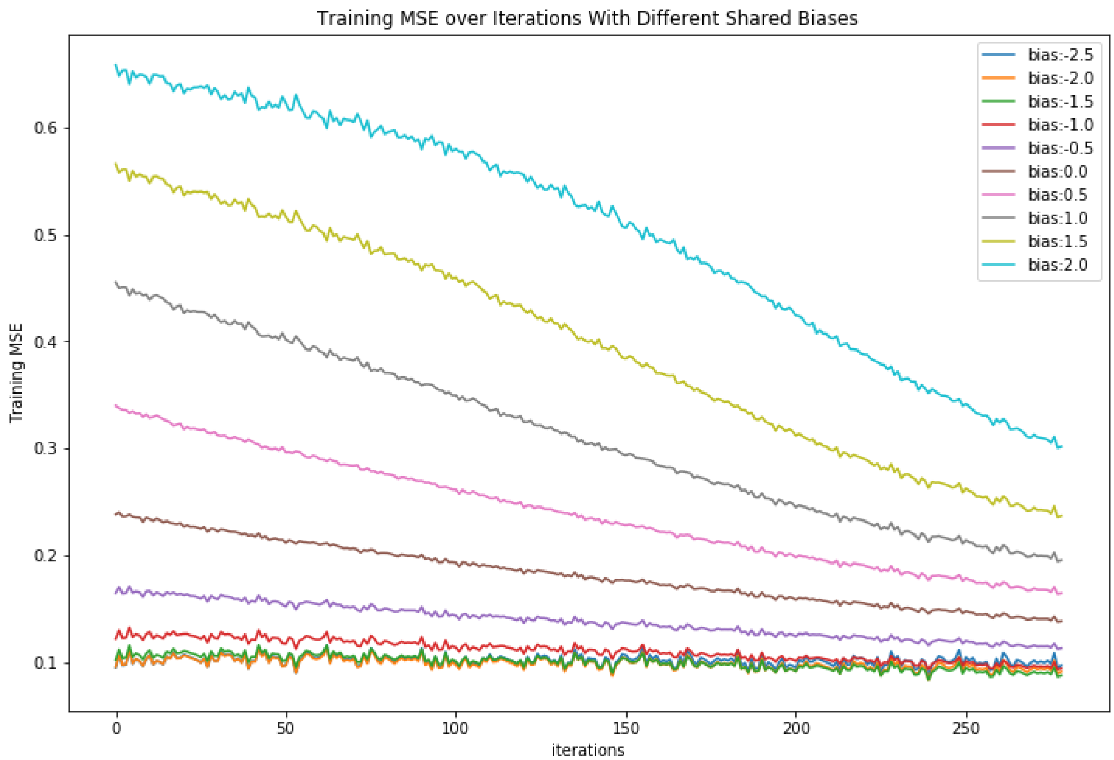
```
***** Loading data *****
Training set: 6000 samples
Test set: 2000 samples
```

Now we calculate the training MSE error for each shared-type bias initialization for the output layer and plot the MSE for each iteration. **Run the code snippet below and comment on the results in the Conclusions section at the end of this assignment.**

```
In [6]: # create figure for plots
        fig = plt.figure(figsize=(12, 8))
        # run experiments and add the resulting MSE to the plot function
        for bias in bias_init_search_array:
            print('for shared bias %f' %bias)
            train_mse_for_iterations, _ = run_convnet(
                                                      learning_rate,
                                                      num_epochs,
                                                      train_set_x,
                                                      num_filters,
                                                      batch_size,
                                                      momentum_type='sgd',
                                                      bias_type='shared', bias_init=bias)

            print('Training MSE after training is done: %f'
                  %train_mse_for_iterations[-1])
            plt.plot(train_mse_for_iterations, label = 'bias:'+str(bias))
        plt.xlabel('iterations')
        plt.ylabel('Training MSE')
        plt.title('Training MSE over Iterations With Different Shared Biases')
        plt.legend()
        plt.show()
```

```
for shared bias -2.500000
...training model...
***** Training Complete *****
Training MSE after training is done: 0.096995
for shared bias -2.000000
...training model...
***** Training Complete *****
Training MSE after training is done: 0.091079
for shared bias -1.500000
...training model...
***** Training Complete *****
Training MSE after training is done: 0.087513
for shared bias -1.000000
...training model...
```

```
***** Training Complete *****
Training MSE after training is done: 0.094183
for shared bias -0.500000
...training model...
***** Training Complete *****
Training MSE after training is done: 0.113136
for shared bias 0.000000
...training model...
***** Training Complete *****
Training MSE after training is done: 0.138410
for shared bias 0.500000
...training model...
***** Training Complete *****
Training MSE after training is done: 0.164598
for shared bias 1.000000
...training model...
***** Training Complete *****
Training MSE after training is done: 0.195237
for shared bias 1.500000
...training model...
***** Training Complete *****
Training MSE after training is done: 0.236679
for shared bias 2.000000
...training model...
***** Training Complete *****
Training MSE after training is done: 0.301556
```

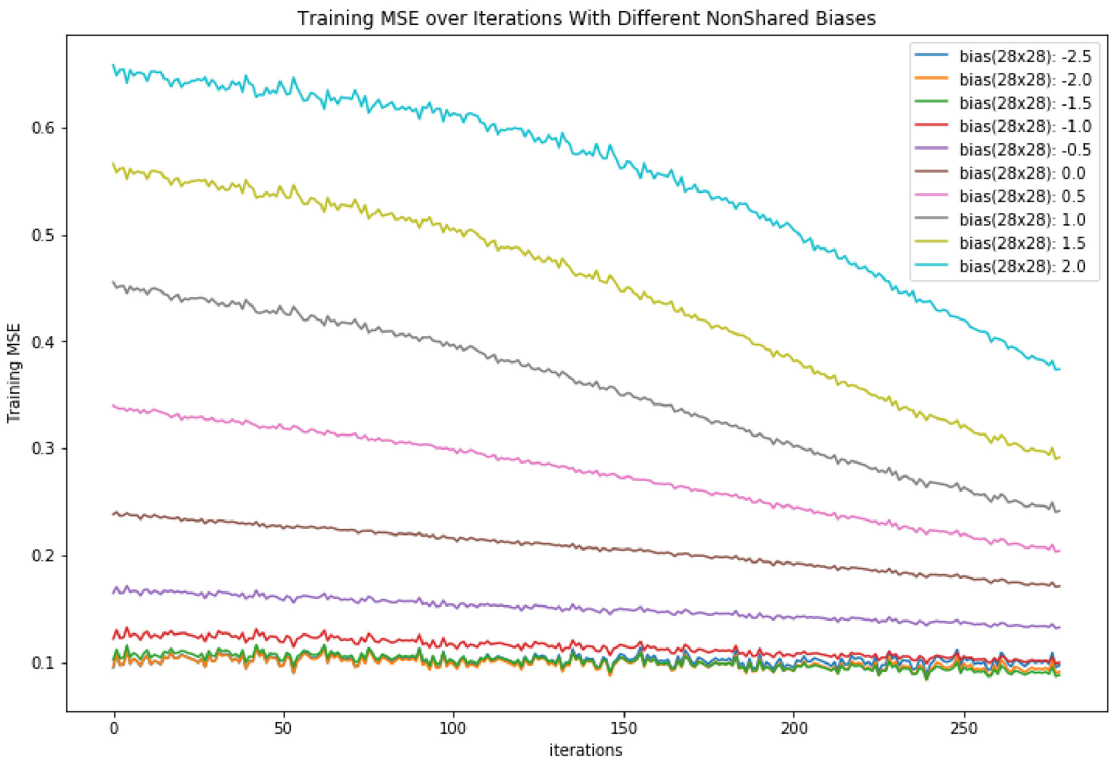



Now we are testing the non-shared initialization for the bias in the output layer. You will use the same values defined previously. **Remember that, in the *run_convnet()* function of Part 1, you have to fill in the initialization line for this mode. Again, after plotting the results, add your comments to the Conclusion section**

```
In [7]: # create figure for plots
        fig = plt.figure(figsize=(12, 8))
        # run experiments and add the resulting MSE to the plot function
        for bias in bias_init_search_array:
            print('for non-shared 28x28 bias %f' %bias)
            train_mse_for_iterations, _ = run_convnet(
                                                    learning_rate,
                                                    num_epochs,
                                                    train_set_x,
                                                    num_filters,
                                                    batch_size, momentum_type='sgd',
                                                    bias_type='non-shared', bias_init=bias)

            print('Training MSE after training is done: %f' %train_mse_for_iterations[-1])
            plt.plot(train_mse_for_iterations, label = 'bias(28x28): '+str(bias))
        plt.xlabel('iterations')
        plt.ylabel('Training MSE')
        plt.title('Training MSE over Iterations With Different NonShared Biases')
```

```
        plt.legend()
        plt.show()
```

```
for non-shared 28x28 bias -2.500000
...training model...
***** Training Complete *****
Training MSE after training is done: 0.097080
for non-shared 28x28 bias -2.000000
...training model...
***** Training Complete *****
Training MSE after training is done: 0.091053
for non-shared 28x28 bias -1.500000
...training model...
***** Training Complete *****
Training MSE after training is done: 0.087830
for non-shared 28x28 bias -1.000000
...training model...
***** Training Complete *****
Training MSE after training is done: 0.099509
for non-shared 28x28 bias -0.500000
...training model...
***** Training Complete *****
Training MSE after training is done: 0.132368
for non-shared 28x28 bias 0.000000
...training model...
***** Training Complete *****
Training MSE after training is done: 0.170978
for non-shared 28x28 bias 0.500000
...training model...
***** Training Complete *****
Training MSE after training is done: 0.204089
for non-shared 28x28 bias 1.000000
...training model...
***** Training Complete *****
Training MSE after training is done: 0.241445
for non-shared 28x28 bias 1.500000
...training model...
***** Training Complete *****
Training MSE after training is done: 0.291294
for non-shared 28x28 bias 2.000000
...training model...
***** Training Complete *****
Training MSE after training is done: 0.373903
```




### 1.0.2 2. Parameter Update

In the second part of the assignment you will write parameter update algorithms for both momentum and Adam techniques. Please check the *gradient_update_sgd()* function, hints given below for writing your codes, as well as read section 8.3 and 8.6 of the deep learning book for more detailed information on these methods.

```
In [8]: def gradient_updates_momentum(cost, params, learning_rate, momentum=0.9):
            # Function to return an update list for the parameters to be updated

            # Inputs:
            # cost: MSE cost Theano variable
            # params :  parameters coming from hidden and output layers
            # learning rate: learning rate defined as hyperparameter
            # momentum : momentum parameter,
            #             usually a high value (0.8, 0.9) was chosen for momentum
            # Outputs:
            # updates : updates to be made and to be defined in the train_model function
            updates = []
            for param in params:
                # for each parameter, we'll create a velocity shared variable
                # since we need to remember the velocity and update in each iteration
                # it should be the same size with the param
```

```
        # we initialize it to 0

        # hint 1: remember the momentum algorithm
        # for each parameter:
        # compute gradient estimate
        # compute velocity update
        # compute parameter update

        # hint2 : use updates.append() function similar to the
        # gradient_updates_sgd() function above

        ################# Insert Your Code Here ##################
        ''' One kind of momentum
        velocity = theano.shared(param.get_value(borrow=True)*0.,
                                 broadcastable=param.broadcastable)
        new_velocity = momentum*velocity+(1-momentum)*T.grad(cost,param)
        updates.append((velocity,new_velocity))
        updates.append((param,param-learning_rate*new_velocity))
        '''
        #Another variant of momentum
        velocity = theano.shared(param.get_value(borrow=True)*0.)
        new_velocity = momentum*velocity-learning_rate*T.grad(cost,param)
        updates.append((velocity,new_velocity))
        updates.append((param,param+new_velocity))

        ################# Insert Your Code Here ##################

    return updates
```

```
In [9]: def gradient_updates_Adam(cost, params, learning_rate):
            # Function to return an update list for the parameters to be updated

            # cost: MSE cost Theano variable
            # params :  parameters coming from hidden and output layers
            # learning rate: learning rate defined as hyperparameter

            # Outputs:
            # updates : updates to be made and to be defined in the train_model function.
            updates = []
            eps = 1e-4 # small constant used for numerical stabilization.
            beta1 = 0.9
            beta2 = 0.999
            # beta1 and beta2 are the exponential decay rates
            # for moment estimates, in [0,1).
            # suggested defaults: 0.9 and 0.999 respectively

            for param in params:
                # hint 1: create a shared variable for time step
```

```
            # initialize time step t = 0

            # hint 2: create shared variables for 1st and 2nd moment variables
            # they should be the same size with the param
            # initialize 1st and 2nd moment variables s = 0, r = 0

            # hint 3: the initializations of these parameters
            # will follow the same structure in momentum function
            # (check the velocity initialization part)

            # hint 4: remember the Adam algorithm
            #         compute gradient
            #         update biased first moment estimate
            #         update biased second moment estimate
            #         correct bias in first moment
            #         correct bias in second moment
            #         compute parameter update

            ################# Insert Your Code Here ##################
            t=theano.shared(1.)
            s=theano.shared(param.get_value(borrow=True)*0.)
            r=theano.shared(param.get_value(borrow=True)*0.)

            g = T.grad(cost,param)
            new_s = beta1*s +(1-beta1)*g
            new_r = beta2*r +(1-beta2)*g*g
            s_bc = new_s/(1-(beta1**t))
            r_bc = new_r/(1-(beta2**t))
            theta = -learning_rate*s_bc/(T.sqrt(r_bc)+eps)
            new_param = param+theta
            new_t = t+1.
            updates.append((s,new_s))
            updates.append((r,new_r))
            updates.append((param, new_param))
            updates.append((t,new_t))
            ################# Insert Your Code Here ##################
        return updates
```

Now, we will run experiments for simple gradient descent, momentum and Adam parameter update techniques. Here, you will choose a *bias_init* value based on the results from the previous part. **Find out the best value for output bias initialization (shared version) and use the same value in the runs below.** When you run each cell, you will get a separate plot for for each method. **After getting this plot, comment on the results in Conclusion section.**

```
In [10]: ################# Insert Your Code Here ##################
         bias_init = -1.50
         ################# Insert Your Code Here ##################
         sgd_train_mse_for_iterations, _ = run_convnet(
```

```
                                                    learning_rate,
                                                    num_epochs,
                                                    train_set_x,
                                                    num_filters,
                                                    batch_size, momentum_type='sgd',
                                                    bias_type='shared', bias_init=bias_init)

        print('Training MSE after training with sgd is done: %f'
              %sgd_train_mse_for_iterations[-1])
```

```
...training model...
***** Training Complete *****
Training MSE after training with sgd is done: 0.087503
```

```
In [11]: momentum_train_mse_for_iterations, _ = run_convnet(
                                                        learning_rate,
                                                        num_epochs,
                                                        train_set_x,
                                                        num_filters,
                                                        batch_size, momentum_type='momentum',
                                                        bias_type='shared', bias_init=bias_ini

        print('Training MSE after training with momentum is done: %f'
              %momentum_train_mse_for_iterations[-1])
```

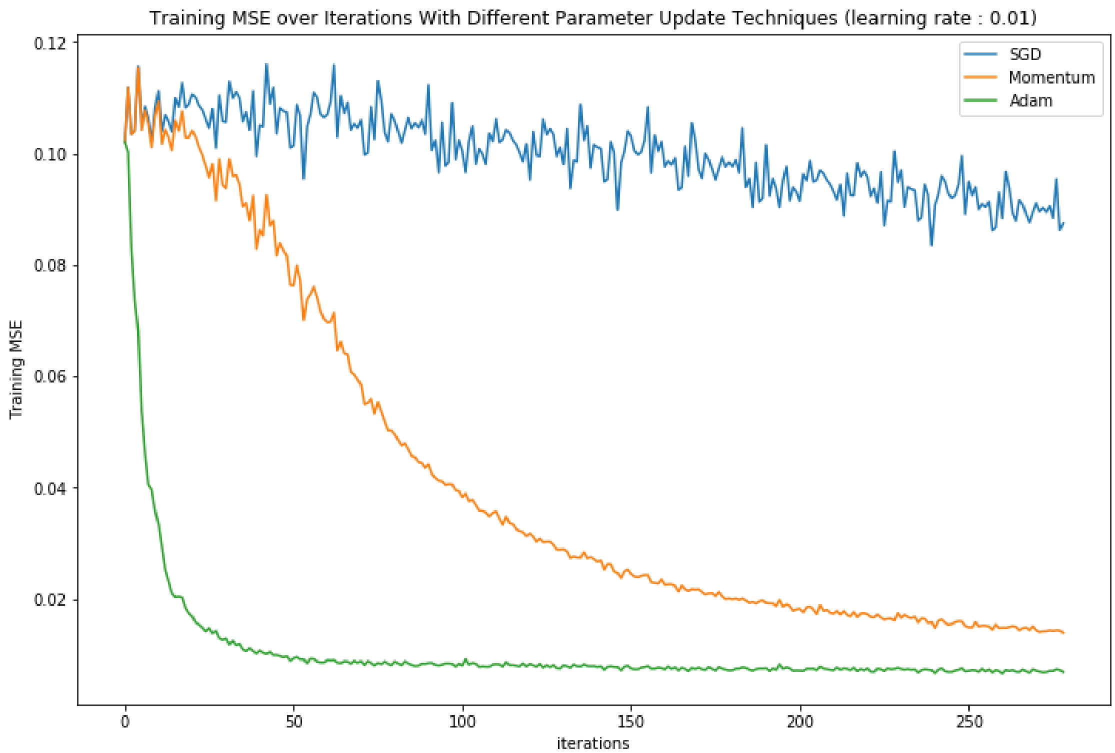
```
...training model...
***** Training Complete *****
Training MSE after training with momentum is done: 0.013970
```

```
In [12]: adam_train_mse_for_iterations, _ = run_convnet(
                                                    learning_rate,
                                                    num_epochs,
                                                    train_set_x,
                                                    num_filters,
                                                    batch_size, momentum_type='Adam',
                                                    bias_type='shared', bias_init=bias_init)

        print('Training MSE after training with Adam is done: %f'
              %adam_train_mse_for_iterations[-1])
```

```
...training model...
***** Training Complete *****
Training MSE after training with Adam is done: 0.006908
```

The section below is for plotting the loss values during the training with different parameter update techniques. There will be no change for this cell.

```
In [13]: fig = plt.figure(figsize=(12, 8))
         plt.plot(sgd_train_mse_for_iterations, label = 'SGD')
         plt.plot(momentum_train_mse_for_iterations, label = 'Momentum')
         plt.plot(adam_train_mse_for_iterations, label = 'Adam')
         plt.xlabel('iterations')
         plt.ylabel('Training MSE')
         plt.title('Training MSE over Iterations With Different Parameter Update Techniques (le
         plt.legend()
         plt.show()
```

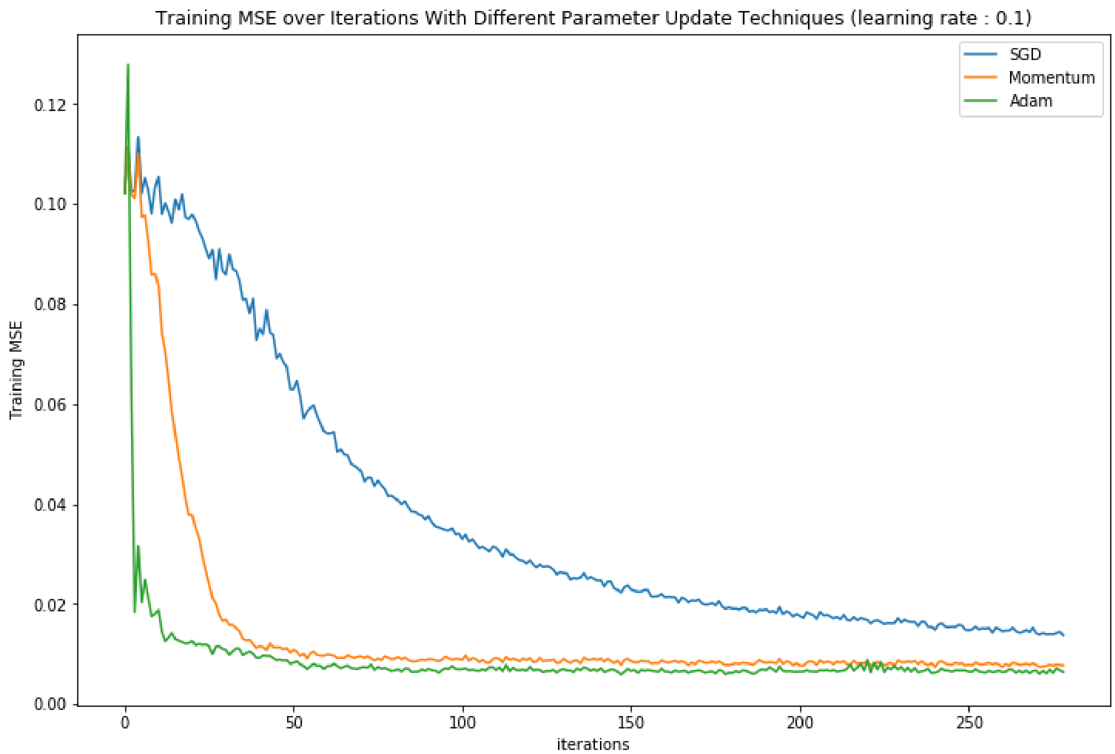



Now you will repeat the experiments with different learning rates = 0.1 and 0.001. After getting the results from parameter update techniques used with different learning rates, report the best model out of these nine models.

```
In [14]: learning_rate = 0.1
         print('for learning rate %f:' %(learning_rate))
         sgd_train_mse_for_iterations, _ = run_convnet(
                                                    learning_rate,
                                                    num_epochs,
                                                    train_set_x,
                                                    num_filters,
                                                    batch_size, momentum_type='sgd',
                                                    bias_type='shared', bias_init=bias_init)
```

```python
        print('Training MSE after training with sgd is done: %f'
              %sgd_train_mse_for_iterations[-1])

        momentum_train_mse_for_iterations, _ = run_convnet(
                                                            learning_rate,
                                                            num_epochs,
                                                            train_set_x,
                                                            num_filters,
                                                            batch_size, momentum_type='momentum',
                                                            bias_type='shared', bias_init=bias_in

        print('Training MSE after training with momentum is done: %f'
              %momentum_train_mse_for_iterations[-1])

        adam_train_mse_for_iterations, _ = run_convnet(
                                                        learning_rate,
                                                        num_epochs,
                                                        train_set_x,
                                                        num_filters,
                                                        batch_size, momentum_type='Adam',
                                                        bias_type='shared', bias_init=bias_init)

        print('Training MSE after training with Adam is done: %f'
              %adam_train_mse_for_iterations[-1])
        fig = plt.figure(figsize=(12, 8))
        plt.plot(sgd_train_mse_for_iterations, label = 'SGD')
        plt.plot(momentum_train_mse_for_iterations, label = 'Momentum')
        plt.plot(adam_train_mse_for_iterations, label = 'Adam')
        plt.xlabel('iterations')
        plt.ylabel('Training MSE')
        plt.title('Training MSE over Iterations With Different Parameter Update Techniques (l
        plt.legend()
        plt.show()
```

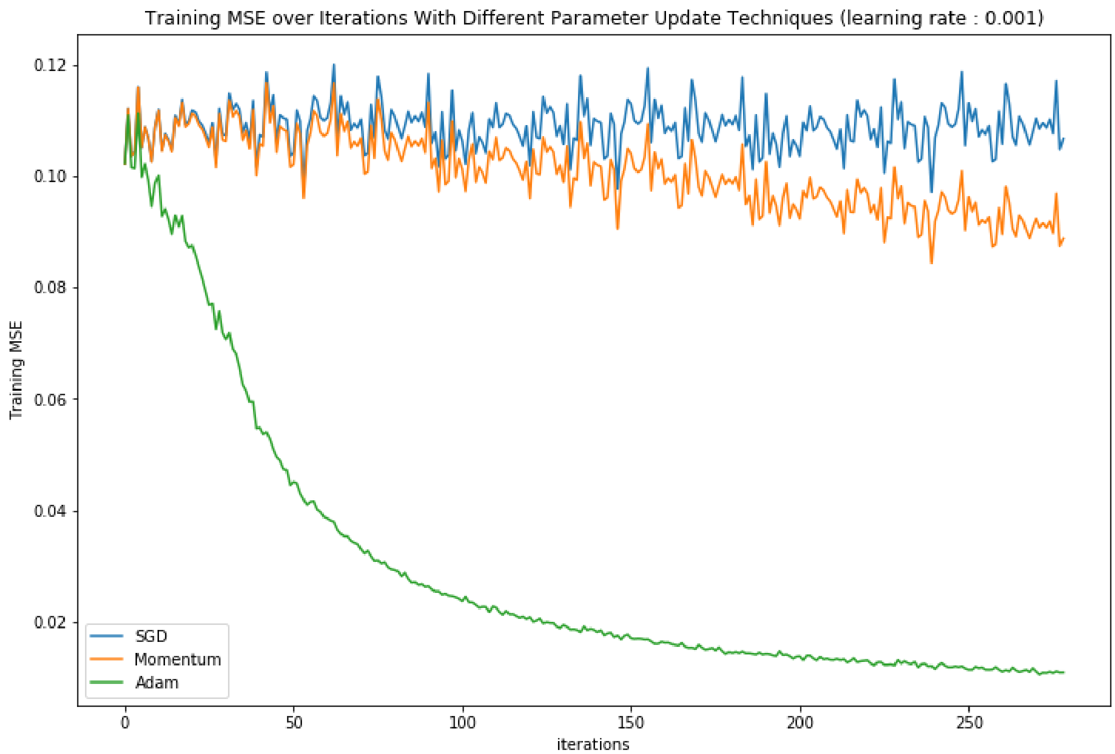
```
for learning rate 0.100000:
...training model...
***** Training Complete *****
Training MSE after training with sgd is done: 0.013760
...training model...
***** Training Complete *****
Training MSE after training with momentum is done: 0.007737
...training model...
***** Training Complete *****
Training MSE after training with Adam is done: 0.006436
```




```
In [15]: learning_rate = 0.001
         print('for learning rate %f:' %(learning_rate))
         sgd_train_mse_for_iterations, _ = run_convnet(
                                                     learning_rate,
                                                     num_epochs,
                                                     train_set_x,
                                                     num_filters,
                                                     batch_size, momentum_type='sgd',
                                                     bias_type='shared', bias_init=bias_init)

         print('Training MSE after training with sgd is done: %f'
               %sgd_train_mse_for_iterations[-1])

         momentum_train_mse_for_iterations, _ = run_convnet(
                                                         learning_rate,
                                                         num_epochs,
                                                         train_set_x,
                                                         num_filters,
                                                         batch_size, momentum_type='momentum',
                                                         bias_type='shared', bias_init=bias_in

         print('Training MSE after training with momentum is done: %f'
               %momentum_train_mse_for_iterations[-1])
```

```
    adam_train_mse_for_iterations, _ = run_convnet(
                                            learning_rate,
                                            num_epochs,
                                            train_set_x,
                                            num_filters,
                                            batch_size, momentum_type='Adam',
                                            bias_type='shared', bias_init=bias_init)

    print('Training MSE after training with Adam is done: %f'
          %adam_train_mse_for_iterations[-1])

    fig = plt.figure(figsize=(12, 8))
    plt.plot(sgd_train_mse_for_iterations, label = 'SGD')
    plt.plot(momentum_train_mse_for_iterations, label = 'Momentum')
    plt.plot(adam_train_mse_for_iterations, label = 'Adam')
    plt.xlabel('iterations')
    plt.ylabel('Training MSE')
    plt.title('Training MSE over Iterations With Different Parameter Update Techniques (le
    plt.legend()
    plt.show()
```

```
for learning rate 0.001000:
...training model...
***** Training Complete *****
Training MSE after training with sgd is done: 0.106640
...training model...
***** Training Complete *****
Training MSE after training with momentum is done: 0.088725
...training model...
***** Training Complete *****
Training MSE after training with Adam is done: 0.010894
```

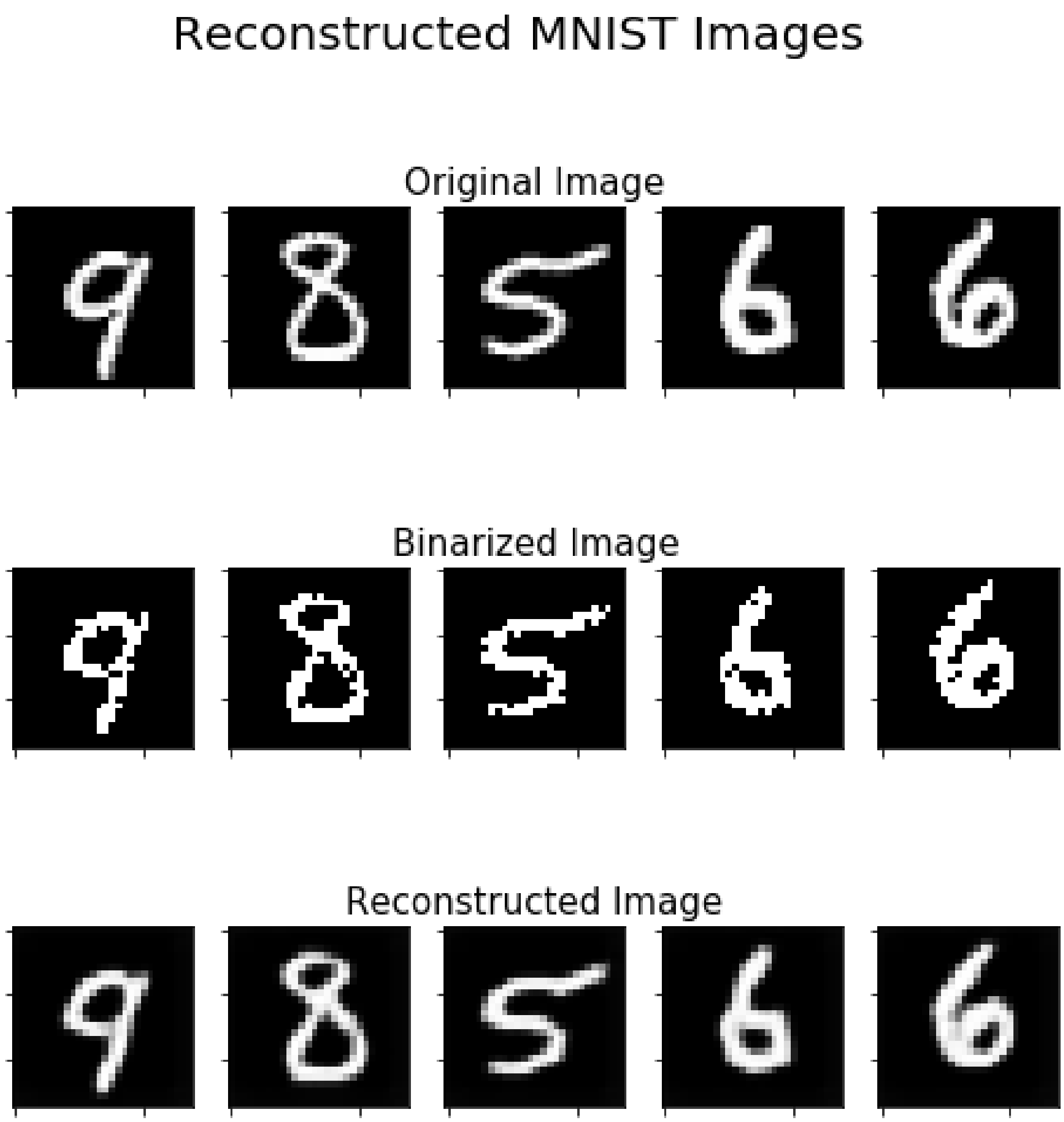



Finally, select the best model giving the minimum MSE by looking the parameter update technique and the learning rate, then run the code snippet below by inserting the parameters for your best model in order to see the predicted images from their binarized versions.

```
In [16]: ################# Insert Your Code Here ###################
         learning_rate = 0.1
         momentum_type_string = "Adam"
         ################# Insert Your Code Here ###################
         _, [original_input, binarized_input, predicted_output] = run_convnet(
                                                     learning_rate,
                                                     num_epochs,
                                                     train_set_x,
                                                     num_filters,
                                                     batch_size, momentum_type=momentum_type_string,
                                                     bias_type='shared', bias_init=bias_init)

         # four axes, returned as a 2-d array
         f, axarr = plt.subplots(3, 5, figsize=(8, 8))
         for i in range(5):
             axarr[0,i].imshow(
                 original_input[i].reshape(28,28).astype('float32'), cmap='gray')
             axarr[1,i].imshow(
                 binarized_input[i].reshape(28,28).astype('float32'), cmap='gray')
             axarr[2,i].imshow(
```

```
                predicted_output[i].reshape(28,28).astype('float32'), cmap='gray')
            # Turn off tick labels
            axarr[0,i].set_yticklabels([])
            axarr[0,i].set_xticklabels([])
            axarr[1,i].set_yticklabels([])
            axarr[1,i].set_xticklabels([])
            axarr[2,i].set_yticklabels([])
            axarr[2,i].set_xticklabels([])
        axarr[0,2].set_title('Original Image', fontsize=15)
        axarr[1,2].set_title('Binarized Image', fontsize=15)
        axarr[2,2].set_title('Reconstructed Image', fontsize=15)
        plt.suptitle('Reconstructed MNIST Images', fontsize=20)
        plt.show()
```

```
...training model...
***** Training Complete *****
```




### 1.0.3 Conclusions

Firstly, Experimenting with 10 value between [-2.5,2.5] as bias initialization.

The best choice among them, which gives the smallest training MSE, is around -1.50. Taking -1.50 as benchmark of bias initialization, the training MSE increases, with the bias initialization decresing or increasing. So the choosen bias initialization is -1.50.

Then Experimenting with 3 different updates algorithms, SGD, Momentum and Adam, under the conditions that the learning rate is 0.01,0.1,0.001 separately.

Under the different learning rates, Adam always performs best, giving the smallest traning MSE, while SGD always gives the biggest traning MSE. So Adam updates algorithm is choosen.

The 3 algorithms performs worst when the learning rate is 0.001. The learnning rate is too small, so the trainng MSE converges too slowly. The learning rate of 0.01 also shows a similar (but relatively better) situation, especially for SGD algorithm.

The 3 algorithms performs best when the learning rate is 0.1. The Adam algorithm gives the smallest training MSE among all the experiments conducted(0.006436). We notice that the momentum also gives a very small training MSE (0.007737).

So finally the parameters choosen: bias initialization = -1.50, algorithm = Adam, learning rate = 0.1.