CS 534 Machine Learning

Project 3, due Sunday, April 15

Chenxi Cai
Yifei Ren
Qingfeng (Kee) Wang

(Notice, all links (table of contents, figures references, table references etc.) are clickable! You can download this PDF and open it using Preview or Adobe Reader to enable this convenient feature.)

Contents

1 Introduction and methods **3**

2 Adjust hyper-parameters: stage I **3**
2.1 Size of the filters . . . 3
2.2 Number of filters . . . 4
2.3 Number of convolution and pooling layers . . . 5
2.4 Learning rate . . . 7
2.5 The ways of paddings . . . 8
2.6 The choices of activation functions . . . 9

3 Adjust hyper-parameters: stage II **10**
3.1 Adam optimizer . . . 10
3.2 Momentum optimizer and SGD . . . 10

4 Final model and analysis **12**

5 Source codes **14**

1 Introduction and methods

Convolutional neural network (CNN) proves to be one of the most successful training models. This time we will use CNN to train MNIST data which is a database of handwritten figures in black and white. It contains 55000 training images and 10000 testing images. In this report, we will split 55000 training data into 80% data for training and 20% for validation. All data are trained in 5000 steps. Validation data accuracy and loss were monitor in every 500 steps. Gradient were calculated in batch size 200.

We constructed a base model and then adjust hyper-parameters in two stages.

First, in section 2, we tune the size and number of the filters, number of convolution and polling layers, learning rate, the ways of paddings and the choices of activations functions. After choosing the optimized hyper-parameters, we finally the behavior of three different optimizers in section 3.

A more detailed discussion of finally chosen model is provided in the section 4. At the end of the report, code of base model is provided.

2 Adjust hyper-parameters: stage I

In this section we first adjust several hyper-parameters listed in the requirement 2. Hyper-parameters are determined by the validation accuracy (or loss) and the time efficiency.

2.1 Size of the filters

In base model, the size of filter is 5-by-5. According to Stanford University video courses lecture 5, common settings of filter size are listed in the Table 1.

Table 1: Commonly adjusted filter sizes. Where F is filters' spatial extend (filter dimension), S is the stride, P is the amount of zero padding.

F	S	P
3	1	1
5	1	2
1	1	0

Here we tried three different cases of filter's size: 1×1, 3×3 and 5×5.

Now we know that the final accuracy after 5000 steps are shown in Table 2. From the table we can obtain that 5×5 is the best size.

Table 2: Validation accuracy at 5000 steps

Size of filters	1×1	3×3	5×5
Accuracy	0.7676	0.9133	0.9252

2.2 Number of filters

In base model, filter number is 32 for conv 1 and 64 for conv 2. According to Stanford University video courses lecture 5, common settings of filter number are power of 2: 32, 64, 128 and 512, etc.

Here we tried four different combinations of number of filters (conv1, conv2): (32, 32), (32, 64), (64, 64) and (128, 128). Now we know that the final validation accuracy at 5000 steps are shown in Table 3.

Table 3: Validation ccuracy at 5000 steps

Number of filter (conv1, conv2)	(32, 32)	(32, 64)	(64, 64)	(128, 128)
Accuracy at 1000 step	0.5977	0.7482	0.7333	0.7294
Accuracy at 5000 step	0.9221	0.9275	0.9268	0.9265

From the table, the accuracy increases little with number, the final accuracies are very close. However, from the accuracy plots, we ca see that the initial accuracies at 1000 steps have large differences. In according to the loss trending shown Figure 1, all four models shows similar loss descending curves with filter (32, 64) the steepest. It also should be noted filter size greater than (32, 64) takes a lot more time to finish the training but only gives marginal accuracy improvement. As a result, it is easy to conclude that filter (32, 64) is the best choice.

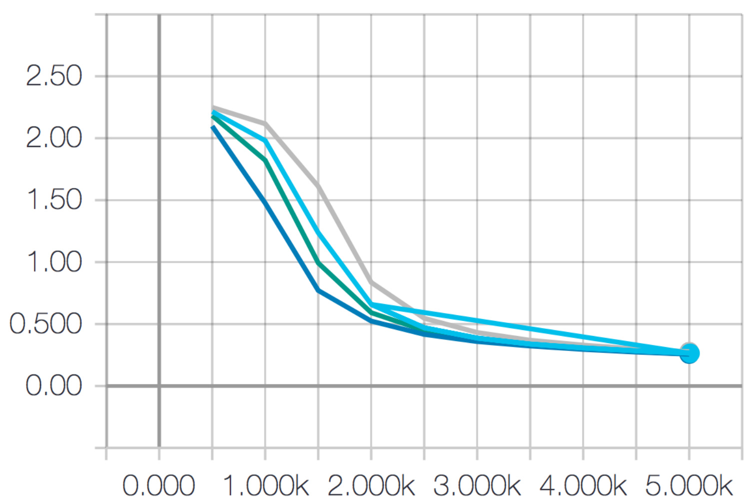

Figure 1: The figure shows the trend of loss function behavior of different filters. The steepest curve at bottom belongs to filter (32, 64). See tensorboard event file for detailed legend.

2.3 Number of convolution and pooling layers

This is somewhat harder to tune. Generally, we have one pooling layer after a convolutional layer, and the dimension is cut into half. Begin with single image size 28×28, it is easy to see we can apply at most 2 pooling layers if we are using `same` padding. As a result, we only choose 1 convolutional layer plus 1 pooling layers (structure shown in Figure 2) and 2 convolutional layers plus 2 pooling layers (base model, structure shown in Figure 3)) to compare.

The behavior of two different CNN structures are shown in Figure 4. At 5000 step, the base model have a lower loss. As a result, we choose 2 convolutional layers plus 2 pooling layers as the final structure.

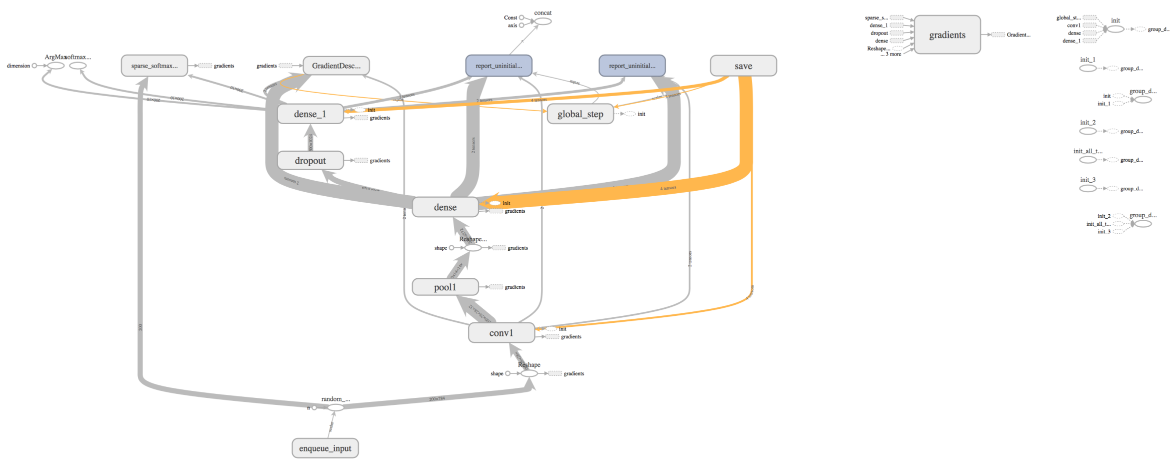

Figure 2: Graph for 1 convolutional layer and 1 pooling layer. Zoom in to see details.

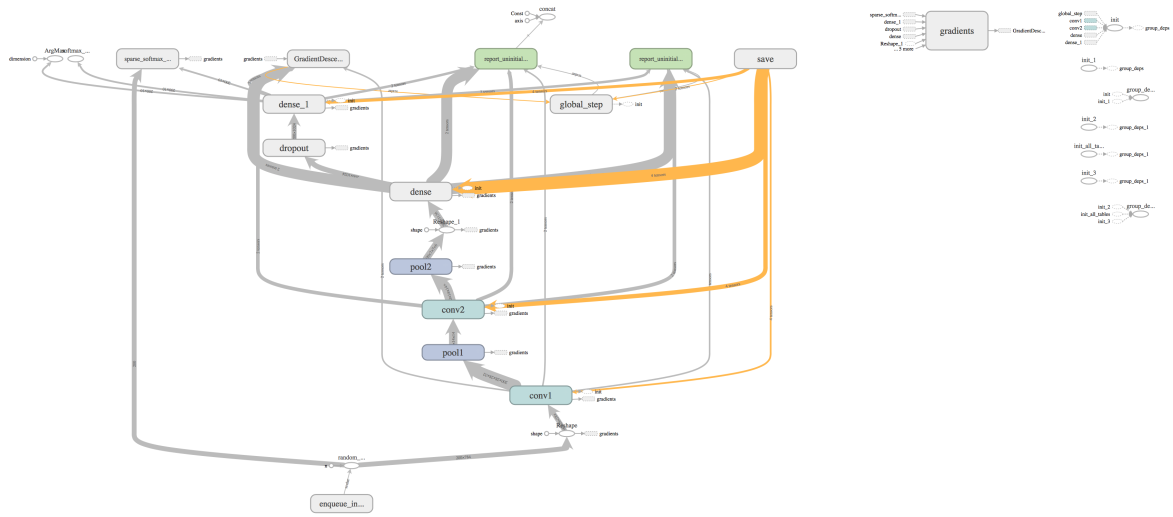

Figure 3: Graph for 2 convolutional layers and 2 pooling layer.Zoom in to see details.

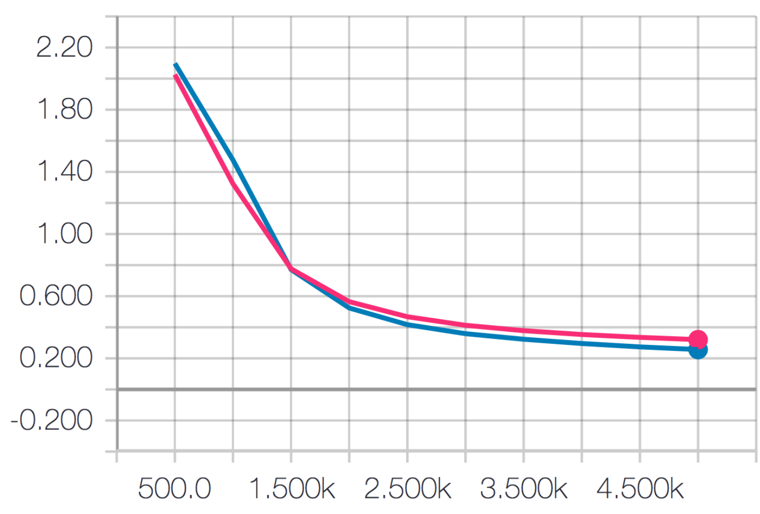

Figure 4: The comparison of loss of two different CNN structures. Red curve, with final loss 0.3203, correspond to 1 convolutional layer and 1 pooling layer. Blue curve, with final loss 0.2567, correspond to 2 convolutional layers and 2 pooling layers.

2.4 Learning rate

There are several choices of learning rate from `1E-5, 1E-4, 1E-3, 1E-2, 1E-1` to `1`. It seems the best learning rate is `1E-1` as it converges to a very high accuracy (low loss) very fast. The Figure 5 shows the trend of learning rate. The loss of curve steadily decrease faster as we increase learning rate from `1E-5` to `1E-1`. However, when learning rate equals to `1`, the loss function remains around 2.25 and does not decrease. As a result, learning rate at `1E-1` is the the best learning rate we choose.

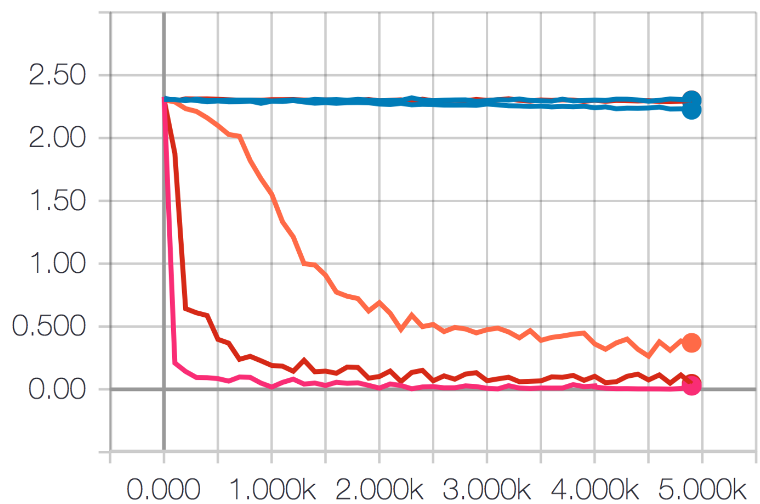

Figure 5: Loss function of training data. The detailed assignment of curves is shown in the event file and can be viewed in Tensorboard. The curve that descent the fastest correspond to learning rate equals to `1E-1`.

2.5 The ways of paddings

So far there are two types of padding methods, i.e., `same` (base model) and `valid`. Here we compared the difference between `same` and `valid`. The accuracy is presented in Figure 6. It is obvious that using `same` padding always gives the better accuracy in this test case. The loss of training is presented in Figure. 7 and shows that the loss of training data decreases much faster when using `same` padding. It can be inferred that `same` padding gives the best result and is the final hyper-parameter we chose.

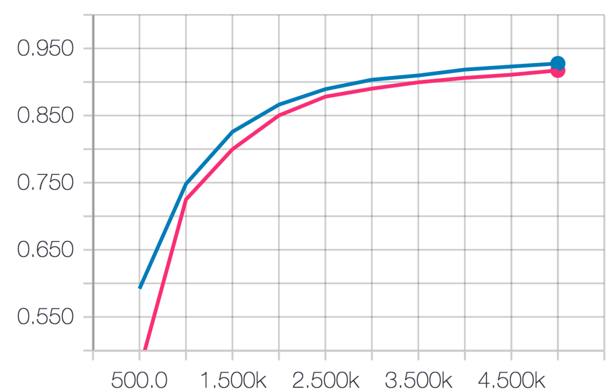

Figure 6: The figure shows the accuracy of validation data of `same` (blue) and `valid` (magenta) padding.

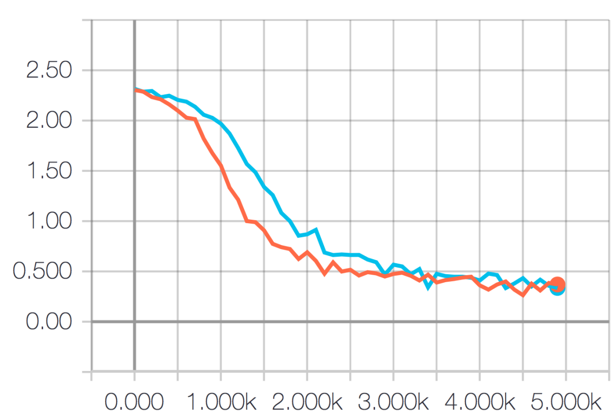

Figure 7: The figure shows the loss training data of `same` (orange) and `valid` (cyan) padding.

2.6 The choices of activation functions

(To be finished by Yifei) The training loss is shown in Figure 9 while the validation accuracy is shown in Figure 8. Obviously Sigmoid activation function does not learn much in this test case. As a result we chose `ReLu` as our final hyper-parameter.

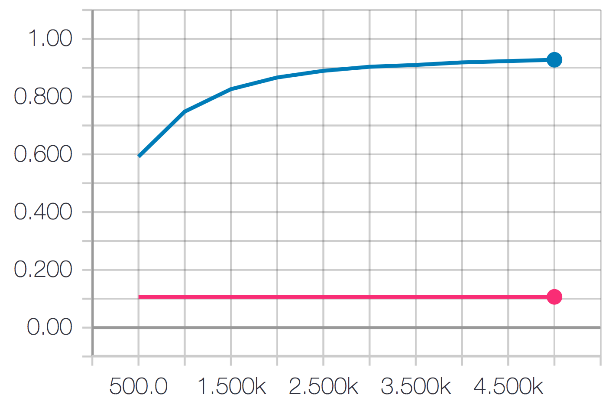

Figure 8: The figure shows the accuracy of validation data of `ReLu` (blue) activation function and `Sigmoid` (magenta) activation function.

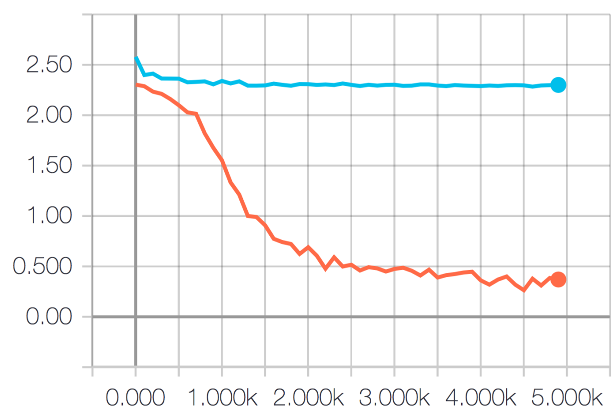

Figure 9: The figure shows the loss training data of `ReLu` (orange) and `Sigmoid` (cyan) activation function.

3 Adjust hyper-parameters: stage II

In this chapter, we use the hyper-parameters optimzied in section 2 and adjust the optimizer. There are three choices of optimizers: `Gradient Descent` (base model), `Adam Optimizer` and `Momentum Optimizer`. We will explain more about later two optimizers.

3.1 Adam optimizer

Adaptive Moment Estimation (Adam) is a method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients, Adam also keeps an exponentially decaying average of past gradients, similar to momentum. Here we set the parameters of AdamOptimizer as:

```
optimizer = tf.train.AdamOptimizer(learning_rate=1E-1)
```

The plot of training loss is showed as Figure 10. Obviously the loss function is very high does not decrease at all. As a matter of fact the accuracy rate is around 10%.

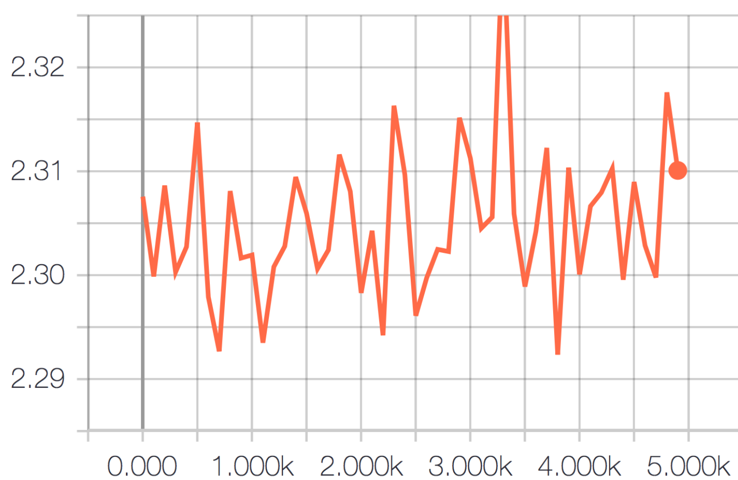

Figure 10: Loss function of training data using Adam optimizer.

3.2 Momentum optimizer and SGD

Steepest gradient descent (SGD) has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another, which are common around local optima. Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations.

Here we set the parameters of MomentumOptimizer as:

```
optimizer = tf.train.MomentumOptimizer(learning_rate=1E-1, momentum=0.9)
```

The momentum is usually set to be 0.9.

The plot of training loss is showed as Figure 11. It seems both optimizers gives good result and oscillate starting at 2500 steps. In addition, the validation accuracy also shows the similar trend in Figure 12. Since the at the final step (5000 step) Momentum Optimizer has better accuracy (0.9891) then SGD (0.9881) and Momentum Optimizer takes about 20% less time to finish 5000 steps, it is reasonable to set Momentum Optimizer as our final optimizer.

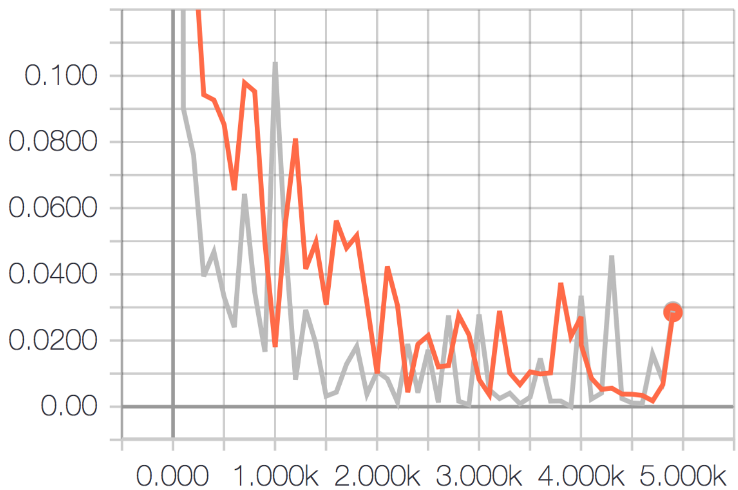

Figure 11: Loss function of training data using Momentum Optimizer (grey) and SGD (orange).

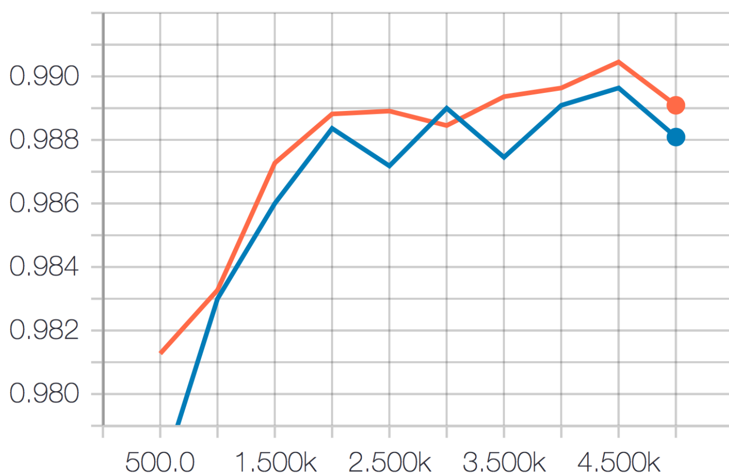

Figure 12: Accuracy of validation data using Momentum Optimizer (orange) and SGD (blue).

4 Final model and analysis

Based on previous experiments, the final model contins the following hyper-parameters:

- learning rate: `1E-1`
- Size of filter: 5×5
- Number of filters: 32 for first convolutional layer and 64 for second convolutional layer
- Number of convolutional and pooling layers: 2 and 2. The structure of the whole model is shown in Figure 3.
- The ways of paddings: `same`
- The choice of activation function: `ReLu`
- Optimizer: `Momentum Optimizer`

The source code containing these parameters is listed at the end of report. The loss of training process is presented in Figure 13. It shows the loss decrease dramatically at initial 100 steps and then tend to oscillate around a very low loss. Meanwhile, Figure 14 shows the accuracy of validation data throughout the training process and it clearly shows the model accuracy increases and converge fast to a very hight value. The finally validation accuracy at step 5000 is 0.9891. With this model, the final testing accuracy is a very satisfactory which is 0.9910.

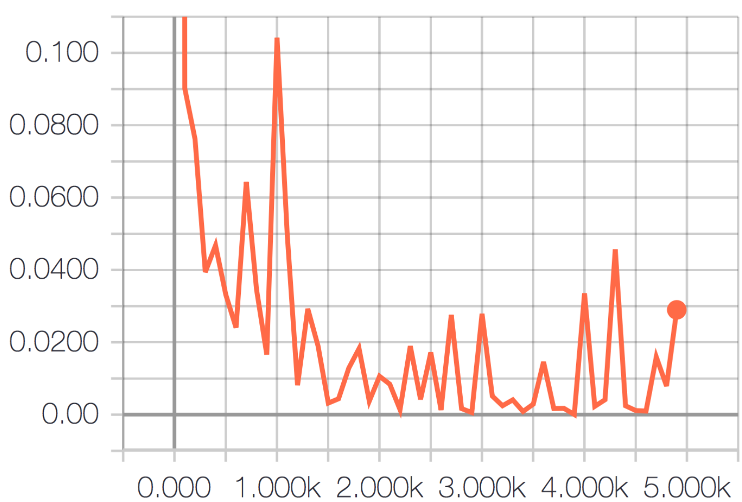

Figure 13: Loss of training.

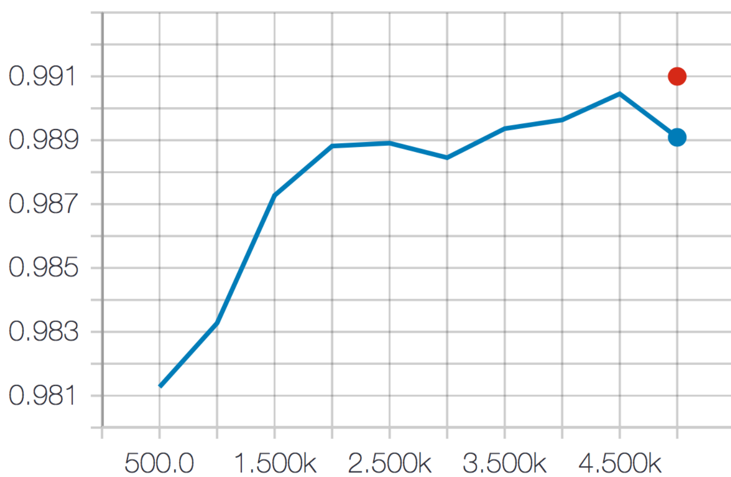

Figure 14: Accuracy of validation data (blue curve) and the final testing data (red dot).

5 Source codes

Here we only present the source code that gives the best result.

```
import tensorflow as tf
import numpy as np


# Parameters
# Default: list all the default variables that need to tune


# Adjust: put the parameter that need to be adjusted
learning_rate = 1E-1

# Save dir:
save_dir = 'tensorboard/optimizer/momentum_optimizer'




#This is to list all INFO
tf.logging.set_verbosity(tf.logging.INFO)


def cnn_model_fn(features, labels, mode):
  """Model function for CNN. This function is used to be called
     later in main function.
    features     :: the feature in array, size nnumber-by-784.
       Here 784 = 28*28 is the flattened representation of a hand
         written figure.
                    Features is a dict structure, {'x': <tf.
                       Tensor 'fifo_queue_DequeueUpTo:1' shape=(
                       nnumber, 784) dtype=float32>}
                    features['x'] corresponds to the acutal
                       tensor object
    labels       :: values from 0 to 9. Size nnumber-by-1. A
       tensor object
    mode         :: One of three modes: TRAIN, EVAL, PREDICT
  """

  # It is a function that used to train data.
```

```
# Input Layer
input_layer = tf.reshape(features["x"], [-1, 28, 28, 1])

# Convolutional Layer #1
conv1 = tf.layers.conv2d(
    inputs=input_layer,
    filters=32,
    kernel_size=[5, 5],
    padding="same",
    activation=tf.nn.relu, name='conv1')

# Pooling Layer #1
pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2, 2],
    strides=2, name = 'pool1')

# Convolutional Layer #2 and Pooling Layer #2
conv2 = tf.layers.conv2d(
    inputs=pool1,
    filters=64,
    kernel_size=[5, 5],
    padding="same",
    activation=tf.nn.relu, name = 'conv2')

pool2 = tf.layers.max_pooling2d(inputs=conv2, pool_size=[2, 2],
    strides=2, name='pool2')

# Dense Layer
pool2_flat = tf.reshape(pool2, [-1, 7 * 7 * 64])
dense = tf.layers.dense(inputs=pool2_flat, units=1024,
   activation=tf.nn.relu)
dropout = tf.layers.dropout(
    inputs=dense, rate=0.4, training=mode == tf.estimator.
       ModeKeys.TRAIN)

# Logits Layer
logits = tf.layers.dense(inputs=dropout, units=10)



predictions = {
    # Generate predictions (for PREDICT and EVAL mode)
    "classes": tf.argmax(input=logits, axis=1),#Calculate class
```

```
            on the fly
        # Add `softmax_tensor` to the graph. It is used for PREDICT
            and by the
        # `logging_hook`.
        "probabilities": tf.nn.softmax(logits, name="softmax_tensor
           ")
  }

  if mode == tf.estimator.ModeKeys.PREDICT:
    return tf.estimator.EstimatorSpec(mode=mode, predictions=
       predictions)

  # Calculate Loss (for both TRAIN and EVAL modes)
  loss = tf.losses.sparse_softmax_cross_entropy(labels=labels,
     logits=logits)
  tf.summary.scalar('loss', loss) #Write the loss into
     tensorboard


  # Configure the Training Op (for TRAIN mode)
  if mode == tf.estimator.ModeKeys.TRAIN:
    optimizer = tf.train.MomentumOptimizer(learning_rate=
       learning_rate, momentum=0.9)#potimized and learning rate
       can change
    train_op = optimizer.minimize(
        loss=loss,
        global_step=tf.train.get_global_step())
    return tf.estimator.EstimatorSpec(mode=mode, loss=loss,
       train_op=train_op)

  # Add evaluation metrics (for EVAL mode)
  eval_metric_ops = {"accuracy": tf.metrics.accuracy(labels=
     labels, predictions=predictions["classes"])}#predictions is
     a dict and calculate classes on the fly
  return tf.estimator.EstimatorSpec(mode=mode, loss=loss,
     eval_metric_ops=eval_metric_ops)


def main(aa):
    # Load training and eval data
    mnist = tf.contrib.learn.datasets.load_dataset("mnist")
```

```
p = int(len(mnist.train.images)*0.8) #Probably need to
    randomize
with tf.name_scope('train_data'):
    train_data = mnist.train.images[0:p] # Returns np.array
    train_labels = np.asarray(mnist.train.labels, dtype=np.
        int32)[0:p]

with tf.name_scope('valid_data'):
    valid_data = mnist.train.images[p:]  # Returns np.array
    valid_labels = np.asarray(mnist.train.labels, dtype=np.
        int32)[p:]

with tf.name_scope('test_data'):
    eval_data = mnist.test.images # Returns np.array
    eval_labels = np.asarray(mnist.test.labels, dtype=np.
        int32)


# Create the Estimator
mnist_classifier = tf.estimator.Estimator(model_fn=
    cnn_model_fn, model_dir=save_dir)
# Set up logging for predictions
tensors_to_log = {"probabilities": "softmax_tensor"}
logging_hook = tf.train.LoggingTensorHook(tensors=
    tensors_to_log, every_n_iter=50)

# # Merge all summaries
# merged_summary = tf.summary.merge_all()
# writer = tf.summary.FileWriter("../../tensorboard/project3/
    pj3-1")
# writer.add_graph(sess.graph)


# Train the model
train_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": train_data},#First input, convert the numpy array
        data into a dict structure
    y=train_labels,
    batch_size= 200,
    num_epochs=None, #Will run forever
    shuffle=True)
```

```
    valid_input_fn = tf.estimator.inputs.numpy_input_fn(
        x={"x": valid_data},
        y=valid_labels,
        num_epochs=1,
        shuffle=False) # Boolean, if True shuffles the queue.
            Avoid shuffle at prediction time.

    experiment = tf.contrib.learn.Experiment(
        mnist_classifier,
        train_input_fn,
        valid_input_fn,
        train_steps = 5000, #This is the step for gradient?
        eval_steps = None,
        train_steps_per_iteration = 500) #Every this step, save
            to ckpt, and evaluate accuracy
    experiment.continuous_train_and_eval()
    # The result of this step is a trained mnist_classifer

    #The rest come from tutorial
    # mnist_classifier.train(
    #       input_fn=train_input_fn,
    #       steps=1,
    #       hooks=[logging_hook])


    # Evaluate the model and print results
    eval_input_fn = tf.estimator.inputs.numpy_input_fn(
        x={"x": eval_data},
        y=eval_labels,
        num_epochs=1,
        shuffle=False)

    eval_results = mnist_classifier.evaluate(input_fn=
       eval_input_fn)
    print(eval_results)




if __name__ == '__main__':
    """Runs whole fitting program automatically"""
    import time
```

```
start_time = time.time()
tf.app.run()
print("--- %s seconds ---" % (time.time() - start_time))
```