MHPF_PT_week1_DavidMezey_LukasDippold

April 25, 2018

1 MHBF - PT - Week 1

Created by:
David Mezey (ST.ID:0396101, BCCN)
Lukas Dippold (ST.ID:385829,TUB)

1.1 Creating Linear Model fn

```
In [1]: from __future__ import absolute_import
        from __future__ import division
        from __future__ import print_function

        import numpy as np
        import tensorflow as tf
```

I have turned off the logging for the jupyter notebook export. I have turned it on only for the CNN, because that takes longer to train.

```
In [2]: tf.logging.set_verbosity(tf.logging.ERROR)
        #tf.logging.set_verbosity(tf.logging.INFO)
```

```
In [3]: def linear_model_fn(features, labels, mode):
          """Model function for Linear Classifier, with 1 dense layer."""
          # Input Layer
          input_layer = tf.reshape(features["x"], [-1, 784])

          #Logits Layer = Only dense layer
          logits = tf.layers.dense(inputs=input_layer, units=10)

          predictions = {
              # Generate predictions (for PREDICT and EVAL mode)
              "classes": tf.argmax(input=logits, axis=1),
              # Add `softmax_tensor` to the graph. It is used for PREDICT and by the
              # `logging_hook`.
              # the softmax layer just normalizes the logits layer to a probability vector
              "probabilities": tf.nn.softmax(logits, name="softmax_tensor")
          }
          #PREDICT MODE
          if mode == tf.estimator.ModeKeys.PREDICT:
            return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)

          # Calculate Loss (for both TRAIN and EVAL modes)
          # highly optimized function, high level of abstraction, very stable
          # a softmax layer with the cross entropy would be less stable
          loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)

          #TRAIN MODE
          if mode == tf.estimator.ModeKeys.TRAIN:
```

```
      optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.5)
      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"])}
    return tf.estimator.EstimatorSpec(
        mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)
```

1.2 Extracting MNIST data, creating estimator

```
In [4]: # Load training and eval data
        mnist = tf.contrib.learn.datasets.load_dataset("mnist")
        train_data = mnist.train.images  # Returns np.array
        train_labels = np.asarray(mnist.train.labels, dtype=np.int32)
        eval_data = mnist.test.images  # Returns np.array
        eval_labels = np.asarray(mnist.test.labels, dtype=np.int32)

        # Create the Estimator
        # Change the model dir to avoid conflict with previous model graps
        mnist_classifier = tf.estimator.Estimator(
          model_fn=linear_model_fn,
        model_dir="/tmp/mnist_linear_classifier")#model_dir="/tmp/mnist_convnet_model")

        # Set up logging for predictions
        # Log the values in the "Softmax" tensor with label "probabilities"
        tensors_to_log = {"probabilities": "softmax_tensor"}
        logging_hook = tf.train.LoggingTensorHook(
          tensors=tensors_to_log, every_n_iter=100)
```

```
Extracting MNIST-data\train-images-idx3-ubyte.gz
Extracting MNIST-data\train-labels-idx1-ubyte.gz
Extracting MNIST-data\t10k-images-idx3-ubyte.gz
Extracting MNIST-data\t10k-labels-idx1-ubyte.gz
```

1.3 Training the model

```
In [5]: # Train the model
        tf.reset_default_graph()
        train_input_fn = tf.estimator.inputs.numpy_input_fn(
            x={"x": train_data},
            y=train_labels,
            batch_size=100,
            num_epochs=None,
            shuffle=True)
        mnist_classifier.train(
            input_fn=train_input_fn,
            steps=1000) #1000 steps
            #hooks=[logging_hook])
```

```
Out[5]: <tensorflow.python.estimator.estimator.Estimator at 0x2a8be828>
```

1.4 Evaluating the model with evaluating set, Reporting accuracy

```
In [6]: # Evaluate the model and print results
        eval_input_fn = tf.estimator.inputs.numpy_input_fn(
          x={"x": eval_data},
          y=eval_labels,
```

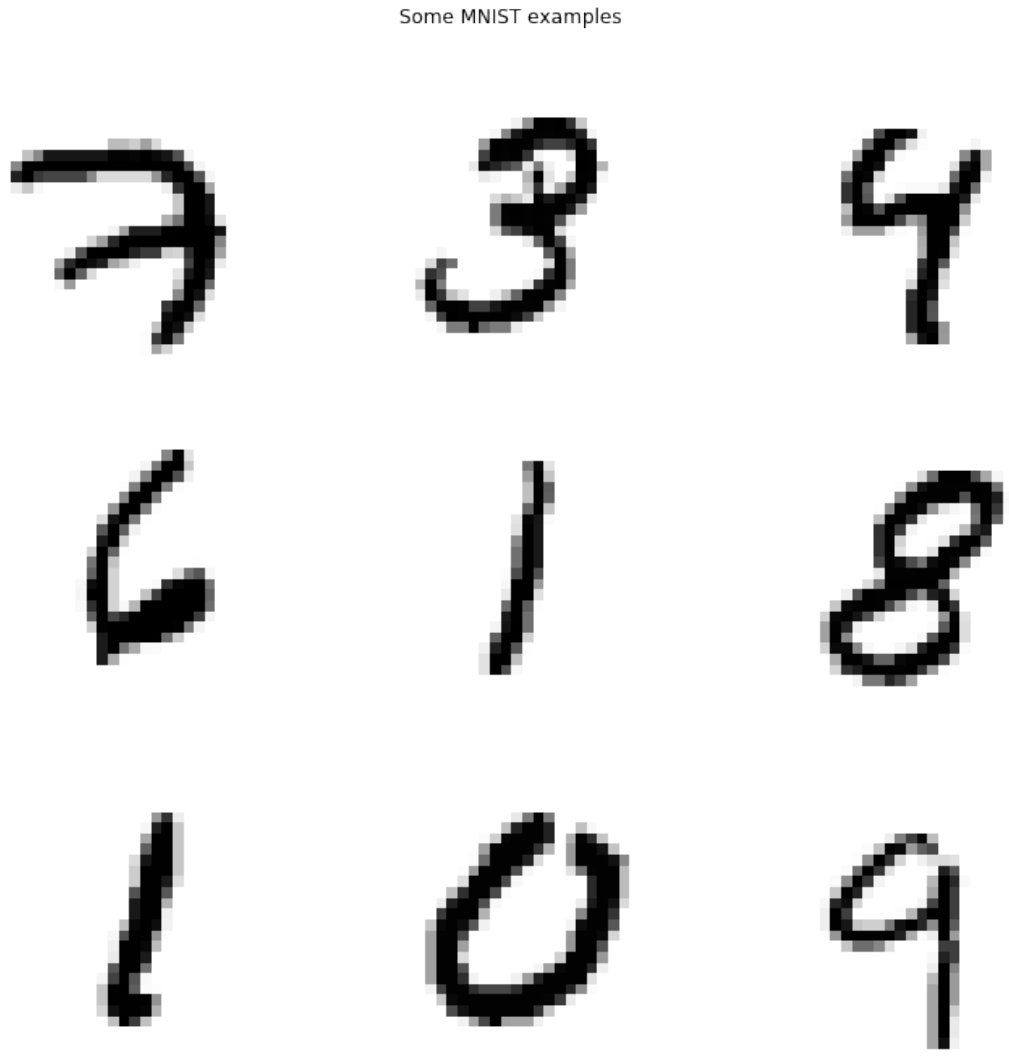
```
          num_epochs=1,
          shuffle=False)
        eval_results = mnist_classifier.evaluate(input_fn=eval_input_fn)
```

```
In [7]: print('Evaluation results : ',eval_results)
        print('Accuracy : ',eval_results['accuracy']*100, '%')
```

```
Evaluation results :  {'accuracy': 0.92119998, 'loss': 0.27563292, 'global_step':
9000}
Accuracy :  92.1199977398 %
```

1.5 Plotting a few MNIST examples from the original dataset

```
In [8]: import matplotlib.pyplot as plt
        %matplotlib inline

        fig, ax = plt.subplots(3,3,figsize=[10,10])
        fig.suptitle('Some MNIST examples', fontsize=12)
        fig.tight_layout(rect=[0, 0.03, 1, 0.95])
        axv = ax.reshape(9)
        for i, ax_ in enumerate(axv):
            plt.axes(ax_)
            plt.axis('off')
            plt.imshow(mnist.train.images[i].reshape(28,28), cmap='Greys')
```



1.6 Studying the parameters of the model:

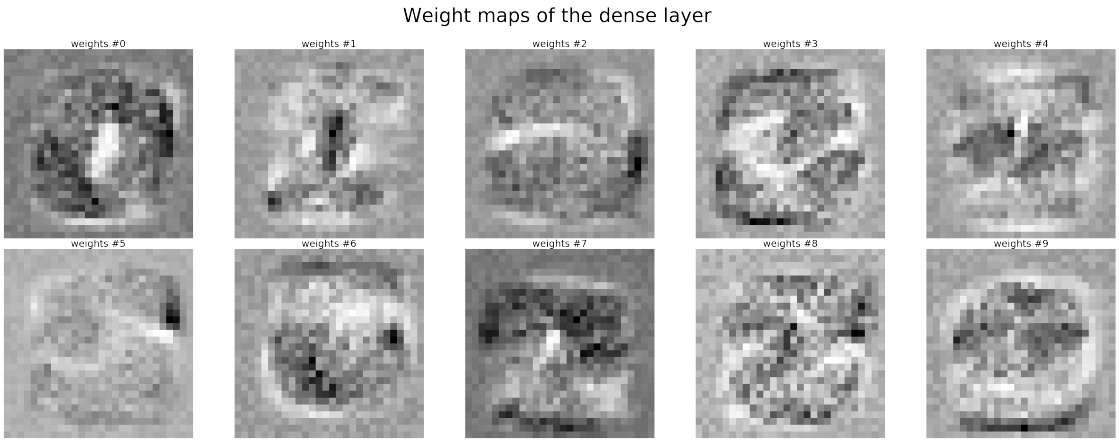
```
In [9]: print('The model has ', len(mnist_classifier.get_variable_names()), ' variables.\n')
        print('With variable names: {}'.format(mnist_classifier.get_variable_names()),'\n')
```

```
The model has  3  variables.

With variable names: ['dense/bias', 'dense/kernel', 'global_step']
```

1.7 Plotting the dense layer's activation

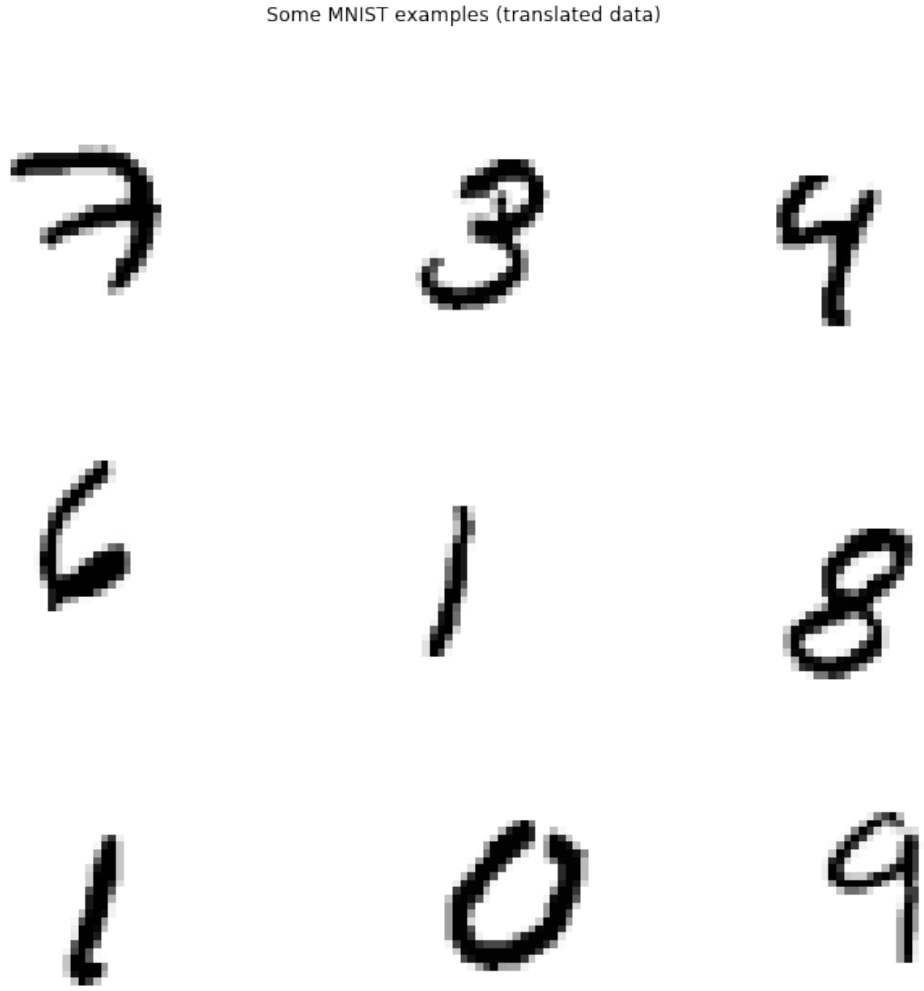
```
In [10]: act_array = mnist_classifier.get_variable_value(name='dense/kernel')

         fig, ax = plt.subplots(2,5,figsize=[50,20])
```

```
fig.suptitle('Weight maps of the dense layer', fontsize=60)
fig.tight_layout(rect=[0, 0.03, 1, 0.90])
axv = ax.reshape(10)
for axi, ax_ in enumerate(axv):
    plt.axes(ax_)
    plt.axis('off')
    plt.title('weights #{}'.format(axi), fontsize=30)
    plt.imshow(act_array[:,axi].reshape(28,28), cmap='Greys')
```



1.8 Translating the MNIST samples to 40x40

```
In [11]: def translate_mnist(mnist, n_trans=40):
             """
             Takes a DataSets object loaded using
             tensorflow.examples.tutorials.mnist.input_data.read_data_sets
             and replaces the 28x28 images with images of size 40x40.

             :param mnist: DataSets
                           MNIST data loaded using TensorFlow
             :return: None
             """
             for name in ('train', 'validation', 'test'):
                 images = getattr(mnist, name).images.reshape((-1, 28, 28))
                 n_ims = images.shape[0]
                 images_ = np.zeros((n_ims, n_trans, n_trans), dtype=images.dtype)
                 trans = np.random.choice(n_trans-28, size=(n_ims, 2))
                 for i in range(n_ims):
                     x, y = trans[i, :]
                     images_[i, x: x+28, y: y+28] = images[i, ...]
                 getattr(mnist, name)._images = \
                     images_.reshape((n_ims,np.product(images_.shape[1:])))

In [12]: translate_mnist(mnist)

In [13]: fig, ax = plt.subplots(3,3,figsize=[10,10])
         fig.suptitle('Some MNIST examples (translated data)', fontsize=12)
         fig.tight_layout(rect=[0, 0.03, 1, 0.95])
         axv = ax.reshape(9)
         for i, ax_ in enumerate(axv):
             plt.axes(ax_)
             plt.axis('off')
             plt.imshow(mnist.train.images[i].reshape(40,40),cmap='Greys')
```



1.9 Creating new model fn for translated images

```
In [14]: def linear_model_fn_translated(features, labels, mode):
             """Model function for Linear Classifier, with 1 dense layer."""

             input_layer = tf.reshape(features["x"], [-1, 1600])

             logits = tf.layers.dense(inputs=input_layer, units=10)

             predictions = {
                 "classes": tf.argmax(input=logits, axis=1),
                 "probabilities": tf.nn.softmax(logits, name="softmax_tensor")
             }

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

             loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)
```

```
  if mode == tf.estimator.ModeKeys.TRAIN:
    optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.5)
    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)

  eval_metric_ops = {
      "accuracy": tf.metrics.accuracy(
          labels=labels, predictions=predictions["classes"])}
  return tf.estimator.EstimatorSpec(
      mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)
```

1.10 Creating new estimator

```
In [15]: #loading translated data into nparrays
         train_data_translated = mnist.train.images  # Returns np.array
         train_labels_translated = np.asarray(mnist.train.labels, dtype=np.int32)
         eval_data_translated = mnist.test.images  # Returns np.array
         eval_labels_translated = np.asarray(mnist.test.labels, dtype=np.int32)

         # Create the translated estimator Estimator
         # Change the model dir to avoid conflict with previous model graps
         mnist_classifier_translated = tf.estimator.Estimator(
           model_fn=linear_model_fn_translated, model_dir="/tmp/mnist_linear_classifier_translate
         d")#model_dir="/tmp/mnist_convnet_model")

         # Set up logging for predictions
         # Log the values in the "Softmax" tensor with label "probabilities"
         tensors_to_log = {"probabilities": "softmax_tensor"}
         logging_hook = tf.train.LoggingTensorHook(
           tensors=tensors_to_log, every_n_iter=100)
```

1.11 Training the new model with translated data

```
In [16]: # Train the model
         tf.reset_default_graph()
         train_input_fn_translated = tf.estimator.inputs.numpy_input_fn(
             x={"x": train_data_translated},
             y=train_labels_translated,
             batch_size=100,
             num_epochs=None,
             shuffle=True)
         mnist_classifier_translated.train(
             input_fn=train_input_fn_translated,
             steps=1000) #1000 steps
             #hooks=[logging_hook])
```

```
Out[16]: <tensorflow.python.estimator.estimator.Estimator at 0x2f36f6a0>
```

1.12 Evaluating the model with translated evaluating set, Reporting accuracy

```
In [17]: # Evaluate the model and print results
         eval_input_fn = tf.estimator.inputs.numpy_input_fn(
           x={"x": eval_data_translated},
           y=eval_labels_translated,
           num_epochs=1,
           shuffle=False)
         eval_results = mnist_classifier_translated.evaluate(input_fn=eval_input_fn)
         print('Evaluation results : ',eval_results)
         print('Accuracy : ',eval_results['accuracy']*100, '%')
```

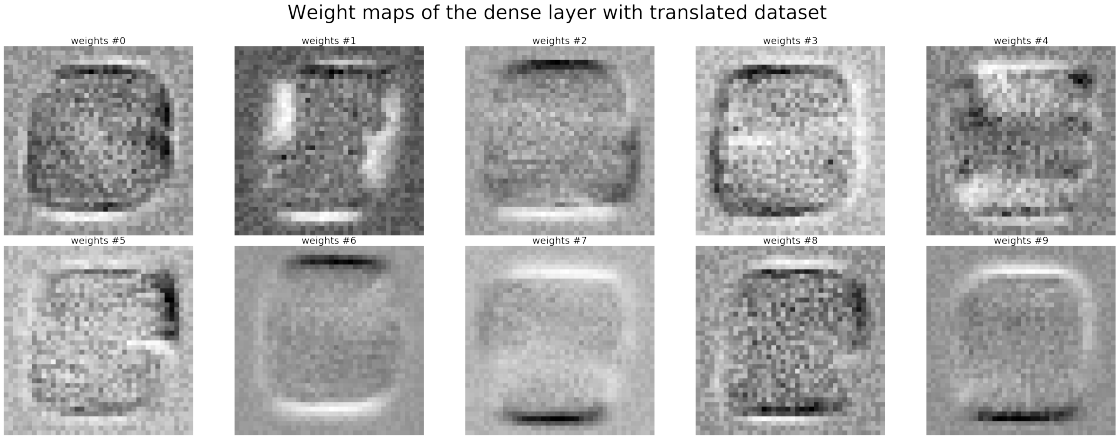
```
Evaluation results :  {'accuracy': 0.40920001, 'loss': 1.7240888, 'global_step': 5000}
Accuracy :  40.9200012684 %
```

```
In [18]: print('The new model has ', len(mnist_classifier_translated.get_variable_names()), '
         variables.\n')
         print('With variable names:
         {}'.format(mnist_classifier_translated.get_variable_names()),'\n')
```

```
The new model has  3  variables.

With variable names: ['dense/bias', 'dense/kernel', 'global_step']
```

1.13 Plotting the dense layer's activation

```
In [19]: act_array = mnist_classifier_translated.get_variable_value(name='dense/kernel')

         fig, ax = plt.subplots(2,5,figsize=[50,20])
         fig.suptitle('Weight maps of the dense layer with translated dataset', fontsize=60)
         fig.tight_layout(rect=[0, 0.03, 1, 0.90])
         axv = ax.reshape(10)
         for axi, ax_ in enumerate(axv):
             plt.axes(ax_)
             plt.axis('off')
             plt.title('weights #{}'.format(axi), fontsize=30)
             plt.imshow(act_array[:,axi].reshape(40,40), cmap='Greys')
```



2 Adding Hidden Layer

```
In [20]: def mlp_model_fn(features, labels, mode):
             """Model function for Linear Classifier, with 1 hidden layer and 1 dense layer."""
             # Input Layer
             input_layer = tf.reshape(features["x"], [-1, 1600])

             # Hidden Layer
             hidden = tf.layers.dense(inputs=input_layer, units=500, activation=tf.nn.relu)

             #Logits Layer = Only dense layer
```

```
  logits = tf.layers.dense(inputs=hidden, units=10)

  predictions = {
      # Generate predictions (for PREDICT and EVAL mode)
      "classes": tf.argmax(input=logits, axis=1),
      # Add `softmax_tensor` to the graph. It is used for PREDICT and by the
      # `logging_hook`.
      # the softmax layer just normalizes the logits layer to a probability vector
      "probabilities": tf.nn.softmax(logits, name="softmax_tensor")
  }
  #PREDICT MODE
  if mode == tf.estimator.ModeKeys.PREDICT:
    return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)

  # Calculate Loss (for both TRAIN and EVAL modes)
  # highly optimized function, high level of abstraction, very stable
  # a softmax layer with the cross entropy would be less stable
  loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)

  #TRAIN MODE
  if mode == tf.estimator.ModeKeys.TRAIN:
    optimizer = tf.train.AdamOptimizer(learning_rate=10**-4)
    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"])}
  return tf.estimator.EstimatorSpec(
      mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)
```

2.1 Creating estimator for MLP

```
In [21]: # Create the MLP estimator
         # Change the model dir to avoid conflict with previous model graps
         mnist_classifier_mlp = tf.estimator.Estimator(
           model_fn=mlp_model_fn,
         model_dir="/tmp/mnist_mlp_classifier")#model_dir="/tmp/mnist_convnet_model")

         # Set up logging for predictions
         # Log the values in the "Softmax" tensor with label "probabilities"
         tensors_to_log = {"probabilities": "softmax_tensor"}
         logging_hook = tf.train.LoggingTensorHook(
           tensors=tensors_to_log, every_n_iter=100)
```

2.2 Training the MLP model with translated data

```
In [22]: # Train the model
         tf.reset_default_graph()
         train_input_fn_mlp = tf.estimator.inputs.numpy_input_fn(
             x={"x": train_data_translated},
             y=train_labels_translated,
             batch_size=50,
             num_epochs=None,
             shuffle=True)
         mnist_classifier_mlp.train(
             input_fn=train_input_fn_mlp,
             steps=10000) #1000 steps
             #hooks=[logging_hook])
```

```
Out[22]: <tensorflow.python.estimator.estimator.Estimator at 0x2bc14198>
```

2.3 Evaluating the MLP model with translated evaluating set, Reporting accuracy

```
In [23]: # Evaluate the model and print results
         eval_input_fn = tf.estimator.inputs.numpy_input_fn(
           x={"x": eval_data_translated},
           y=eval_labels_translated,
           num_epochs=1,
           shuffle=False)
         eval_results = mnist_classifier_mlp.evaluate(input_fn=eval_input_fn)
         print('Evaluation results : ',eval_results)
         print('Accuracy : ',eval_results['accuracy']*100, '%')
```

```
Evaluation results :  {'accuracy': 0.95380002, 'loss': 0.15558781, 'global_step':
40000}
Accuracy :  95.3800022602 %
```

```
In [24]: print('The new model has ', len(mnist_classifier_mlp.get_variable_names()), '
         variables.\n')
         print('With variable names:
         \n\n{}'.format(mnist_classifier_mlp.get_variable_names()),'\n')
         #printing these variables would not be informative
```

```
The new model has  15  variables.

With variable names:

['beta1_power', 'beta2_power', 'dense/bias', 'dense/bias/Adam', 'dense/bias/Adam_1',
'dense/kernel', 'dense/kernel/Adam', 'dense/kernel/Adam_1', 'dense_1/bias',
'dense_1/bias/Adam', 'dense_1/bias/Adam_1', 'dense_1/kernel', 'dense_1/kernel/Adam',
'dense_1/kernel/Adam_1', 'global_step']
```

3 Convolutional network (with 1 convolutional layer)

```
In [25]: tf.logging.set_verbosity(tf.logging.INFO)

         def cnn_model_fn(features, labels, mode):
           """Model function for CNN."""
           # Input Layer
           # Reshape X to 4-D tensor: [batch_size, width, height, channels]
           # MNIST images are 28x28 pixels, and have one color channel
           input_layer = tf.reshape(features["x"], [-1, 40, 40, 1])

           # Convolutional Layer #1
           # Computes 32 features using a 5x5 filter with ReLU activation.
           # Padding is added to preserve width and height.
           # Input Tensor Shape: [batch_size, 40, 40, 1]
           # Output Tensor Shape: [batch_size, 40, 40, 32]
           conv1 = tf.layers.conv2d(
               inputs=input_layer,
               filters=32,
               kernel_size=[5, 5],
               padding="same",
               activation=tf.nn.relu)

           # Pooling Layer #1
           # First max pooling layer with a 2x2 filter and stride of 2
           # Input Tensor Shape: [batch_size, 40, 40, 32]
           # Output Tensor Shape: [batch_size, 20, 20, 32]
           pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2, 2], strides=2)
```

```
    '''# Convolutional Layer #2
    # Computes 64 features using a 5x5 filter.
    # Padding is added to preserve width and height.
    # Input Tensor Shape: [batch_size, 14, 14, 32]
    # Output Tensor Shape: [batch_size, 14, 14, 64]
    conv2 = tf.layers.conv2d(
        inputs=pool1,
        filters=64,
        kernel_size=[5, 5],
        padding="same",
        activation=tf.nn.relu)

    # Pooling Layer #2
    # Second max pooling layer with a 2x2 filter and stride of 2
    # Input Tensor Shape: [batch_size, 14, 14, 64]
    # Output Tensor Shape: [batch_size, 7, 7, 64]
    pool2 = tf.layers.max_pooling2d(inputs=conv2, pool_size=[2, 2], strides=2)'''

    # Flatten tensor into a batch of vectors
    # Input Tensor Shape: [batch_size, 7, 7, 64]
    # Output Tensor Shape: [batch_size, 7 * 7 * 64]
    pool2_flat = tf.reshape(pool1, [-1, 20 * 20 * 32])

    # Dense Layer
    # Densely connected layer with 1024 neurons
    # Input Tensor Shape: [batch_size, 7 * 7 * 64]
    # Output Tensor Shape: [batch_size, 1024]
    dense = tf.layers.dense(inputs=pool2_flat, units=1024, activation=tf.nn.relu)

    # Add dropout operation; 0.6 probability that element will be kept
    dropout = tf.layers.dropout(
        inputs=dense, rate=0.4, training=mode == tf.estimator.ModeKeys.TRAIN)

    # Logits layer
    # Input Tensor Shape: [batch_size, 1024]
    # Output Tensor Shape: [batch_size, 10]
    logits = tf.layers.dense(inputs=dropout, units=10)

    predictions = {
        # Generate predictions (for PREDICT and EVAL mode)
        "classes": tf.argmax(input=logits, axis=1),
        # Add `softmax_tensor` to the graph. It is used for PREDICT and by the
        # `logging_hook`.
        "probabilities": tf.nn.softmax(logits, name="softmax_tensor"),
        "activations": conv1
    }
    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)

    # Configure the Training Op (for TRAIN mode)
    if mode == tf.estimator.ModeKeys.TRAIN:
      optimizer = tf.train.AdamOptimizer(learning_rate=0.0001)
      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"])}
    return tf.estimator.EstimatorSpec(
        mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)
```

In [26]:
```
# Create the Estimator
```

```
        mnist_classifier_cnn = tf.estimator.Estimator(
          model_fn=cnn_model_fn, model_dir="/tmp/mnist_convnet_model_cnn")

        # Set up logging for predictions
        # Log the values in the "Softmax" tensor with label "probabilities"
        tensors_to_log = {"probabilities": "softmax_tensor"}
        logging_hook = tf.train.LoggingTensorHook(
          tensors=tensors_to_log, every_n_iter=50)
```

```
INFO:tensorflow:Using default config.
INFO:tensorflow:Using config: {'_model_dir': '/tmp/mnist_convnet_model_cnn',
'_tf_random_seed': None, '_save_summary_steps': 100, '_save_checkpoints_steps': None,
'_save_checkpoints_secs': 600, '_session_config': None, '_keep_checkpoint_max': 5,
'_keep_checkpoint_every_n_hours': 10000, '_log_step_count_steps': 100, '_service':
None, '_cluster_spec': <tensorflow.python.training.server_lib.ClusterSpec object at
0x000000002FA2AB70>, '_task_type': 'worker', '_task_id': 0, '_master': '',
'_is_chief': True, '_num_ps_replicas': 0, '_num_worker_replicas': 1}
```

In [27]:
```
        # Train the model
        # Please note that the model was trained with more than 5000 steps not 100
        tf.reset_default_graph()
        tf.logging.set_verbosity(tf.logging.INFO)
        train_input_fn_cnn = tf.estimator.inputs.numpy_input_fn(
          x={"x": train_data_translated},
          y=train_labels_translated,
          batch_size=50,
          num_epochs=None,
          shuffle=True)
        mnist_classifier_cnn.train(
          input_fn=train_input_fn_cnn,
          steps=100)
          #hooks=[logging_hook])
```

```
INFO:tensorflow:Create CheckpointSaverHook.
INFO:tensorflow:Restoring parameters from /tmp/mnist_convnet_model_cnn\model.ckpt-5202
INFO:tensorflow:Saving checkpoints for 5203 into
/tmp/mnist_convnet_model_cnn\model.ckpt.
INFO:tensorflow:loss = 0.218678, step = 5203
INFO:tensorflow:Saving checkpoints for 5302 into
/tmp/mnist_convnet_model_cnn\model.ckpt.
INFO:tensorflow:Loss for final step: 0.180008.
```

Out[27]: <tensorflow.python.estimator.estimator.Estimator at 0x2fa2afd0>

In [28]:
```
        # Evaluate the model and print results
        eval_input_fn = tf.estimator.inputs.numpy_input_fn(
          x={"x": eval_data_translated},
          y=eval_labels_translated,
          num_epochs=1,
          shuffle=False)
        eval_results = mnist_classifier_cnn.evaluate(input_fn=eval_input_fn)
        print('Evaluation results : ',eval_results)
        print('Accuracy : ',eval_results['accuracy']*100, '%')
```

```
INFO:tensorflow:Starting evaluation at 2018-04-25-08:17:55
INFO:tensorflow:Restoring parameters from /tmp/mnist_convnet_model_cnn\model.ckpt-5302
INFO:tensorflow:Finished evaluation at 2018-04-25-08:18:03
INFO:tensorflow:Saving dict for global step 5302: accuracy = 0.9651, global_step =
5302, loss = 0.11732
Evaluation results :  {'accuracy': 0.96509999, 'loss': 0.11732049, 'global_step':
5302}
Accuracy :  96.5099990368 %
```

```
In [29]: print('The new model has ', len(mnist_classifier_cnn.get_variable_names()), '
         variables.\n')
         print('With variable names:
         \n\n{}'.format(mnist_classifier_cnn.get_variable_names()),'\n')
         #printing these variables would not be informative
```

```
The new model has  21  variables.

With variable names:

['beta1_power', 'beta2_power', 'conv2d/bias', 'conv2d/bias/Adam',
'conv2d/bias/Adam_1', 'conv2d/kernel', 'conv2d/kernel/Adam', 'conv2d/kernel/Adam_1',
'dense/bias', 'dense/bias/Adam', 'dense/bias/Adam_1', 'dense/kernel',
'dense/kernel/Adam', 'dense/kernel/Adam_1', 'dense_1/bias', 'dense_1/bias/Adam',
'dense_1/bias/Adam_1', 'dense_1/kernel', 'dense_1/kernel/Adam',
'dense_1/kernel/Adam_1', 'global_step']
```

3.1 Prediction

```
In [30]: predict_input_fn_cnn = tf.estimator.inputs.numpy_input_fn(
           x={"x": train_data_translated},
           y=train_labels_translated,
           batch_size=50,
           num_epochs=None,
           shuffle=False)

         np.shape(train_data_translated)

         allpredictions = mnist_classifier_cnn.predict(input_fn=predict_input_fn_cnn)
```

3.2 Saving the activation functions of the convlayer

```
In [31]: act_list=[]
         for i,pred in enumerate(allpredictions):
             if i<3:
                 act_list.append(pred['activations'])
             else:
                 break
```

```
INFO:tensorflow:Restoring parameters from /tmp/mnist_convnet_model_cnn\model.ckpt-5302
```

3.3 Visualizing the filters

```
In [32]: plt.figure()
         fig, ax = plt.subplots(6,6,figsize=[13,13])
         fig.suptitle('Filters of the convolutional layer', fontsize=15)
         fig.tight_layout(rect=[0, 0.03, 1, 0.94])
         axv = ax.reshape(36)
         for axi,ax_ in enumerate(axv):
             plt.axes(ax_)
             plt.axis('off')
             if axi<32:
                 plt.imshow(act_list[1][:,:,axi], cmap='Greys')
                 plt.title('Filter #{}'.format(axi+1))
             if axi==35:
                 plt.imshow(train_data_translated[1].reshape(40,40), cmap='Greys')
                 plt.title('Original Input')
```

```
<matplotlib.figure.Figure at 0x2f2a8c88>
```

4 Discussion:

Because of the special network architecture, the convolutional network will perform better on translated images. This is because:

- The convolutional layer will keep the position information of the original image. If it is translated, then the image on the conv layer's filters will be translated as well. These filters then will be fed in to the pooling layer.
- In the pooling layer the layer units will be activated according to the values in their receptive fields. (e.g. maximum pooling). This layer gives the translation invariance (for specific

amount of translation), because if the translated image is still in the receptive field of the units properly, the units will behave just like the translation has never happened.

Alltogether, the main difference between the MLP and the CNN is that in their basic architecture, and therefore in their ability or lack of translation invariance. Convolutional networks are created for input images, with correlated neighbouring pixels and therefore they will have a better performance in tasks like the MNIST classification. It is important to note, however, that the training time can vary vastly and one may need an intense computational power to create and handle large CNN's.

4.1 Optional:

As far as I know there are some approaches to handle the training of a network. Such approach can be "Early Stopping", when the training is forced to stop once the loss function doesn't change too much (This definition may seem wishy washy, however these criterions are always tailored to the specific aim and task).

My approach would be somewhat like "Early Stopping". I would investigate the accuracy of the network during training epochs and would stop the training if the network reaches a preferred evaluation accuracy. This, on the other hand may never happen and therefore it is crucial to build in a mechanism which handles this case. One could think about a system clock which will shut down the training after a given training time/steps, and reset the weights of the network to those with the highest performance. (The feasability of this approach is not guaranteed, but would be a nice try)