# Deep Learning Assignment 3

March 7, 2019

**Professor: Francois Rivest**

**Name: Daniel Lee (20125992)**

## 1 Build a Convolutional Neural Network using Estimators

Explanation:

1. LINE 1: The imports are required packages to build the CNN.
2. LINE 6: tf.logging.set_verbosity sets the threshold for the type of log message that will be used.

```
In [0]: from __future__ import absolute_import, division, print_function

        import tensorflow as tf
        import numpy as np

        tf.logging.set_verbosity(tf.logging.INFO)
```

Explanation:

1. LINE 4: The input layer is created with the MNIST dataset, which is reshaped into a tensor with the dimensionss: -1 (batch size) x 28 (height) x 28 (width) x 1 (channel). The -1 places each 28 x 28 image into a column vector as a row. The 1 represents the number of channel layers used, which is just 1 layer in this case because we only need the colour black for the MNIST dataset.
2. LINE 7: The first convolutional layer applies 32 filters that are of 5 x 5 kernel size. The filter contains randomly initialized weights, which is multiplied on each sub-region of an image. In addition, the parameter, padding="same" is used to pad the sub-regions with a low value (e.g. 0) where the filter exceeds the right or bottom boundary of an image. The following process is done 32 times (each filter) for each image in the batch: An overlapping filter of 5 x 5 kernel size is applied on each subregion of an image, which produces a single output value. The ReLu activation function is then applied to the output of each sub-region to add non-linearity, then is stored in a new tensor (i.e., feature map). Thus, a tensor of batch size x 28 x 28 x 32 dimension is created.

3. LINE 15: Max pooling is then used to reduce the dimensionality or downsample the non-linearities of the 28 x 28 feature maps from convolutional layer 1. The following process is done 32 times (each filter) for each image in the batch: A non-overlapping filter of 2 x 2 kernel size with a stride of 2 is applied on each sub-region of the feature map, so that the filter only extracts the maximum value in each sub-region, which are placed in a new tensor. Thus, this creates a tensor of batch size x 14 x 14 x 32 dimension.

4. LINE 18: The second convolutional layer applies 64 filters that are of 5 x 5 kernel size. The following process is done 64 times (each filter) for each image in the batch: An overlapping 5 x 5 kernel size is applied on each of sub-region of the tensor, which produces a single output value. The ReLu activation function is then applied to the output of each sub-region to add non-linearity, then is stored in a new tensor (i.e., feature map). Thus, a tensor of batch size x 14 x 14 x 64 dimension is created.

5. LINE 24: Afterwards, max pooling is used again with a filter of 2 x 2 kernel size and a stride of 2 to reduce the dimensionality of the 14 x 14 feature maps. The following process is done 64 times (each filter) for each image in the batch: A non-overlapping filter of 2 x 2 kernel size with a stride of 2 is applied on each sub-region of the feature map, so that the filter only extracts the maximum value in each sub-region, which are placed in a new tensor. Thus, this creates a tensor of batch size x 7 x 7 x 64 dimension.

6. LINE 27: The tensor from the second pooling layer is flattened into a 1-dimensional tensor where the length is now batch size x 7 x 7 x 64 because the dense or fully connected layer takes a 1-dimensional tensor as input.

7. LINE 28: The dense layer uses 1,024 nodes (i.e neurons), where each node is densely connected to each node in the dropout layer. Each node will use the 1-dimensional tensor of length batch size x 7 x 7 x 64, and then apply the ReLu activation function, which then produces a tensor of batch size x 1,024 dimension.

8. LINE 29: A dropout layer (i.e., a form of regularization) is used to prevent overfitting when training by shutting off certain nodes based on the given probability. Subsequent nodes are able to find patterns from different parts of the data by looking at fewer samples. In this code block, the 0.4 represents the probability of dropping or shutting down a node in the layer. Dropout is applied on each node of the dense layer.

9. LINE 33: The outputs from the dropout layer then become the inputs for the nodes of the logit layer. In our case, the logit layer has 10 nodes or neurons to match the number of target classes, which in our case are the digits from 0 to 9. The logit function is applied on each output from the dropout layer (batch size x 1,024 tensor, and then applies a linear activation function to produce raw values in a batch size x 10 tensor.

10. LINE 35: Argmax is then used to get the index of the element with the highest value in each row of the batch size x 10 tensor that was output from the logit layer. The softmax activation function then creates a probability distribution for the 10 values (sums up to 1) from the argmax function for each image in the batch. The probability distribution in the batch size x 10 tensor represents the predictions for each target class for each image.

11. LINE 43: IF the mode of the model or estimator is being used to predict. If so, an EstimatorSpec model is returned with the mode and prediction values as arguments.

12. LINE 47: The softmax cross entropy loss function is used to calculate the loss between the target labels and the logit values.

13. LINE 50: IF the mode of the model or estimator is being used to train then it uses a stochastic gradient descent optimizer to shuffle the samples with an alpha or learning rate parameter of 0.001, which optimally locates the the minimum on the cross entropy loss function. The optimizer then uses the minimize function to compute the gradients of the loss with respect to the variable (e.g., weights) that were stored on the tensor graph. Then it uses the computed gradients to update the variables such as the weights in the tensor graph. Afterwards, the EstimatorSpec model is returned with the mode, loss, and the training step from the minimize function as arguments.

14. LINE 58: IF the mode of the model or estimator is being used to evaluate then the model computes the accuracy of the class predictions with respect to the actual labels.

15. LINE 62: This returns the complete EstimatorSpec model that can be used, with an explicit mode (e.g., training, predicting, evaluating), the loss, and the evaluation metrics as arguments.

```
In [0]: def cnn_model_fn(features, labels, mode):
            """Model function for CNN."""
            # 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)

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

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

            # 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),
    # 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)

# Configure the Training Op (for TRAIN mode)
if mode == tf.estimator.ModeKeys.TRAIN:
  optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
  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)
```

## 1.1 Training and Evaluating the CNN MNIST Classifier

Explanation:

1. LINE 2: The MNIST dataset is loaded and separated into training data and training label variables, along with test data and test label variables.

2. LINE 5: The training data is normalized by 255 (RGB).

3. LINE 6: Casts the training labels to a 32-bit integer type.

4. LINE 8: The training data is normalized by 255 (RGB).

5. LINE 9: Casts the training labels to a 32-bit integer type.

```
In [0]: # Load training and eval data
        ((train_data, train_labels),
         (eval_data, eval_labels)) = tf.keras.datasets.mnist.load_data()

        train_data = train_data/np.float32(255)
        train_labels = train_labels.astype(np.int32)  # not required

        eval_data = eval_data/np.float32(255)
        eval_labels = eval_labels.astype(np.int32)  # not required
```

Explanation:

1. LINE 2: Creates the classifier or model using the cnn_model_fn function from above, and the file path of where the model should be saved as arguments.

```
In [0]: # Create the Estimator
        mnist_classifier = tf.estimator.Estimator(
            model_fn=cnn_model_fn, model_dir="/tmp/mnist_convnet_model")
```

Explanation:

1. LINE 2: Creates a mapping of the probabilities with the tensor from the softmax layer that it associates with.
2. LINE 4: Sets up a hook in the CNN to log a dictionary that contains a corresponding label to a tensor from the TensorFlow graph for every 50 steps in an epoch.

```
In [0]: # Set up logging for predictions
        tensors_to_log = {"probabilities": "softmax_tensor"}

        logging_hook = tf.train.LoggingTensorHook(
            tensors=tensors_to_log, every_n_iter=50)
```

Explanation:

1. LINE 2: This creates an input object with parameters that use mini-batches of a 100 samples from the training data and labels, which are randomly shuffled, and uses the step count to signify when the training ends instead of epochs (none).
2. LINE 10: This trains the model with the input object defined above with only 1 step count, and the logging hook.

```
In [0]: # Train the model
        train_input_fn = tf.estimator.inputs.numpy_input_fn(
            x={"x": train_data},
            y=train_labels,
            batch_size=100,
            num_epochs=None,
            shuffle=True)
        # train one step and display the probabilties
        mnist_classifier.train(
            input_fn=train_input_fn,
            steps=1,
            hooks=[logging_hook])
```

Explanation:

1. LINE 1: This uses the mini-batches to train the model with 1,000 steps per epoch.

```
In [0]: mnist_classifier.train(input_fn=train_input_fn, steps=1000)
```

Explanation:

1. LINE 1: After training the model, this function computes the accuracy of the test set and the test labels on the newly trained model. Only 1 epoch is needed, since it is just a forward pass to get the predictions.

```
In [0]: 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)
```