Predictions

Restore an inference model and make predictions on an input dataset.

Chapter Goals:

- Learn how to restore an inference model and retrieve specific tensors from the computation graph
- Implement a function that makes predictions using a saved inference model

A. Restoring the model

To restore an inference model, we use the `tf.saved_model.loader.load` function. This function restores both the inference graph as well as the inference model's parameters.

Since the function's first argument is a `tf.Session` object, it's a good idea to restore the inference model within the scope of a particular `tf.Session`.

```
import tensorflow as tf

tags = [tf.saved_model.tag_constants.SERVING]
model_dir = 'inference_model'
with tf.Session(graph=tf.Graph()) as sess:
    tf.saved_model.loader.load(sess, tags, model_dir)
```

Restoring an inference model inside the scope of the tf.Session object, sess.

The second argument for `tf.saved_model.loader.load` is a list of tag constants. For inference, we use the `SERVING` tag. The function's third argument is the path to the saved inference model's directory.

In the example above, we restored the inference model within the scope of a `tf.Session` object named `sess`. Note that `sess` is initialized with a new, empty computation graph in `tf.Graph()`. This ensures that there are no graph conflicts, similar to `tf.reset_default_graph`, but also that it doesn't erase the graph outside the scope of `sess`.

B. Retrieving tensors

After restoring the inference model, the inference graph is represented by `sess.graph`, which is a `Graph` object. We use the `Graph` object's `get_tensor_by_name` function to retrieve specific tensors from the computation graph.

```
import tensorflow as tf

tags = [tf.saved_model.tag_constants.SERVING]
model_dir = 'inference_model'
with tf.Session(graph=tf.Graph()) as sess:
    tf.saved_model.loader.load(sess, tags, model_dir)
    preds = sess.graph.get_tensor_by_name('predictions:0')
```

Retrieving the predictions tensor, preds, with get_tensor_by_name.

The input argument for `get_tensor_by_name` is the name of the tensor we wish to retrieve. In the above example, the tensor name has `:0` as a suffix. Whenever we retrieve a single tensor from the computation graph, we need to add the `:0` suffix.

Time to Code!

In this chapter you'll be completing the `make_predictions` function, which makes predictions using a saved inference graph. Specifically, you'll be creating the helper function `load_inference_parts`, which loads the inference graph and gets the necessary tensors.

We can load the saved inference model using the function `tf.saved_model.loader.load`. The tag list we use to load the inference model is a singleton list containing `tf.saved_model.tag_constants.SERVING`.

Call the loading function with `sess`, the specified tag list, and `saved_model_dir` as the three input arguments.

Next, we'll get the input placeholder and the predictions tensor from the loaded computation graph. The computation graph corresponds to `sess.graph`, and we use `sess.graph.get_tensor_by_name` to retrieve a specific tensor based on its name.

Set `inputs` equal to the tensor retrieving function with

`'inference_input:0'` as the only argument.

Set `predictions_tensor` equal to the tensor retrieving function with `'predictions:0'` as the only argument.

Since the model probabilities are used to calculate the predictions, the probabilities tensor is part of the computation graph. Therefore, we can retrieve the tensor using its name.

Set `probs_tensor` equal to the tensor retrieving function with `'probs:0'` as the only argument.

Return a tuple containing `inputs` as the first element, `predictions_tensor` as the second element, and `probs_tensor` as the third element.

```
import numpy as np
import tensorflow as tf

class ClassificationModel(object):
    def __init__(self, output_size):
        self.output_size = output_size

    # Helper for make_predictions
    def load_inference_parts(self, sess, saved_model_dir):
        # CODE HERE
        pass

    # Make predictions with the inference model
    def make_predictions(self, saved_model_dir, input_data):
        with tf.Session(graph=tf.Graph()) as sess:
            inputs, predictions_tensor, probs_tensor = self.load_inference_parts(sess, saved_
            predictions, probs = sess.run((predictions_tensor, probs_tensor), feed_dict={inpu
        return predictions, probs

    # See the "Efficient Data Processing Techniques" section for details
    def dataset_from_numpy(self, input_data, batch_size, labels=None, is_training=True, num_e
        dataset_input = input_data if labels is None else (input_data, labels)
        dataset = tf.data.Dataset.from_tensor_slices(dataset_input)
        if is_training:
            dataset = dataset.shuffle(len(input_data)).repeat(num_epochs)
        return dataset.batch(batch_size)

    # See the "Machine Learning for Software Engineers" course on Educative
    def run_model_setup(self, inputs, labels, hidden_layers, is_training, calculate_accuracy=
        layer = inputs
        for num_nodes in hidden_layers:
            layer = tf.layers.dense(layer, num_nodes,
                activation=tf.nn.relu)
        logits = tf.layers.dense(layer, self.output_size,
            name='logits')
        self.probs = tf.nn.softmax(logits, name='probs')
        self.predictions = tf.argmax(
            self.probs, axis=-1, name='predictions')
        if calculate_accuracy:
            class_labels = tf.argmax(labels, axis=-1)
            is_correct = tf.equal(
                self.predictions, class_labels)
            is_correct_float = tf.cast(
                is_correct,
                tf.float32)
            self.accuracy = tf.reduce_mean(
                is_correct_float)
        if labels is not None:
            labels_float = tf.cast(
                labels, tf.float32)
            cross_entropy = tf.nn.softmax_cross_entropy_with_logits_v2(
                labels=labels_float,
                logits=logits)
            self.loss = tf.reduce_mean(
                cross_entropy)
        if is_training:
            adam = tf.train.AdamOptimizer()
            self.train_op = adam.minimize(
                self.loss, global_step=self.global_step)

    # Run training of the classification model
    def run_model_training(self, input_data, labels, hidden_layers, batch_size, num_epochs,
        self.global_step = tf.train.get_or_create_global_step()
        dataset = self.dataset_from_numpy(input_data, batch_size,
            labels=labels, num_epochs=num_epochs)
        iterator = dataset.make_one_shot_iterator()
        inputs, labels = iterator.get_next()
        self.run_model_setup(inputs, labels, hidden_layers, True)
        tf.summary.scalar('accuracy', self.accuracy)
        tf.summary.histogram('inputs', inputs)
        log_vals = {'loss': self.loss, 'step': self.global_step}
        logging_hook = tf.train.LoggingTensorHook(
            log_vals, every_n_iter=1000)
        nan_hook = tf.train.NanTensorHook(self.loss)
        hooks = [nan_hook, logging_hook]
        with tf.train.MonitoredTrainingSession(
            checkpoint_dir=ckpt_dir,
            hooks=hooks) as sess:
            while not sess.should_stop():
                sess.run(self.train_op)
```

Below is an example plot of some predictions (run the show_plot() function). The points in this plot represent the winning percentages of MLB teams in 2017.

The plot's *z*-axis corresponds to winning percentage, the *y*-axis corresponds to runs per game, and the *x*-axis corresponds to batting average.

The inference model was trained on runs per game and batting average to classify a team into one of three classes:

- *Poor*: Winning percentage below 0.450 (73 wins)
- *Average*: Winning percentage between 0.450 and 0.543 (88 wins)
- *Good*: Winning percentage above 0.543

The winning percentage thresholds are represented by the two planes in the 3-D plot. Correct predictions by the model are marked in blue, while incorrect predictions are marked in red.

If you hover over any of the points, it will give you information on the model's prediction and probability per class.

```
show_plot()
```