Save For Inference

Save a condensed model to be used for inference on real-time data.

Chapter Goals:

- Learn how to save a model for inference

A. Deploying a model

As mentioned in chapter 4, a saved model checkpoint consists of three files: `.data`, `.index`, and `.meta`. Since the `.meta` file contains the entire computation graph structure, which includes all the data in the training dataset, it can get quite large. The large file size becomes an issue when deploying an *inference* model.

An inference model is a fully trained and evaluated model used to make predictions on real-time data. When we deploy an inference model for production, we don't usually deploy the code used to build the model, either for proprietary reasons or because there are too many auxiliary code files. When we don't have the code that sets up the inference graph, we need a separate file that specifies the computation graph's structure.

B. Inference graph

The main issue with the `.meta` file is that it contains many unnecessary portions of the computation graph, with respect to inference. For inference, we only need a `tf.placeholder` to represent the input data. We also don't need any parts of the computation graph specific to training, such as the loss calculation or dataset.

So instead of using a training checkpoint for the inference model, we create a bare-bones computation graph, consisting only of the input placeholder and the computations necessary to obtain a prediction.

If the model just finished training (meaning the training computation graph is still in memory), it's necessary to use `tf.reset_default_graph` prior to building the inference graph, in order to avoid graph conflicts.

```
import tensorflow as tf

inputs = tf.placeholder(
    tf.float32, shape=(None, 3), name='inputs')
logits = tf.layers.dense(inputs, 1, name='logits')
try:
    logits = tf.layers.dense(inputs, 1, name='logits')
except ValueError: # Need to reset graph
    tf.reset_default_graph()
    inputs = tf.placeholder(
        tf.float32, shape=(None, 3), name='inputs')
    logits = tf.layers.dense(inputs, 1, name='logits')
    print(logits)
```

Using tf.reset_default_graph to clear the previous computation graph from memory.

C. Saving the model

We save the inference model using `tf.saved_model.simple_save`. The function's first argument is a `tf.Session` object and the second argument is the path to the directory where we save the inference model. Note that the directory with which we save the inference model must not already exist.

The third argument is a dictionary containing the input tensor(s) as values, with string labels as keys. The fourth required argument is also a dictionary with string keys, but for the output tensor(s), e.g. the model prediction.

The function will save a file called `saved_model.pb` and a directory called `variables` in the specified directory.

```
ls inference_dir
```

In the example above, `inference_dir` is the directory where `tf.saved_model.simple_save` saved the inference model. The `saved_model.pb` file contains the bare-bones computation graph, and is much smaller than the corresponding `.meta` file. The `variables` directory contains the model's saved parameters.

In the next chapter, you'll learn how to restore the inference model and make predictions.

Time to Code!

In this chapter you'll complete the `save_inference_graph` function, which saves the model's computation graph for inference. The function is already filled with code that restores the model state from a checkpoint.

The input dictionary for the inference graph contains `input_placeholder` as its only value, which represents the input data for the inference graph.

Set `input_dict` equal to a dictionary with a single key, `'inputs'`, that maps to `input_placeholder`.

The output dictionary for the inference graph contains `self.predictions` as its only value. The corresponding key is `'predictions'`.

Set `output_dict` equal to a dictionary consisting of the specified key-value pair.

After creating the dictionaries for the inference graph's input and output, we can save the model using `tf.saved_model.simple_save`.

Call the specified function with `sess`, `export_dir`, `input_dict`, and `output_dict` as the four input arguments.

```
import numpy as np
import tensorflow as tf

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

    # Save the model's computation graph for inference
    def save_inference_graph(self, sess, ckpt_dir, input_placeholder, export_dir):
        ckpt = tf.train.get_checkpoint_state(ckpt_dir)
        if ckpt is not None:
            saver = tf.train.Saver()
            saver.restore(sess, ckpt.model_checkpoint_path)
            # CODE HERE
            pass

    # 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, c
        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)
```