# Making Predictions

Chapter Goals:

- Use the regression model to make predictions on an unlabeled test dataset

### A. The final model

After continuous training and evaluation of the sales prediction model, we found that the initial configuration of the model (2 hidden layers with 200 and 100 nodes, respectively) performs reasonably better than smaller models and has essentially the same performance as larger models. Therefore, we stuck with the initial configuration for the final model

The final model was trained with 2.5M training steps, which took about 8 hours using a single NVIDIA Tesla P100 GPU. The final model's evaluation loss was 2873.11, which is pretty good considering the large range of possible sales values (up to $70,000).

### B. `Estimator` predictions

After notifying the project's supervisor that the model is ready, she hands us the unlabeled testing data in a CSV file. The testing data has the exact same format as the training data, minus the `'Sales'` feature. Our job is to make sales predictions for each of the observations.

Just like in training and evaluation, we make predictions with the `Estimator` by using a lambda wrapper for the input pipeline's dataset. In this case, the input pipeline contains all the unlabeled data in a TFRecords dataset.

Using the code from the Data Analysis and Data Processing Labs on the testing data's CSV file, we store the CSV data into a TFRecords file called *test.tfrecords*. We then use this file for the predictions input pipeline.

```
input_fn = lambda:create_tensorflow_dataset(
    'test.tfrecords', 1, training=False, has_labels=False)
predictions = regression_model.predict(input_fn)
```

Using the `Estimator` object ( `regression_model` ) to make predictions on the test dataset. The predictions are stored in a generator ( `predictions` ).

Using the `Estimator` object's `predict` function, we make predictions on the unlabeled dataset one observation at a time (i.e. a batch size of 1). The `predict` function returns a generator object, which we convert into a list of the prediction values (see the **Code** tab for details).

Since we don't shuffle the test dataset, the model's predictions can be easily mapped back to their corresponding data observation. In other words, the value at index $i$ of the final prediction list is the sales prediction for the $i^{th}$ observation in the test set (the $i^{th}$ row of the CSV file).

Code for using the model to make predictions is shown below

```
class SalesModel(object):
  def __init__(self, hidden_layers):
    self.hidden_layers = hidden_layers

  def run_regression_predict(self, ckpt_dir, data_file):
    regression_model = self.create_regression_model(ckpt_dir)
    input_fn = lambda:create_tensorflow_dataset(data_file, 1, training=False, has_labels=Fals
    predictions = regression_model.predict(input_fn)
    pred_list = []
    for pred_dict in predictions:
        pred_list.append(pred_dict['predictions'][0])
    return pred_list

  def run_regression_eval(self, ckpt_dir):
    regression_model = self.create_regression_model(ckpt_dir)
    input_fn = lambda:create_tensorflow_dataset('eval.tfrecords', 50, training=False)
    return regression_model.evaluate(input_fn)

  def run_regression_training(self, ckpt_dir, batch_size, num_training_steps=None):
    regression_model = self.create_regression_model(ckpt_dir)
    input_fn = lambda:create_tensorflow_dataset('train.tfrecords', batch_size)
    regression_model.train(input_fn, steps=num_training_steps)

  def create_regression_model(self, ckpt_dir):
    config = tf.estimator.RunConfig(log_step_count_steps=5000)
    regression_model = tf.estimator.Estimator(
      self.regression_fn,
      config=config,
      model_dir=ckpt_dir)
    return regression_model

  def regression_fn(self, features, labels, mode, params):
    feature_columns = create_feature_columns()
```

```
    inputs = tf.feature_column.input_layer(features, feature_columns)
    batch_predictions = self.model_layers(inputs)
    predictions = tf.squeeze(batch_predictions)

    if labels is not None:
      loss = tf.losses.absolute_difference(labels, predictions)

    if mode == tf.estimator.ModeKeys.TRAIN:
      global_step = tf.train.get_or_create_global_step()
      adam = tf.train.AdamOptimizer()
      train_op = adam.minimize(
        loss, global_step=global_step)
      return tf.estimator.EstimatorSpec(mode, loss=loss, train_op=train_op)
    if mode == tf.estimator.ModeKeys.EVAL:
      return tf.estimator.EstimatorSpec(mode, loss=loss)
    if mode == tf.estimator.ModeKeys.PREDICT:
      prediction_info = {
          'predictions': batch_predictions
      }
      return tf.estimator.EstimatorSpec(mode, predictions=prediction_info)

  def model_layers(self, inputs):
    layer = inputs
    for num_nodes in self.hidden_layers:
      layer = tf.layers.dense(layer, num_nodes,
        activation=tf.nn.relu)
    batch_predictions = tf.layers.dense(layer, 1)
    return batch_predictions
```