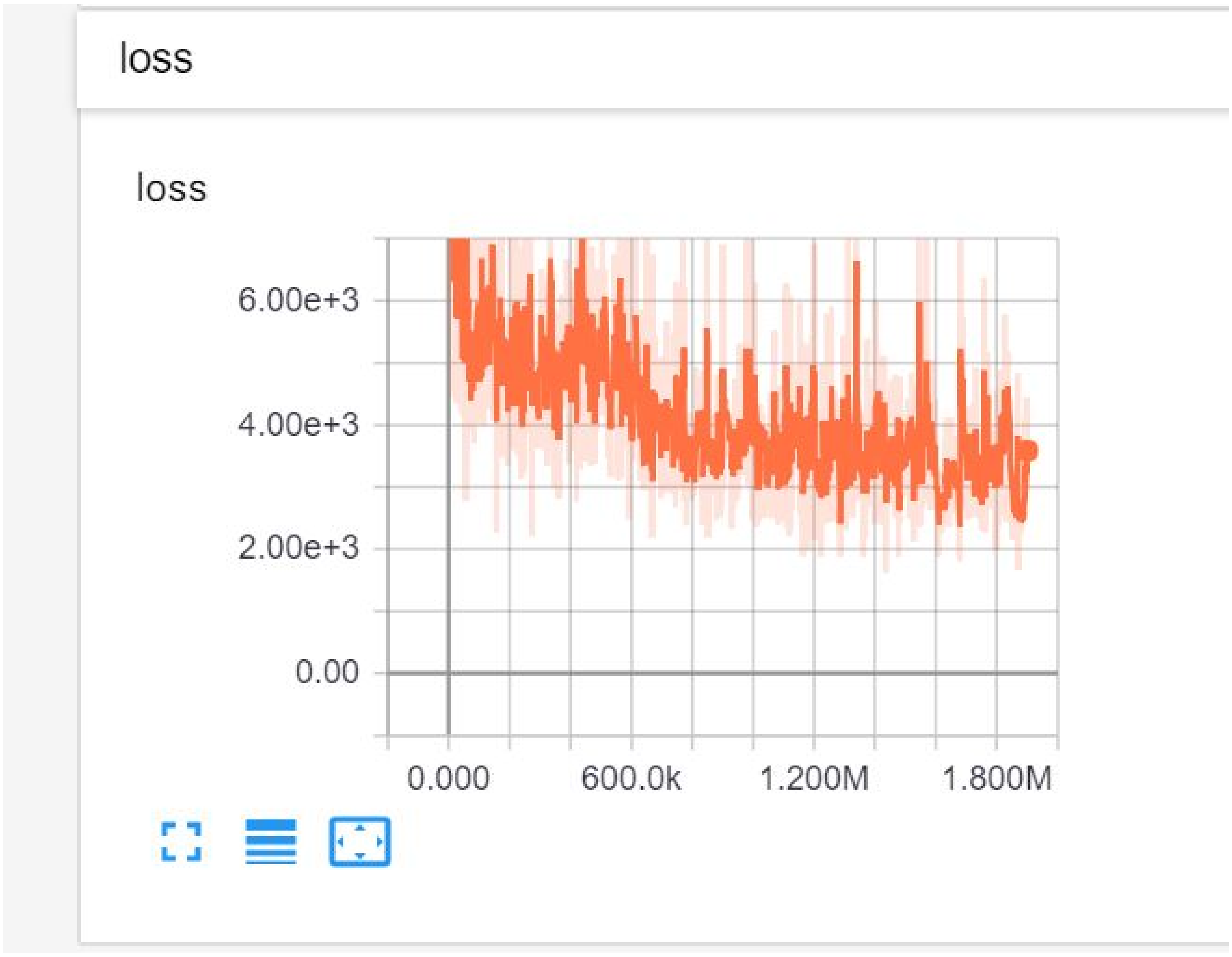
Model Evaluation

Chapter Goals:

- Evaluate the regression model

A. Evaluating with the `Estimator`

We train the model long enough that the loss begins to show signs of convergence (for our 2 hidden layer MLP model, this is around 2M training steps). To get a better look at how the loss is progressing, we can take a look at the TensorBoard training visualizations (see the Model Execution section of Deep Learning for Industry for more details on TensorBoard).



The TensorBoard line plot showing the loss progression.

We evaluate with the `Estimator` in almost the exact same way we train, with the main difference being that we use the `evaluate` function rather than the `train` function.

The evaluation dataset is contained in the *eval.tfrecords* file (created in the Data Processing Lab). The batch size argument for creating the evalution TFRecords dataset only affects evaluation speed. A larger batch size can provide a speedup in evaluation, although you have to make sure the batch is small enough to be contained in memory.

We'll use a batch size of 50 in our evaluation, which is definitely small enough to fit in memory and also faster than using a batch size of 1. The entire evaluation process takes about a minute.

```
input_fn = lambda:create_tensorflow_dataset(
    'eval.tfrecords', 50, training=False)
regression_model.evaluate(input_fn)
```

Using the Estimator object (regression_model) to evaluate the model trained with 1.9M steps. The loss (MAE) on the evaluation set is about 3267.

Code for evaluating the regression model is shown below

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

  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
```