intro_to_neural_nets

April 24, 2019

Copyright 2017 Google LLC.

1 Intro to Neural Networks

Learning Objectives: * Define a neural network (NN) and its hidden layers using the TensorFlow `DNNRegressor` class * Train a neural network to learn nonlinearities in a dataset and achieve better performance than a linear regression model

In the previous exercises, we used synthetic features to help our model incorporate nonlinearities.

One important set of nonlinearities was around latitude and longitude, but there may be others.

We'll also switch back, for now, to a standard regression task, rather than the logistic regression task from the previous exercise. That is, we'll be predicting `median_house_value` directly.

1.1 Setup

First, let's load and prepare the data.

```
In [0]: from __future__ import print_function

        import math

        from IPython import display
        from matplotlib import cm
        from matplotlib import gridspec
        from matplotlib import pyplot as plt
```

```
import numpy as np
import pandas as pd
from sklearn import metrics
import tensorflow as tf
from tensorflow.python.data import Dataset

tf.logging.set_verbosity(tf.logging.ERROR)
pd.options.display.max_rows = 10
pd.options.display.float_format = '{:.1f}'.format

california_housing_dataframe = pd.read_csv("https://download.mlcc.google.com/mledu-data

california_housing_dataframe = california_housing_dataframe.reindex(
    np.random.permutation(california_housing_dataframe.index))
```

In [0]:

```
def preprocess_features(california_housing_dataframe):
  """Prepares input features from California housing data set.

  Args:
    california_housing_dataframe: A Pandas DataFrame expected to contain data
      from the California housing data set.
  Returns:
    A DataFrame that contains the features to be used for the model, including
    synthetic features.
  """
  selected_features = california_housing_dataframe[
    ["latitude",
     "longitude",
     "housing_median_age",
     "total_rooms",
     "total_bedrooms",
     "population",
     "households",
     "median_income"]]
  processed_features = selected_features.copy()
  # Create a synthetic feature.
  processed_features["rooms_per_person"] = (
    california_housing_dataframe["total_rooms"] /
    california_housing_dataframe["population"])
  return processed_features

def preprocess_targets(california_housing_dataframe):
  """Prepares target features (i.e., labels) from California housing data set.

  Args:
    california_housing_dataframe: A Pandas DataFrame expected to contain data
      from the California housing data set.
  Returns:
```

```
    A DataFrame that contains the target feature.
  """
  output_targets = pd.DataFrame()
  # Scale the target to be in units of thousands of dollars.
  output_targets["median_house_value"] = (
    california_housing_dataframe["median_house_value"] / 1000.0)
  return output_targets
```

In [0]:
```
# Choose the first 12000 (out of 17000) examples for training.
training_examples = preprocess_features(california_housing_dataframe.head(12000))
training_targets = preprocess_targets(california_housing_dataframe.head(12000))

# Choose the last 5000 (out of 17000) examples for validation.
validation_examples = preprocess_features(california_housing_dataframe.tail(5000))
validation_targets = preprocess_targets(california_housing_dataframe.tail(5000))

# Double-check that we've done the right thing.
print("Training examples summary:")
display.display(training_examples.describe())
print("Validation examples summary:")
display.display(validation_examples.describe())

print("Training targets summary:")
display.display(training_targets.describe())
print("Validation targets summary:")
display.display(validation_targets.describe())
```

1.2 Building a Neural Network

The NN is defined by the DNNRegressor class.

Use **hidden_units** to define the structure of the NN. The hidden_units argument provides a list of ints, where each int corresponds to a hidden layer and indicates the number of nodes in it. For example, consider the following assignment:

hidden_units=[3,10]

The preceding assignment specifies a neural net with two hidden layers:

- The first hidden layer contains 3 nodes.
- The second hidden layer contains 10 nodes.

If we wanted to add more layers, we'd add more ints to the list. For example, hidden_units=[10,20,30,40] would create four layers with ten, twenty, thirty, and forty units, respectively.

By default, all hidden layers will use ReLu activation and will be fully connected.

In [0]:
```
def construct_feature_columns(input_features):
  """Construct the TensorFlow Feature Columns.

  Args:
    input_features: The names of the numerical input features to use.
```

```
    Returns:
      A set of feature columns
    """
    return set([tf.feature_column.numeric_column(my_feature)
                for my_feature in input_features])
```

In [0]:
```
def my_input_fn(features, targets, batch_size=1, shuffle=True, num_epochs=None):
    """Trains a neural net regression model.

    Args:
      features: pandas DataFrame of features
      targets: pandas DataFrame of targets
      batch_size: Size of batches to be passed to the model
      shuffle: True or False. Whether to shuffle the data.
      num_epochs: Number of epochs for which data should be repeated. None = repeat in
    Returns:
      Tuple of (features, labels) for next data batch
    """

    # Convert pandas data into a dict of np arrays.
    features = {key:np.array(value) for key,value in dict(features).items()}

    # Construct a dataset, and configure batching/repeating.
    ds = Dataset.from_tensor_slices((features,targets)) # warning: 2GB limit
    ds = ds.batch(batch_size).repeat(num_epochs)

    # Shuffle the data, if specified.
    if shuffle:
      ds = ds.shuffle(10000)

    # Return the next batch of data.
    features, labels = ds.make_one_shot_iterator().get_next()
    return features, labels
```

In [0]:
```
def train_nn_regression_model(
    learning_rate,
    steps,
    batch_size,
    hidden_units,
    training_examples,
    training_targets,
    validation_examples,
    validation_targets):
  """Trains a neural network regression model.

  In addition to training, this function also prints training progress information,
  as well as a plot of the training and validation loss over time.
```

```
Args:
  learning_rate: A `float`, the learning rate.
  steps: A non-zero `int`, the total number of training steps. A training step
    consists of a forward and backward pass using a single batch.
  batch_size: A non-zero `int`, the batch size.
  hidden_units: A `list` of int values, specifying the number of neurons in each lay
  training_examples: A `DataFrame` containing one or more columns from
    `california_housing_dataframe` to use as input features for training.
  training_targets: A `DataFrame` containing exactly one column from
    `california_housing_dataframe` to use as target for training.
  validation_examples: A `DataFrame` containing one or more columns from
    `california_housing_dataframe` to use as input features for validation.
  validation_targets: A `DataFrame` containing exactly one column from
    `california_housing_dataframe` to use as target for validation.

Returns:
  A `DNNRegressor` object trained on the training data.
"""

periods = 10
steps_per_period = steps / periods

# Create a DNNRegressor object.
my_optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
my_optimizer = tf.contrib.estimator.clip_gradients_by_norm(my_optimizer, 5.0)
dnn_regressor = tf.estimator.DNNRegressor(
    feature_columns=construct_feature_columns(training_examples),
    hidden_units=hidden_units,
    optimizer=my_optimizer,
)

# Create input functions.
training_input_fn = lambda: my_input_fn(training_examples,
                                        training_targets["median_house_value"],
                                        batch_size=batch_size)
predict_training_input_fn = lambda: my_input_fn(training_examples,
                                                training_targets["median_house_value'
                                                num_epochs=1,
                                                shuffle=False)
predict_validation_input_fn = lambda: my_input_fn(validation_examples,
                                                  validation_targets["median_house_va
                                                  num_epochs=1,
                                                  shuffle=False)

# Train the model, but do so inside a loop so that we can periodically assess
# loss metrics.
print("Training model...")
print("RMSE (on training data):")
```

```
    training_rmse = []
    validation_rmse = []
    for period in range (0, periods):
      # Train the model, starting from the prior state.
      dnn_regressor.train(
          input_fn=training_input_fn,
          steps=steps_per_period
      )
      # Take a break and compute predictions.
      training_predictions = dnn_regressor.predict(input_fn=predict_training_input_fn)
      training_predictions = np.array([item['predictions'][0] for item in training_predi

      validation_predictions = dnn_regressor.predict(input_fn=predict_validation_input_fn
      validation_predictions = np.array([item['predictions'][0] for item in validation_pr

      # Compute training and validation loss.
      training_root_mean_squared_error = math.sqrt(
          metrics.mean_squared_error(training_predictions, training_targets))
      validation_root_mean_squared_error = math.sqrt(
          metrics.mean_squared_error(validation_predictions, validation_targets))
      # Occasionally print the current loss.
      print("  period %02d : %0.2f" % (period, training_root_mean_squared_error))
      # Add the loss metrics from this period to our list.
      training_rmse.append(training_root_mean_squared_error)
      validation_rmse.append(validation_root_mean_squared_error)
    print("Model training finished.")

    # Output a graph of loss metrics over periods.
    plt.ylabel("RMSE")
    plt.xlabel("Periods")
    plt.title("Root Mean Squared Error vs. Periods")
    plt.tight_layout()
    plt.plot(training_rmse, label="training")
    plt.plot(validation_rmse, label="validation")
    plt.legend()

    print("Final RMSE (on training data):   %0.2f" % training_root_mean_squared_error)
    print("Final RMSE (on validation data): %0.2f" % validation_root_mean_squared_error)

    return dnn_regressor
```

1.3 Task 1: Train a NN Model

Adjust hyperparameters, aiming to drop RMSE below 110.

Run the following block to train a NN model.

Recall that in the linear regression exercise with many features, an RMSE of 110 or so was pretty good. We'll aim to beat that.

Your task here is to modify various learning settings to improve accuracy on validation data.

Overfitting is a real potential hazard for NNs. You can look at the gap between loss on training data and loss on validation data to help judge if your model is starting to overfit. If the gap starts to grow, that is usually a sure sign of overfitting.

Because of the number of different possible settings, it's strongly recommended that you take notes on each trial to help guide your development process.

Also, when you get a good setting, try running it multiple times and see how repeatable your result is. NN weights are typically initialized to small random values, so you should see differences from run to run.

```
In [0]: dnn_regressor = train_nn_regression_model(
            learning_rate=0.01,
            steps=500,
            batch_size=10,
            hidden_units=[10, 2],
            training_examples=training_examples,
            training_targets=training_targets,
            validation_examples=validation_examples,
            validation_targets=validation_targets)
```

1.3.1 Solution

Click below to see a possible solution

NOTE: This selection of parameters is somewhat arbitrary. Here we've tried combinations that are increasingly complex, combined with training for longer, until the error falls below our objective (training is nondeterministic, so results may fluctuate a bit each time you run the solution). This may not be the best combination; others may attain an even lower RMSE. If your aim is to find the model that can attain the best error, then you'll want to use a more rigorous process, like a parameter search.

```
In [0]: dnn_regressor = train_nn_regression_model(
            learning_rate=0.001,
            steps=2000,
            batch_size=100,
            hidden_units=[10, 10],
            training_examples=training_examples,
            training_targets=training_targets,
            validation_examples=validation_examples,
            validation_targets=validation_targets)
```

1.4 Task 2: Evaluate on Test Data

Confirm that your validation performance results hold up on test data.

Once you have a model you're happy with, evaluate it on test data to compare that to validation performance.

Reminder, the test data set is located here.

```
In [0]: california_housing_test_data = pd.read_csv("https://download.mlcc.google.com/mledu-data

        # YOUR CODE HERE
```

1.4.1 Solution

Click below to see a possible solution.

Similar to what the code at the top does, we just need to load the appropriate data file, preprocess it and call predict and mean_squared_error.

Note that we don't have to randomize the test data, since we will use all records.

```
In [0]: california_housing_test_data = pd.read_csv("https://download.mlcc.google.com/mledu-data

        test_examples = preprocess_features(california_housing_test_data)
        test_targets = preprocess_targets(california_housing_test_data)

        predict_testing_input_fn = lambda: my_input_fn(test_examples,
                                                       test_targets["median_house_value"],
                                                       num_epochs=1,
                                                       shuffle=False)

        test_predictions = dnn_regressor.predict(input_fn=predict_testing_input_fn)
        test_predictions = np.array([item['predictions'][0] for item in test_predictions])

        root_mean_squared_error = math.sqrt(
            metrics.mean_squared_error(test_predictions, test_targets))

        print("Final RMSE (on test data): %0.2f" % root_mean_squared_error)
```