improving_neural_net_performance

April 24, 2019

1 Improving Neural Net Performance

Learning Objective: Improve the performance of a neural network by normalizing features and applying various optimization algorithms

NOTE: The optimization methods described in this exercise are not specific to neural networks; they are effective means to improve most types of models.

1.1 Setup

First, we'll load 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 Train the Neural Network

Next, we'll train the neural network.

```
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 network 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(
            my_optimizer,
            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:
            my_optimizer: An instance of `tf.train.Optimizer`, the optimizer to use.
            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 tuple `(estimator, training_losses, validation_losses)`:
```

```
    estimator: the trained `DNNRegressor` object.
    training_losses: a `list` containing the training loss values taken during train
    validation_losses: a `list` containing the validation loss values taken during t
"""

periods = 10
steps_per_period = steps / periods

# Create a DNNRegressor object.
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_f
  validation_predictions = np.array([item['predictions'][0] for item in validation_p
  # 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, training_rmse, validation_rmse
```

```
In [0]: _ = train_nn_regression_model(
            my_optimizer=tf.train.GradientDescentOptimizer(learning_rate=0.0007),
            steps=5000,
            batch_size=70,
            hidden_units=[10, 10],
            training_examples=training_examples,
            training_targets=training_targets,
            validation_examples=validation_examples,
            validation_targets=validation_targets)
```

1.3 Linear Scaling

It can be a good standard practice to normalize the inputs to fall within the range -1, 1. This helps SGD not get stuck taking steps that are too large in one dimension, or too small in another. Fans of numerical optimization may note that there's a connection to the idea of using a preconditioner here.

```
In [0]: def linear_scale(series):
          min_val = series.min()
          max_val = series.max()
          scale = (max_val - min_val) / 2.0
          return series.apply(lambda x:((x - min_val) / scale) - 1.0)
```

1.4 Task 1: Normalize the Features Using Linear Scaling

Normalize the inputs to the scale -1, 1.

Spend about 5 minutes training and evaluating on the newly normalized data. How well can you do?

As a rule of thumb, NN's train best when the input features are roughly on the same scale.

Sanity check your normalized data. (What would happen if you forgot to normalize one feature?)

```
In [0]: def normalize_linear_scale(examples_dataframe):
          """Returns a version of the input `DataFrame` that has all its features normalized l
          #
          # Your code here: normalize the inputs.
          #
          pass

        normalized_dataframe = normalize_linear_scale(preprocess_features(california_housing_da
        normalized_training_examples = normalized_dataframe.head(12000)
        normalized_validation_examples = normalized_dataframe.tail(5000)

        _ = train_nn_regression_model(
            my_optimizer=tf.train.GradientDescentOptimizer(learning_rate=0.0007),
            steps=5000,
            batch_size=70,
            hidden_units=[10, 10],
            training_examples=normalized_training_examples,
            training_targets=training_targets,
            validation_examples=normalized_validation_examples,
            validation_targets=validation_targets)
```

1.4.1 Solution

Click below for one possible solution.

Since normalization uses min and max, we have to ensure it's done on the entire dataset at once.

We can do that here because all our data is in a single DataFrame. If we had multiple data sets, a good practice would be to derive the normalization parameters from the training set and apply those identically to the test set.

```
In [0]: def normalize_linear_scale(examples_dataframe):
          """Returns a version of the input `DataFrame` that has all its features normalized l
          processed_features = pd.DataFrame()
          processed_features["latitude"] = linear_scale(examples_dataframe["latitude"])
          processed_features["longitude"] = linear_scale(examples_dataframe["longitude"])
          processed_features["housing_median_age"] = linear_scale(examples_dataframe["housing_r
          processed_features["total_rooms"] = linear_scale(examples_dataframe["total_rooms"])
          processed_features["total_bedrooms"] = linear_scale(examples_dataframe["total_bedroom
          processed_features["population"] = linear_scale(examples_dataframe["population"])
          processed_features["households"] = linear_scale(examples_dataframe["households"])
```

```
    processed_features["median_income"] = linear_scale(examples_dataframe["median_income
    processed_features["rooms_per_person"] = linear_scale(examples_dataframe["rooms_per_
    return processed_features

  normalized_dataframe = normalize_linear_scale(preprocess_features(california_housing_da
  normalized_training_examples = normalized_dataframe.head(12000)
  normalized_validation_examples = normalized_dataframe.tail(5000)

  _ = train_nn_regression_model(
      my_optimizer=tf.train.GradientDescentOptimizer(learning_rate=0.005),
      steps=2000,
      batch_size=50,
      hidden_units=[10, 10],
      training_examples=normalized_training_examples,
      training_targets=training_targets,
      validation_examples=normalized_validation_examples,
      validation_targets=validation_targets)
```

1.5 Task 2: Try a Different Optimizer

** Use the Adagrad and Adam optimizers and compare performance.**

The Adagrad optimizer is one alternative. The key insight of Adagrad is that it modifies the learning rate adaptively for each coefficient in a model, monotonically lowering the effective learning rate. This works great for convex problems, but isn't always ideal for the non-convex problem Neural Net training. You can use Adagrad by specifying `AdagradOptimizer` instead of `GradientDescentOptimizer`. Note that you may need to use a larger learning rate with Adagrad.

For non-convex optimization problems, Adam is sometimes more efficient than Adagrad. To use Adam, invoke the `tf.train.AdamOptimizer` method. This method takes several optional hyperparameters as arguments, but our solution only specifies one of these (`learning_rate`). In a production setting, you should specify and tune the optional hyperparameters carefully.

```
In [0]: #
        # YOUR CODE HERE: Retrain the network using Adagrad and then Adam.
        #
```

1.5.1 Solution

Click below for the solution

First, let's try Adagrad.

```
In [0]: _, adagrad_training_losses, adagrad_validation_losses = train_nn_regression_model(
            my_optimizer=tf.train.AdagradOptimizer(learning_rate=0.5),
            steps=500,
            batch_size=100,
            hidden_units=[10, 10],
            training_examples=normalized_training_examples,
            training_targets=training_targets,
            validation_examples=normalized_validation_examples,
            validation_targets=validation_targets)
```

Now let's try Adam.

```
In [0]: _, adam_training_losses, adam_validation_losses = train_nn_regression_model(
            my_optimizer=tf.train.AdamOptimizer(learning_rate=0.009),
            steps=500,
            batch_size=100,
            hidden_units=[10, 10],
            training_examples=normalized_training_examples,
            training_targets=training_targets,
            validation_examples=normalized_validation_examples,
            validation_targets=validation_targets)
```

Let's print a graph of loss metrics side by side.

```
In [0]: plt.ylabel("RMSE")
        plt.xlabel("Periods")
        plt.title("Root Mean Squared Error vs. Periods")
        plt.plot(adagrad_training_losses, label='Adagrad training')
        plt.plot(adagrad_validation_losses, label='Adagrad validation')
        plt.plot(adam_training_losses, label='Adam training')
        plt.plot(adam_validation_losses, label='Adam validation')
        _ = plt.legend()
```

1.6 Task 3: Explore Alternate Normalization Methods

Try alternate normalizations for various features to further improve performance.

If you look closely at summary stats for your transformed data, you may notice that linear scaling some features leaves them clumped close to `-1`.

For example, many features have a median of `-0.8` or so, rather than `0.0`.

```
In [0]: _ = normalized_training_examples.hist(bins=20, figsize=(18, 12), xlabelsize=10)
```

We might be able to do better by choosing additional ways to transform these features.

For example, a log scaling might help some features. Or clipping extreme values may make the remainder of the scale more informative.

```
In [0]: def log_normalize(series):
          return series.apply(lambda x:math.log(x+1.0))

        def clip(series, clip_to_min, clip_to_max):
          return series.apply(lambda x:(
            min(max(x, clip_to_min), clip_to_max)))

        def z_score_normalize(series):
          mean = series.mean()
          std_dv = series.std()
          return series.apply(lambda x:(x - mean) / std_dv)

        def binary_threshold(series, threshold):
          return series.apply(lambda x:(1 if x > threshold else 0))
```

The block above contains a few additional possible normalization functions. Try some of these, or add your own.

Note that if you normalize the target, you'll need to un-normalize the predictions for loss metrics to be comparable.

```
In [0]: def normalize(examples_dataframe):
          """Returns a version of the input `DataFrame` that has all its features normalized."
          #
          # YOUR CODE HERE: Normalize the inputs.
          #
          pass

        normalized_dataframe = normalize(preprocess_features(california_housing_dataframe))
        normalized_training_examples = normalized_dataframe.head(12000)
        normalized_validation_examples = normalized_dataframe.tail(5000)

        _ = train_nn_regression_model(
            my_optimizer=tf.train.GradientDescentOptimizer(learning_rate=0.0007),
            steps=5000,
            batch_size=70,
            hidden_units=[10, 10],
            training_examples=normalized_training_examples,
            training_targets=training_targets,
            validation_examples=normalized_validation_examples,
            validation_targets=validation_targets)
```

1.6.1 Solution

Click below for one possible solution.

These are only a few ways in which we could think about the data. Other transformations may work even better!

`households`, `median_income` and `total_bedrooms` all appear normally-distributed in a log space.

`latitude`, `longitude` and `housing_median_age` would probably be better off just scaled linearly, as before.

`population`, `totalRooms` and `rooms_per_person` have a few extreme outliers. They seem too extreme for log normalization to help. So let's clip them instead.

```
In [0]: def normalize(examples_dataframe):
          """Returns a version of the input `DataFrame` that has all its features normalized."
          processed_features = pd.DataFrame()

          processed_features["households"] = log_normalize(examples_dataframe["households"])
          processed_features["median_income"] = log_normalize(examples_dataframe["median_income
          processed_features["total_bedrooms"] = log_normalize(examples_dataframe["total_bedroo

          processed_features["latitude"] = linear_scale(examples_dataframe["latitude"])
          processed_features["longitude"] = linear_scale(examples_dataframe["longitude"])
```

```
  processed_features["housing_median_age"] = linear_scale(examples_dataframe["housing_m

  processed_features["population"] = linear_scale(clip(examples_dataframe["population"]
  processed_features["rooms_per_person"] = linear_scale(clip(examples_dataframe["rooms_
  processed_features["total_rooms"] = linear_scale(clip(examples_dataframe["total_rooms

  return processed_features

normalized_dataframe = normalize(preprocess_features(california_housing_dataframe))
normalized_training_examples = normalized_dataframe.head(12000)
normalized_validation_examples = normalized_dataframe.tail(5000)

_ = train_nn_regression_model(
    my_optimizer=tf.train.AdagradOptimizer(learning_rate=0.15),
    steps=1000,
    batch_size=50,
    hidden_units=[10, 10],
    training_examples=normalized_training_examples,
    training_targets=training_targets,
    validation_examples=normalized_validation_examples,
    validation_targets=validation_targets)
```

1.7 Optional Challenge: Use only Latitude and Longitude Features

Train a NN model that uses only latitude and longitude as features.

Real estate people are fond of saying that location is the only important feature in housing price. Let's see if we can confirm this by training a model that uses only latitude and longitude as features.

This will only work well if our NN can learn complex nonlinearities from latitude and longitude.

NOTE: We may need a network structure that has more layers than were useful earlier in the exercise.

```
In [0]: #
        # YOUR CODE HERE: Train the network using only latitude and longitude
        #
```

1.7.1 Solution

Click below for a possible solution.

It's a good idea to keep latitude and longitude normalized:

```
In [0]: def location_location_location(examples_dataframe):
          """Returns a version of the input `DataFrame` that keeps only the latitude and longi
          processed_features = pd.DataFrame()
          processed_features["latitude"] = linear_scale(examples_dataframe["latitude"])
          processed_features["longitude"] = linear_scale(examples_dataframe["longitude"])
          return processed_features
```

```
lll_dataframe = location_location_location(preprocess_features(california_housing_data
lll_training_examples = lll_dataframe.head(12000)
lll_validation_examples = lll_dataframe.tail(5000)

_ = train_nn_regression_model(
    my_optimizer=tf.train.AdagradOptimizer(learning_rate=0.05),
    steps=500,
    batch_size=50,
    hidden_units=[10, 10, 5, 5, 5],
    training_examples=lll_training_examples,
    training_targets=training_targets,
    validation_examples=lll_validation_examples,
    validation_targets=validation_targets)
```

This isn't too bad for just two features. Of course, property values can still vary significantly within short distances.