Regressions

Advanced Machine Learning

March 22, 2018

In this lab you are going to implement several regression models with `tensorflow`.

1 Background

This section provides the background on tensorflow. If your are familiar with this library, skip to section 3.

1.1 Dataflow graph

TensorFlow is a tool for machine learning that was designed with scalable neural network models in mind. It is a tool for computations, which are expressed in term of dataflow graph. Consider the following code snippet

```
1  import tensorflow as tf
2
3  x = tf.constant(1., name="x")
4  y = tf.constant(2., name="y")
5  z = tf.constant(3., name="z")
6
7  # c = (x + y) * z
8  c = tf.multiply(tf.add(x, y), z, name="c")
```

Here we define three constants, and perform addition and the result of multiplication is presumably stored in the variable c (in tensorflow this is not technically true). On the figure 1 you can see that all operations can be represented in terms of directed graph, where x, y, and z are the leaf nodes, and c corresponds to the top level operation.

The level of code verbosity is not mandatory as in the code above. Operation on line 7 is equivalent to the line 8, and results in the same computation graph shown on figure 2. The graph is the same, with the only exception that the name of the top level node is not set to c, and this allows us to see that it corresponds not to the value of c, but to the top level operation.

```
1  x = tf.constant(1., name="x")
2  y = tf.constant(2., name="y")
3  z = tf.constant(3., name="z")
4  c = (x + y) * z
```

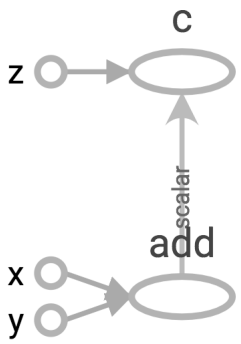

Figure 1: Simple dataflow graph

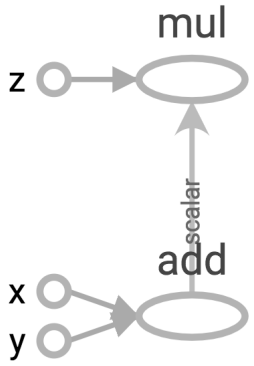

Figure 2: Simple dataflow graph for line 7

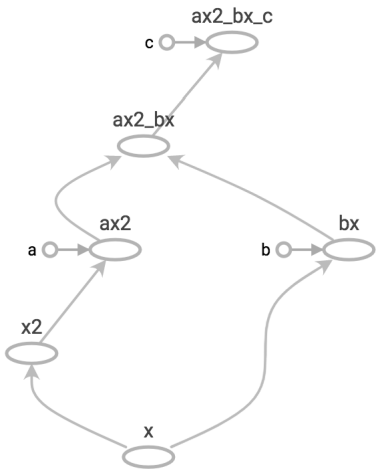

Figure 3: Dataflow graph for quadratic equation

Consider a more complicated example below, where we calculate the value of quadratic equation with constants a, b, and c and the variable x. Here we explicitly assign a name for every operation, which can be useful when debugging. Alternatively, lines 6-10 can be replaced by line 11. You can notice an unusual object: placeholder. The purpose of different objects is to be discussed in the next section.

```
1  a = tf.constant(1., name="a")
2  b = tf.constant(2., name="b")
3  c = tf.constant(3., name="c")
4  x = tf.placeholder(dtype=tf.float32, name="x")
5
6  x_2 = tf.square(x, name="x2")
7  a_x_2 = tf.multiply(a, x_2, name="ax2")
8  b_x = tf.multiply(b, x, name="bx")
9  temp = tf.add(a_x_2, b_x, name="ax2_bx")
10 y = tf.add(temp, c, name="ax2_bx_c")
11 # y = a * x**2. + b * x + c
```

Notice how every data unit and operations on the figure 3 are represented as graph nodes. For more information about computational graphs and their visualization please refer to documentation.

1.2 Tensors

In Tensorflow everything is expressed through tensors. Complete information about tensors can be found in the official documentation, according to which tensor is:

”a generalization of vectors and matrices to potentially higher dimensions. Internally, TensorFlow represents tensors as n-dimensional arrays of base datatypes.

When writing a TensorFlow program, the main object you manipulate and pass around is the `tf.Tensor`. A `tf.Tensor` object represents a partially defined computation that will eventually produce a value. TensorFlow programs work by first building a graph of `tf.Tensor` objects, detailing how each tensor is computed based on the other available tensors and then by running parts of this graph to achieve the desired results"

According to the description above, operations like square, multiplication, add that we used above also produce a tensor object. It is said to be "partially defined" because a tensor object can exist even if the actual value of the original nodes, such as x, was not yet defined. The most common tensor objects that contain some data and can be used as the starting point of the computations graph are

- `tf.Variable`: this tensor is mutable, and its value can be changed during the execution of your program
- `tf.constant`: this tensor is immutable, once you set the value of the constant it is incorporated into the computation graph. Changing the constant will require creating a new graph.
- `tf.placeholder`: this tensor is immutable, it can accept `numpy` array as its value and is often used to represent you training data. The reason os making it immutable is clear: you do not want to change your training data during the learning process.

For additional information please refer to documentation.

2 Regression with tensorflow

2.1 Linear regression

2.1.1 Vectorization

Linear regression can be described with the following expression

$$\hat{y} = W \cdot \mathbf{x} + b$$

Here $\mathbf{x} \in \mathcal{R}^{k\times 1}$, $W \in \mathcal{R}^{1\times k}$, $b \in \mathcal{R}$ where k is the number of features. Implementing the linear regression in this fashion is not wise since it does not utilize the benefits of full vectorization. Let us have a matrix $\mathbf{X} \in \mathcal{R}^{n\times k}$ and vector $\mathbf{y} \in \mathcal{R}^{n\times 1}$that represent our input data and target values respectively. Here n is the number of training examples. Then the linear regression can expressed in the form

$$\hat{\mathbf{y}}^T = W \cdot \mathbf{X}^T + \mathbf{b} \tag{1}$$

where $\hat{\mathbf{y}}$is the estimated value of $\mathbf{y}$, $W \in \mathcal{R}^{1\times k}$, $b \in \mathcal{R}$. As you can see this way the dimensions of trainable parameters are not affected, but calculation of target values is vectorized.

The simplest way to optimize parameters of a linear regression is through MSE loss, which appears to be convex. For vectorized implementation, the error can be expressed with the equation (2)

$$loss = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{2}$$

2.1.2 Implementation

We are going to have a look at weather history dataset, and try to estimate temperature from humidity, wind speed, wind bearing, visibility, and pressure. In this part you are provided with sample code.

The process of building and executing a program in tensorflow normally consists of three steps:

1. Build a dataflow graph
2. Initialize variables
3. Evaluate graph

Building tensorflow graph As it was described before, graph represents the order of computation. Building a graph means specifying the operations that are to perform in your program. Let's first define the initial nodes of the graph for calculating linear regression

```
1 X = tf.placeholder(shape=(None, num_features), dtype=tf.float32,
      name="input")
2 Y = tf.placeholder(shape=(None, 1), dtype=tf.float32, name="target"
      )
3 W = tf.get_variable(shape=w_shape, dtype=tf.float32, name="weights"
      , initializer=None)
4 b = tf.get_variable(shape=b_shape, dtype=tf.float32, name="bias",
      initializer=None)
```

Here we utilize two types of objects: placeholder that will hold the input date, and the weights and biases that are to be trained. It is worts noting several things. First, in tensorflow you always specify the type of data you have, this information is taken in consideration by backend. Second, you could notice that the shape of placeholders is not well defined. We do this to allow tensorflow to accept input data with dynamic first dimension. The statement `(None, num_features)` is equivalent to `(Any, num_features)`. Third, tensorflow provide an option of random initialization for variables. In our case we do not need any specific initialization procedure since linear regression always converges. More information about initializers can be found in documentation.

Linear regression can be implemented in tensorflow in the following way.

```
1 Y_hat = tf.transpose(tf.matmul(W, tf.transpose(X)) + b)
2 loss = tf.reduce_mean(tf.square(Y - Y_hat, name="loss"))
```

In the listing above, the first line corresponds to the equation (1). You could notice that the dimension of the bias vector is not compatible in terms of linear algebra, but the calculation is still possible due to broadcasting. The second line defines the loss from the equation (2).

On the next step we need to define the optimization procedure.

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.1)
opt = optimizer.minimize(loss)
```

First, we create an optimizer object and then we define the optimization tensor `opt`. When tensor `opt` evaluated, tensorflow will automatically detect trainable parameters, compute gradients and perform a single gradient descent optimization step.

Initialization Prior to evaluation all variables should be initialized. In tensorflow, this performed by defining initialization tensor, and evaluating it in the session scope.

```
initializer = tf.global_variables_initializer()
with tf.Session() as sess:
  sess.run(initializer)
```

Evaluation Tensors are evaluated in the scope of session object `tf.Session()`. Tensors are evaluated in a method `run` that accepts the node of the graph and the input data as parameters. To perform optimization in our example, we need to specify input parameters X and Y.

```
with tf.Session() as sess:
  sess.run(initializer)
  for i in range(epochs):
    # perfrom single optimization step
    sess.run(opt, {X: X_train, Y: Y_train})
```

Putting everything together Now that we covered the steps of creating and evaluating the computation graph in tensorflow, let's put everything together.

```
import tensorflow as tf
from utils import load_data
from sklearn.model_selection import train_test_split

data, target = load_data("weatherHistory.csv")
X_train, X_test, Y_train, Y_test = train_test_split(data, target,
    test_size=.3)

num_features = X_train.shape[1]
w_shape = (1, num_features)
b_shape = (1, 1)

X = tf.placeholder(shape=(None, num_features), dtype=tf.float32,
    name="input")
Y = tf.placeholder(shape=(None, 1), dtype=tf.float32, name="target"
    )
```

```
14  W = tf.get_variable(shape=w_shape, dtype=tf.float32, name="weights"
        , initializer=None)
15  b = tf.get_variable(shape=b_shape, dtype=tf.float32, name="bias",
        initializer=None)
16
17  Y_hat = tf.transpose(tf.matmul(W, tf.transpose(X)) + b)
18  loss = tf.reduce_mean(tf.square(Y - Y_hat, name="loss"))
19
20  optimizer = tf.train.AdamOptimizer(learning_rate=0.1)
21  opt = optimizer.minimize(loss)
22
23  initializer = tf.global_variables_initializer()
24
25  epochs = 100
26  l = 0.
27
28  with tf.Session() as sess:
29    sess.run(initializer)
30    for i in range(epochs):
31      _, loss_train = sess.run([opt, loss],
32                               {X: X_train, Y: Y_train})
33      loss_test = sess.run(loss, {X: X_test, Y: Y_test})
34      print("\rIteraiton: %d "
35          "Train loss: %.2f "
36          "Test loss: %.2f" %
37          (i, loss_train, loss_test), end="")
38    print("\nWeights: ", sess.run(W))
```

The graph for this model is shown on figure 4

3 Logistic regression

Implement logistic regression in tensorflow. Use `candy.csv` as your input data. In this task you need to determine whether the candy is chocolate or not. For this task you need to modify the loss function to cross entropy.

$$loss = \frac{1}{n}\sum_{i=1}^{n} y_i \cdot log(\hat{y}_i) + (1 - y_i) \cdot log(1 - \hat{y}_i) \tag{3}$$

For the sake of numerical stability, tensorflow computes the sigmoid function on its own when evaluating cross entropy, thus you do not need to evaluate sigmoid function during training. You can calculate the loss as

```
1  loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
       labels=Y, logits=Y_hat))
```

For the inference during test time you can use tf.sigmoid.

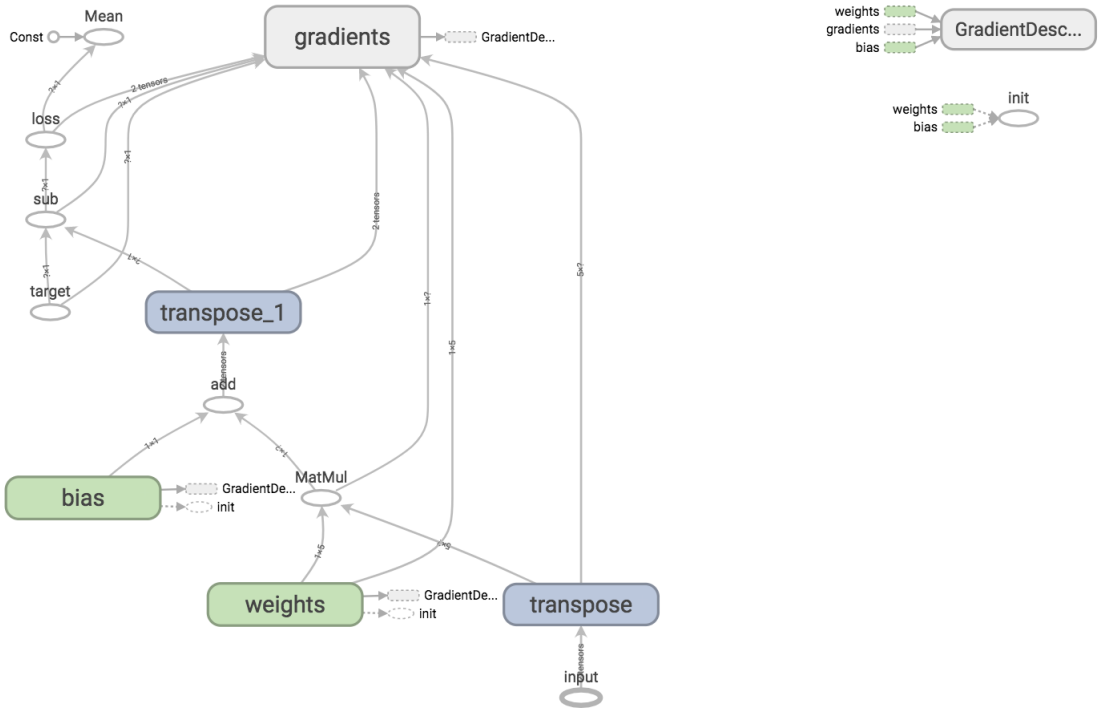

Figure 4: Linear regression tensorflow graph