TensorFlow for linear regression

Glenn Bruns
CSUMB

Much material in this deck from Géron, Hands-on Machine Learning with Scikit-Learn and TensorFlow

Learning outcomes

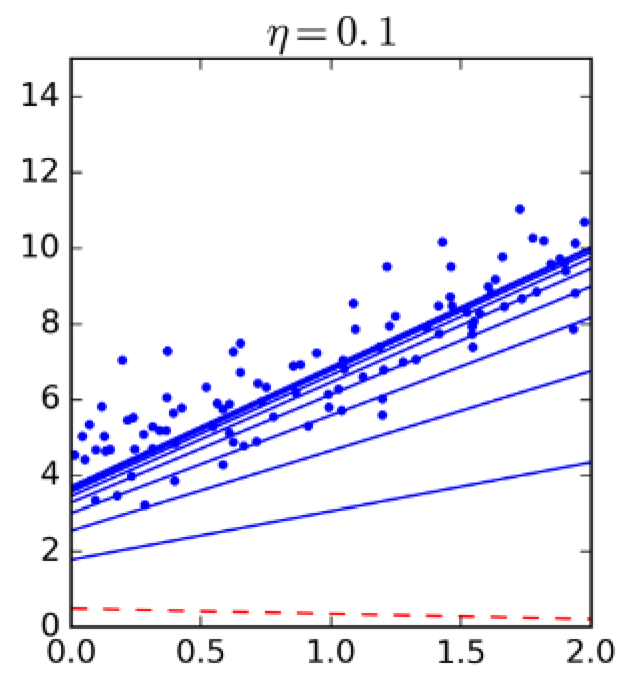
After this lecture you should be able to:

1. Write TensorFlow code to perform linear regression with gradient descent
2. Explain how every line of code works

Recap of optimization for ML

How to use optimization to find the “best” line through a set of training data?

1. We have a linear model, with parameters for slope and y-intercept
2. We have a bunch of training data
3. We define a cost function, where the “cost” is high if the line fits the data poorly
4. We get the best line by finding the parameter values that minimize the cost function
5. We can find the minimum using gradient descent.



Linear regression

Prediction in linear regression:

$$\hat{y} = \theta^T \cdot \mathbf{x}$$

θ is the model's parameter vector

$\mathbf{x}$ is the feature vector ($\mathbf{x}_0$ is always 1)

$\hat{y}$ is the estimated (predicted) value of y

MSE cost function for linear regression:

$$MSE(\mathbf{X}, \theta) = \frac{1}{m} \sum_{i=1}^{m} \left(\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)}\right)^2$$

Reminder: Geron writes $\mathbf{x}^{(i)}$ for the ith row of matrix X.

Given training data $\mathbf{X}$, we want the value of θ that *minimizes* $MSE(\mathbf{X}, \theta)$

Gradient of the cost function

Partial derivative of the cost function, for some θ_j:

$$\frac{\partial}{\partial \theta_j} MSE(\mathbf{X}, \theta) = \frac{2}{m} \sum_{i=1}^{m} \left(\theta^T \cdot x^{(i)} - y^{(i)} \right) x_j^{(i)}$$

The vector of all the partial derivatives is the gradient of the function:

$$\nabla_\theta \, \mathrm{MSE}(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \mathrm{MSE}(\theta) \\ \frac{\partial}{\partial \theta_1} \mathrm{MSE}(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \mathrm{MSE}(\theta) \end{pmatrix} = \frac{2}{m} \mathbf{X}^T \cdot (\mathbf{X} \cdot \theta - \mathbf{y})$$

Batch gradient descent

```
eta = 0.1  # learning rate
n_iterations = 1000
m = 100

theta = np.random.randn(2,1)  # random initialization

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

theta that maximizes negative cost =

theta that minimizes cost

$$\frac{2}{m}\mathbf{X}^T \cdot (\mathbf{X} \cdot \theta - \mathbf{y})$$

source: Géron, Hands-On Machine Learning text

Gradient descent with TensorFlow

```
n_epochs = 1000
learning_rate = 0.01
```

```
X = tf.constant(scaled_housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0, seed=42),
name="theta")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
gradients = 2/m * tf.matmul(tf.transpose(X), error)
training_op = tf.assign(theta, theta - learning_rate * gradients)

init = tf.global_variables_initializer()
```

```
with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        if epoch % 100 == 0:
            print("Epoch", epoch, "MSE =", mse.eval())
        sess.run(training_op)

    best_theta = theta.eval()
```

Graph construction phase

```
X = tf.constant(scaled_housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0, seed=42),
name="theta")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
gradients = 2/m * tf.matmul(tf.transpose(X), error)
training_op = tf.assign(theta, theta - learning_rate * gradients)

init = tf.global_variables_initializer()
```

`X, scaled_housing_data_plus_bias`:	m x (n+1) matrix
`housing.target`:	array of length m
`y, housing.target.reshape(-1,1)`:	m x 1 matrix
`theta`:	(n+1) x 1 matrix

$$\nabla_\theta \mathrm{MSE}(\theta) = \frac{2}{\mathrm{m}} \mathbf{X}^{\mathrm{T}} \cdot (\overbrace{\underbrace{\mathbf{X} \cdot \theta}^{\text{y\_pred}} - \mathbf{y}}_{\text{error}})$$

Execution phase

```
with tf.Session() as sess:
    sess.run(init)                   # initialize variables

    for epoch in range(n_epochs):
        if epoch % 100 == 0:
            print("Epoch", epoch, "MSE =", mse.eval())
        sess.run(training_op)  # one step of gradient descent

    best_theta = theta.eval()  # result is in theta node
```

Note that `sess.run(training_op)` could be `training_op.eval()`

What termination condition for gradient descent is used here?

Computing gradients with autodiff

```
…
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
gradients = 2/m * tf.matmul(tf.transpose(X), error)
training_op = tf.assign(theta, theta - learning_rate * gradients)
```

Limitations of symbolic differentiation:
- for many functions, it's difficult or impossible to do
- even if possible, efficient implementation can be tricky

TensorFlow supports reverse-mode autodiff

```
…
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
gradients = tf.gradients(mse, [theta])[0]
training_op = tf.assign(theta, theta - learning_rate * gradients)
```

Using TensorFlow's gradient descent

```
…
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
gradients = tf.gradients(mse, [theta])[0]
training_op = tf.assign(theta, theta - learning_rate * gradients)
```

You can use one of TensorFlow's built-in optimizers instead:

```
…
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(mse)
```

Switching to an alternative TF optimizer is super-easy.

Mini-batch gradient descent

If we want to move from batch to mini-batch, what code will change?

```
X = tf.constant(scaled_housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0, seed=42),
name="theta")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square
optimizer = tf.train.GradientD
training_op = optimizer.minimi

init = tf.global_variables_ini

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        if epoch % 100 == 0:
            print("Epoch", epoch, "MSE =", mse.eval())
        sess.run(training_op)

    best_theta = theta.eval()
```

Solution idea:

- turn X and y into "placeholders", that can be fed with data
- before each step of gradient descent, feed X and y with a part of the training/target data

Mini-batch gradient descent, part 1

Before (batch):

```
X = tf.constant(scaled_housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")
…
```

After (mini-batch):

```
X = tf.placeholder(tf.float32, shape=(None, n + 1), name="X")
y = tf.placeholder(tf.float32, shape=(None, 1), name="y")
…
```

The purpose of a placeholder is just to accept a feed.

It looks like a variable but it can't be evaluated.

Its value is determined by the 'feed_dict' argument to Session.run().

Mini-batch gradient descent, part 2

before (batch):

```
with tf.Session() as sess:
    sess.run(init)
    for epoch in range(n_epochs):
       sess.run(training_op)
    best_theta = theta.eval()
```

after (mini-batch):

```
def fetch_batch(epoch, batch_index, batch_size):
    # X_batch is batch_size rows of scaled_housing_data_plus_bias
    # Y_batch is the same rows of the reshaped housing.target
    return X_batch, y_batch

with tf.Session() as sess:
  sess.run(init)
  for epoch in range(n_epochs):
     for batch_index in range(n_batches):
        X_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
        sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
  best_theta = theta.eval()
```

Summary

- We've seen how to do linear regression with TensorFlow
- Features of TensorFlow we learned about:
  - computing gradients with autodiff
  - TensorFlow optimizers, including gradient descent
  - mini-batch gradient descent using placeholders and feed_dict