LINEAR REGRESSION

Edit

Note: This post will only cover the bare essentials of linear regression in order to understand it's impact and use case to tackle more elaborate deep learning applications.

I. OBJECTIVE:

Forward pass involves using our weights with X to determine the predictions y.

$$\hat{y} = \beta_0 + \beta_1 X + \epsilon$$
$$\hat{y} = \text{predicted response}$$
$$\beta = weights$$
$$\epsilon = error$$

Forward Pass:

$$\hat{y} = \beta_0 + \beta_1 X + \epsilon$$
$$[NX1] = [1X1] + [NXD][DX1]$$

The objective is to accurately predict **y** using **X**. The bias and weights are values we need to determine with the objective to minimize the **mean squared error** function:

$$J(\theta) = \frac{1}{2n}\sum_i (\hat{y}_i - y_i)^2$$
$$y = \text{actual y value}$$

II. BACKPROPAGATION:

STEPS:

1. Randomly initiate bias and weights.
2. Forward pass with weights and X to generate predictions y.
3. Calculate L2 loss J.
4. Determine gradient of J with respect to weights.
5. Update the weights based on gradient (which is a step towards decreasing the overall mean squared error (MSE).

$$J(\theta) = \frac{1}{2}\sum_i (\hat{y}_i - y_i)^2$$

$$\frac{\partial J}{\partial \beta_1} = X(\hat{y} - y)$$
$$\frac{\partial J}{\partial \beta_0} = 1(\hat{y} - y)$$
$$\longrightarrow \quad \beta_i = \beta_i - \alpha \frac{\partial J}{\partial \beta_i} J$$

III. REGULARIZATION:

Regularization helps decrease over fitting. Below is L2 regularization. There are many forms of regularization but they all work to reduce overfitting in our models. With L2 regularization, we are penalizing the weights with large magnitudes because we want diffuse weights. Having certain weights with high magnitudes will lead to preferential bias with the inputs and we want the model to work with all the inputs and not just a select few. So applying the L2 penalty allows us to diffuse the weights by decaying them.

$$J(\theta) = \sum_i (\hat{y}_i - y_i)^2 + \frac{\lambda}{2}\sum_k \sum_l W_{kl}^2$$

$$\frac{\partial J}{\partial \beta_1} = X(\hat{y} - y) + \lambda W$$
$$\frac{\partial J}{\partial \beta_1} = 1(\hat{y} - y) + \lambda W$$
$$\longrightarrow \quad \beta_i = \beta_i - \alpha \frac{\partial J}{\partial \beta_i}$$

IV. CODE ANALYSIS:

Our first few lines are the tensorflow and numpy dependencies that we need followed by a few hyperparameters. Learning rate, regularization coefficient should all be deduced empirically by testing across different ranges for optimal performance.

```
import tensorflow as tf
import numpy as np

class parameters():

    def __init__(self):
        self.DATA_LENGTH = 10000
        self.LEARNING_RATE = 1e-10
        self.REG = 1e-10
        self.NUM_EPOCHS =  2000
        self.BATCH_SIZE = 5000
        self.DISPLAY_STEP = 100 # epoch
```

The next part involves creating our data and separating into batches. Here in our dummy example, we generate a range x and get y by factoring in some linear noise to our inputs x. Using our data, we will split it into batches of length batch_size so we can feed many batches simultaneously. Lastly, we will generate all batches in the set for multiple epochs for training.

```
def generate_data(data_length):
    """
    Load the data.
    """
    X = np.array(range(data_length))
    y = 3.657*X + np.random.randn(*X.shape) * 0.33
    return X, y

def generate_batches(data_length, batch_size):
    """
    Create &lt;num_batches&gt; batches from X and y
    """
    X, y = generate_data(data_length)

    # Create batches
    num_batches = data_length // batch_size
    data_X = np.zeros([num_batches, batch_size], dtype=np.
    data_y = np.zeros([num_batches, batch_size], dtype=np.
    for batch_num in range(num_batches):
        data_X[batch_num,:] = X[batch_num*batch_size:(batc
        data_y[batch_num,:] = y[batch_num*batch_size:(batc
        yield data_X[batch_num].reshape(-1, 1), data_y[bat

def generate_epochs(num_epochs, data_length, batch_size):
    """
    Create batches for &lt;num_epochs&gt; epochs.
    """
    for epoch_num in range(num_epochs):
        yield generate_batches(data_length, batch_size)
```

I'm going to assume you have some familiarity with tensorflow but if not check out this basics **video**. First we will have placeholders for our inputs. The shape is [None, 1] where the None is batch_size. In later examples you will see we will use shape=[None, None] to have complete freedom with batch_size and seq_len.

Next we will set out weights W and bias b. Our prediction will just be the forward pass with XW+b. We will then compute the MSE with L2 regularization and use this cost as the quantity to minimize using our optimizer.

We also have a **step()** function inside the model class. This will take in a batch of inputs and do one step of training.

We also have a **create_model()** function that will take in a tf session and a few parameters to initialize the model. We pass in session because in later example you will see that we want to save the model after training and reload later on. Here is the location that we will reload saved models if we have any.

```
class model(object):
    """
    Train the linear model to minimize L2 loss function.
    """

    def __init__(self, learning_rate, reg):
        # Inputs
        self.X = tf.placeholder(tf.float32, [None, 1], "X"
        self.y = tf.placeholder(tf.float32, [None, 1], "y"

        # Set model weights
        with tf.variable_scope('weights'):
            self.W = tf.Variable(tf.truncated_normal([1,1]
            self.b = tf.Variable(tf.truncated_normal([1,1]

        # Forward pass
        self.prediction = tf.add(tf.matmul(self.X, self.W)

        # L2 loss
        self.cost = tf.reduce_mean(tf.pow(self.prediction-

        # Gradient descent (backprop)
        self.optimizer = tf.train.GradientDescentOptimizer

    def step(self, sess, batch_X, batch_y):

        input_feed = {self.X:batch_X, self.y:batch_y}
        output_feed = [self.prediction,
                       self.cost,
                       self.optimizer,
                       self.W,
                       self.b]

        outputs = sess.run(output_feed, input_feed)

        return outputs[0], outputs[1], outputs[2], outputs

```

```
def create_model(sess, FLAGS):
    linear_model = model(FLAGS.LEARNING_RATE, FLAGS.REG)
    sess.run(tf.initialize_all_variables())
    return linear_model
```

Lastly, we will train for several epochs. Note that we chose an arbitrary number of epochs but later on in this blog, we will explore empirical techniques to use to determine when to stop training (gradient norm, etc.).

```
def train(FLAGS):

    with tf.Session() as sess:

        # Create the model
        model = create_model(sess, FLAGS)

        for epoch_num, epoch in enumerate(generate_epochs(
            for simult_batch_num, (input_X, labels_y) in e
                prediction, training_loss, _, W, b = model

            # Display
            if epoch_num%FLAGS.DISPLAY_STEP == 0:
                print "EPOCH %i: \n Training loss: %.3f, W
                    epoch_num, training_loss, W, b)

if __name__ == '__main__':

    FLAGS = parameters()
    train(FLAGS)
```

V. RESULTS:

```
EPOCH 0:
 Training loss: 2228963328.000, W: 0.122, b:-0.007
EPOCH 100:
 Training loss: 2534575.750, W: 3.538, b:-0.006
EPOCH 200:
 Training loss: 2884.403, W: 3.653, b:-0.006
EPOCH 300:
 Training loss: 3.820, W: 3.657, b:-0.006
EPOCH 400:
 Training loss: 0.541, W: 3.657, b:-0.006
```

results: weights drop but the bias doesn't seem to change much from initial starting value. It might be better to combine bias with weights and append a 1 to all Xs.

VI. RAW CODE:

GitHub Repo (Updating all repos, will be back up soon!)

Posted in: *Regression*
Tagged: *Tutorials*

← *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*

Logistic Regression →

LEAVE A REPLY