mnist_tensorflow

August 16, 2017

1 MNIST FOR BEGINNERS - TENSORFLOW

1.1 Download and read in the MNIST dataset

```
In [6]: from tensorflow.examples.tutorials.mnist import input_data # must have tensorflow instal
        mnist = input_data.read_data_sets("MNIST_data/", one_hot=True) # one_hot is a specific e
```

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

1.2 The MNIST dataset

The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits. The MNIST data is split into three parts: 55,000 data points of training data (mnist.train), 10,000 points of test data (mnist.test), and 5,000 points of validation data (mnist.validation).

Each image is 28 pixels by 28 pixels, this can be input as a flattned vector of 784 features.

Flattening the data throws away information about the 2D structure of the image. Isn't that bad? Well, the best computer vision methods do exploit this structure (*Convolutional Neural Networks*), but the simple method we will be using here, a softmax regression won't.

1.3 Softmax Regression

Since every image will consist of a digit 0-9 learning the MNIST dataset is a type of classification with 10 classes.

Softmax regression is a generalization of logistic regression that we can use for multi-class classification (under the assumption that the classes are mutually exclusive). In contrast, we use the (standard) Logistic Regression model in binary classification tasks.

When learning we don't want to just give yes/or no answers for the classification since that would be harder to learn precisely. We want to be able to look at an image and give the probabilities for it being each digit. For example, our model might look at a picture of a nine and be 80% sure it's a nine, but give a 5% chance to it being an eight (because of the top loop) and a bit of probability to all the others because it isn't 100% sure.

This is a classic case where a softmax regression is a natural, simple model. If you want to assign probabilities to an object being one of several different things, softmax is the thing to do, because softmax gives us a list of values between 0 and 1 that add up to 1.

Recall the softmax function:

In mathematics, the softmax function, or normalized exponential function,[1]:198 is a generalization of the logistic function that "squashes" a K-dimensional vector $\mathbf{z}$ of arbitrary real values to a K-dimensional vector $\sigma(\mathbf{z})$ of real values in the range [0, 1] that add up to 1. The function is given by:

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for j = 1, , K.}$$

A softmax regression has two steps: first we add up the evidence of our input being in certain classes, and then we convert that evidence into probabilities.

To tally up the evidence that a given image is in a particular class, we do a weighted sum of the pixel intensities. The weight is negative if that pixel having a high intensity is evidence against the image being in that class, and positive if it is evidence in favor.

We also add some extra evidence called a bias. Basically, we want to be able to say that some things are more likely independent of the input.

I.e softmax is our activation function, so for a neuron y with k inputs we get:

$$y = softmax(\sum_j W_{i,j} \cdot k_j + b_i)$$

You can think of softmax as a function that first exponents its input and then normalizes the inputs such that the outputs sum up to one and can be considered a valid probability.

We can vectorize the implementation of the neural network to be similar as the computer will compute it:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$

Vectorized network

We can also simplify the node expression now when we have vectorized to the following:

$$y = softmax(Wx + b)$$

where W is a weight matrix, with one column per layer in the network, x is a input vector of features, b is a bias vector, one per input and y is a output classified vector with probabilities.

1.4 Import TensorFlow library

```
In [7]: import tensorflow as tf
```

1.5 TensorFlow computations and variables

In tensorflow we describe a graph of interacting operations. TensorFlow programs are usually structured into a construction phase, that assembles a graph, and an execution phase that uses a session to execute ops in the graph.

For example, it is common to create a graph to represent and train a neural network in the construction phase, and then repeatedly execute a set of training ops in the graph in the execution phase.

Here below we create a placeholder variable, a placeholder variable is a variable that we will input to TensorFlow when we ask it to run a computation. In the computation for this neural network we want to be able to input any number of images, each image is a 784 (28×28) vector, so we can represent a collection of images as a 2D matrix.

```
In [8]: x = tf.placeholder(tf.float32, [None, 784]) # x placeholder for set of input images
```

For weights and biases (the parameters of our model) we will use variables, variables are **modifiable** tensors that will live inside TensorFlow's graph of operations and can be accessed and updated/modified by the operations.

```
In [9]: W = tf.Variable(tf.zeros([784, 10])) # Modifiable weights matrix, initialized to tensor
        b = tf.Variable(tf.zeros([10])) # Modifiable bias vector, initialized to tensor of zeros
```

1.6 Implementing the model

Now we implement the model, i.e we define the computational graph for learning the dataset:

```
In [11]: y = tf.nn.softmax(tf.matmul(x, W) + b) #x = input, W = weights, b = bias. softmax = act
```

As you see in the model we multiply input vector x by weight matrix W by using tensorflows matrix operation and then we add the bias vector and finally we input this to the softmax function which also is part of tensorflow library.

1.7 Training

In order to train we must define loss-function/error-function.

One very common, very nice function to determine the loss of a model is called "cross-entropy." We'll use that one, which is defined as follows:

$$H_{y'}(y) = -\sum_i y'_i \log(y_i)$$

Where y is our predicted probability distribution, and y is the true distribution (the one-hot vector with the digit labels). In some rough sense, the cross-entropy is measuring how inefficient our predictions are for describing the truth.

To implement cross-entropy we need to first add a new placeholder to input the correct answers, which will be a matrix of output vectors, the dimension of the matrix is determined by the number of training examples.

```
In [14]: y_ = tf.placeholder(tf.float32, [None, 10]) # placeholder for correct answers from trai
```

To implement the cross-entropy function in tensorflow is this easy:

```
In [15]: cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
```

First, tf.log computes the logarithm of each element of y. Next, we multiply each element of y_ with the corresponding element of tf.log(y). Then tf.reduce_sum adds the elements in the second dimension of y, due to the reduction_indices=[1] parameter. Finally, tf.reduce_mean computes the mean over all the examples in the batch.

Now with loss-function defined we let Tensorflow train it through computations by using the backpropagation algorithm.

TensorFlow is defines such that backpropagation is only considered as the step that declares how to find the partial derivative of each weight affects the error, i.e with respect to the loss function to minimize. Then you separately choose which optimizer to use to modify the variables to reduce the loss.

```
In [16]: train_step = tf.train.GradientDescentOptimizer(0.05).minimize(cross_entropy) # learning
```

What TensorFlow actually does here, behind the scenes, is to add new operations to your graph which implement backpropagation and gradient descent. Then it gives you back a single operation which, when run, does a step of gradient descent training, slightly tweaking your variables to reduce the loss.

We can now launch the model in an InteractiveSession (session for use in interactive contexts such as a shell or notebook):

```
In [17]: sess = tf.InteractiveSession()
```

Just due to how TensorFlow library/framework is defined we first have to create an operation just to initialize the variables and models we just defined and run that initialization operation:

```
In [18]: tf.global_variables_initializer().run()
```

Now! Let's train the network for 1000 steps, i.e 1000 computations of gradient and taking steps.

```
In [19]: for _ in range(1000):
             batch_xs, batch_ys = mnist.train.next_batch(100)
             sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
```

For each step of the loop above, we first get the next "batch" which consists of 100 training examples (i.e we dont use all training examples before calculating loss and gradient, i.e stochastic gradient descent), then we run in our session the train_step (computation we defined earlier) and take as input the batch and and the supervised examples. Remember that x and y_ are placeholders which here gets replaced with the batch data, and as we run the computation, the y placeholder will get replaced with the predicted output.

Using small batches of random data is called stochastic training -- in this case, stochastic gradient descent. Ideally, we'd like to use all our data for every step of training because that would give us a better sense of what we should be doing, but that's expensive. So, instead, we use a different subset every time. Doing this is cheap and has much of the same benefit.

1.8 Evaluating our model

tf.argmax is an extremely useful function which gives you the index of the highest entry in a tensor along some axis. For example, tf.argmax(y,1) is the label our model thinks is most likely for each

input, while tf.argmax(y_,1) is the correct label. We can use tf.equal to check if our prediction matches the truth.

Remember that y and y_ are matrices with many examples (the latest batch of training), i.e 100 examples. So the code below will give us a list of booleans (not a list per se since it is a tensor, but conceptually):

```
In [24]: correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
```

To determine what fraction are correct, we cast to floating point numbers and then take the mean. For example, [True, False, True, True] would become [1,0,1,1] which would become 0.75.

```
In [26]: accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
         print(sess.run(accuracy, feed_dict={x: mnist.test.images, y_: mnist.test.labels}))
```

```
0.902
```

This tells us that our prediction rate is $\approx$ 90% correct