

TensorFlow: Static Graphs

A fully-connected ReLU network with one hidden layer and no biases, trained to predict y from x by minimizing squared Euclidean distance.

This implementation uses basic TensorFlow operations to set up a computational graph, then executes the graph many times to actually train the network.

One of the main differences between TensorFlow and PyTorch is that TensorFlow uses static computational graphs while PyTorch uses dynamic computational graphs.

In TensorFlow we first set up the computational graph, then execute the same graph many times.

```
import tensorflow as tf
import numpy as np

# First we set up the computational graph:

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create placeholders for the input and target data; these will be filled
# with real data when we execute the graph.
x = tf.placeholder(tf.float32, shape=(None, D_in))
y = tf.placeholder(tf.float32, shape=(None, D_out))

# Create Variables for the weights and initialize them with random data.
# A TensorFlow Variable persists its value across executions of the graph.
w1 = tf.Variable(tf.random_normal((D_in, H)))
w2 = tf.Variable(tf.random_normal((H, D_out)))

# Forward pass: Compute the predicted y using operations on TensorFlow Tensors.
# Note that this code does not actually perform any numeric operations; it
# merely sets up the computational graph that we will later execute.
h = tf.matmul(x, w1)
h_relu = tf.maximum(h, tf.zeros(1))
y_pred = tf.matmul(h_relu, w2)

# Compute loss using operations on TensorFlow Tensors
loss = tf.reduce_sum((y - y_pred) ** 2.0)

# Compute gradient of the loss with respect to w1 and w2.
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])


# Update the weights using gradient descent. To actually update the weights
# we need to evaluate new_w1 and new_w2 when executing the graph. Note that
# in TensorFlow the the act of updating the value of the weights is part of
# the computational graph; in PyTorch this happens outside the computational
# graph.
learning_rate = 1e-6
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)

# Now we have built our computational graph, so we enter a TensorFlow session to
# actually execute the graph.
with tf.Session() as sess:
    # Run the graph once to initialize the Variables w1 and w2.
    sess.run(tf.global_variables_initializer())

    # Create numpy arrays holding the actual data for the inputs x and targets
    # y
    x_value = np.random.randn(N, D_in)
    y_value = np.random.randn(N, D_out)
    for _ in range(500):
        # Execute the graph many times. Each time it executes we want to bind
        # x_value to x and y_value to y, specified with the feed_dict argument.
        # Each time we execute the graph we want to compute the values for loss,
        # new_w1, and new_w2; the values of these Tensors are returned as numpy
        # arrays.
        loss_value, _, _ = sess.run([loss, new_w1, new_w2],
                                    feed_dict={x: x_value, y: y_value})
    print(loss_value)
```

Total running time of the script: (0 minutes 0.000 seconds)


 Download Python source
code: `tf_two_layer_net.py`

 Download Jupyter notebook:
`tf_two_layer_net.ipynb`

Gallery generated by Sphinx-Gallery

 Previous

Next 



0.4.0

Search docs

BEGINNER TUTORIALS

Deep Learning with PyTorch: A 60 Minute Blitz

PyTorch for former Torch users

Learning PyTorch with Examples

Tensors

Autograd

nn module

Examples

Tensors

Autograd

PyTorch: Tensors and autograd

PyTorch: Defining new autograd functions

TensorFlow: Static Graphs

nn module

Transfer Learning tutorial

Data Loading and Processing Tutorial

Deep Learning for NLP with Pytorch

INTERMEDIATE TUTORIALS

Classifying Names with a Character-Level RNN

Generating Names with a Character-Level RNN

Translation with a Sequence to Sequence Network and Attention

Reinforcement Learning (DQN) tutorial