Deep Neural Networks Using Tensorflow

Amirkoushyar Ziabari

aziabari@purdue.edu

June 2018
Outline

- Deep Learning Frameworks
- Tensors
- What is Tensorflow (TF)?
- Tensorflow Structure
- MNIST data classification (CPU)
  - Example 1: Multi-layer Perception
  - Example 2: CNN
# Deep Learning Frameworks

<table>
<thead>
<tr>
<th>Framework</th>
<th>Distributed Execution</th>
<th>Architecture Optimization</th>
<th>Visualization</th>
<th>Community Support</th>
<th>Portability</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tensorflow</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>• Google, wide usage, ecosystem and community support • Visualization is superior</td>
</tr>
<tr>
<td>Pytorch</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>• Facebook • Easy to use &amp; technically impressive</td>
</tr>
<tr>
<td>CNTK</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>• Microsoft • Licensig issues</td>
</tr>
<tr>
<td>MXNet</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>• Behind other frameworks in DE, and lacks details</td>
</tr>
<tr>
<td>Caffe2</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>• Facebook • Still moving</td>
</tr>
<tr>
<td>Caffe</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>• Facebook • Lacks future community support</td>
</tr>
<tr>
<td>Theano</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>• University of Montreal • Lacks future community support</td>
</tr>
</tbody>
</table>

Tensors

- Tensors are the standard way of representing data in Tensorflow (deep learning)
- Tensors are multidimensional arrays, an extension of matrices to data with higher dimensions
<table>
<thead>
<tr>
<th>Rank</th>
<th>Math Entity</th>
<th>Python Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Scalar (magnitude only)</td>
<td><code>s = 483</code></td>
</tr>
<tr>
<td>1</td>
<td>Vector (magnitude and direction)</td>
<td><code>v = [1.1, 2.2, 3.3]</code></td>
</tr>
<tr>
<td>2</td>
<td>Matrix (table of numbers)</td>
<td><code>m = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]</code></td>
</tr>
<tr>
<td>3</td>
<td>3-Tensor (cube of numbers)</td>
<td><code>t = [[[2], [4], [6]], [[8], [10], [12]], [[14], [16], [18]]]</code></td>
</tr>
<tr>
<td>n</td>
<td>n-Tensor (you get the idea)</td>
<td><code>....</code></td>
</tr>
</tbody>
</table>
In addition to dimensionality, Tensors have different data types:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Python type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DT_FLOAT</td>
<td>tf.float32</td>
<td>32 bits floating point.</td>
</tr>
<tr>
<td>DT_DOUBLE</td>
<td>tf.float64</td>
<td>64 bits floating point.</td>
</tr>
<tr>
<td>DT_INT8</td>
<td>tf.int8</td>
<td>8 bits signed integer.</td>
</tr>
<tr>
<td>DT_INT16</td>
<td>tf.int16</td>
<td>16 bits signed integer.</td>
</tr>
<tr>
<td>DT_INT32</td>
<td>tf.int32</td>
<td>32 bits signed integer.</td>
</tr>
<tr>
<td>DT_INT64</td>
<td>tf.int64</td>
<td>64 bits signed integer.</td>
</tr>
<tr>
<td>DT_UINT8</td>
<td>tf.uint8</td>
<td>8 bits unsigned integer.</td>
</tr>
<tr>
<td>DT_STRING</td>
<td>tf.string</td>
<td>Variable length byte arrays. Each element of a tensor is a byte array.</td>
</tr>
<tr>
<td>DT_BOOL</td>
<td>tf.bool</td>
<td>Boolean.</td>
</tr>
</tbody>
</table>
What is Tensorflow?

- Tensorflow (TF): a python library to implement deep networks
  - Very simple to install on all operating systems (tensorflow.org/install)
  - Pycharm, ipython, etc can be used to run TF on windows
  - In Tensorflow, computation is approached as a dataflow graph

X, W, b: Tensors
Relu, Add, Matmul: functions
**TensorFlow Structure**

TensorFlow core programs consist of two discrete sections:

- Building a computational graph
- Running a computational graph

A computational graph is a series of TensorFlow operations arranged into a graph of nodes.

```python
import tensorflow as tf

a = tf.constant(5.0, tf.float32)
b = tf.constant(6.0)
c = a*b
```

**Build a computational graph**

**Run the computational graph**
Goal

- Develop a Neural Network To Classify MNIST DATA
Training Process

1. Start
2. Read the Dataset
3. Define features and labels
4. Encode The Dependent variable
5. Pre-processing of dataset
   - Divide the dataset into two parts for training and testing
7. Train the model
8. Implement the model
9. TensorFlow data structure for holding features, labels etc..
10. Reduce MSE (actual output - desired output)
11. Make prediction on the test data
12. End
13. Repeat the process to decrease the loss
Code Structure

1. Read the input data; define parameters, constants.
2. Define input/target size, type
   Assign space for input/target
3. Define weights and biases
4. Define and construct the model
   (e.g. Convolutional Neural Net)
5. Define loss function
6. Choose optimization technique
7. Define Training operation
8. Define Initialization operation
9. Define events logs and saving operations
10. Define Session and run initialization
11. Train the model (run the training op.)
    Print outputs, Save (or restore) model and events logs
Multi-layer Perception

Neural Network Overview

\[ L_1 = W_1 \cdot x + b_1 \]
\[ L_2 = W_2 \cdot x + b_2 \]
\[ \text{Out} = W_{\text{out}} \cdot x + b_{\text{out}} \]
**MNIST Dataset Overview**

This example is using MNIST handwritten digits. The dataset contains 60,000 examples for training and 10,000 examples for testing. The digits have been size-normalized and centered in a fixed-size image (28x28 pixels) with values from 0 to 1. For simplicity, each image has been flattened and converted to a 1-D numpy array of 784 features (28*28).

# Import Tensorflow Library

```python
import tensorflow as tf
```

# You may add other libraries as needed
# such as numpy, matplotlib, etc
Step 1

# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

# Network Parameters and constants
learning_rate = 0.001 #learning rate or step size for optimization
batch_size = 100 # batch size to input the in batches
display_step = 1
model_path = "EventsLogs\model.ckpt"
n_hidden_1 = 256 # 1st layer number of features
n_hidden_2 = 256 # 2nd layer number of features
n_input = 784 # MNIST data input (img shape: 28*28)
n_classes = 10 # MNIST total classes (0-9 digits)
### One-hot encoding

<table>
<thead>
<tr>
<th>Categorical Feature</th>
<th>f1</th>
<th>f2</th>
<th>f3</th>
<th>f4</th>
<th>f5</th>
<th>f6</th>
<th>f7</th>
<th>f8</th>
<th>f9</th>
<th>f10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
# tf Graph input: Placeholders

```python
x = tf.placeholder("float", [None, n_input])
y = tf.placeholder("float", [None, n_classes])
```
Step 3

# Store layers weights & biases
weights = {
    'w1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
    'w2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, n_classes]))
}

biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}
Model for Step 4!

\[ L_1 = W_1 \times x + b_1 \]

\[ L_2 = W_2 \times L_1 + b_2 \]

\[ \text{out} = W_{\text{out}} \times L_2 + b_{\text{out}} \]
# Create model

def multilayer_perceptron(x, weights, biases):
    # Hidden layer
    layer_1 = tf.add(tf.matmul(x, weights['w1']), biases['b1'])
    layer_2 = tf.add(tf.matmul(layer_1, weights['w2']), biases['b2'])
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    return out_layer

# Construct model
pred = multilayer_perceptron(x, weights, biases)
Step 5

# Define loss function
loss =
tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=pred, labels=y))

\[ n : \text{number of batches} \]
\[ \hat{y} = \text{softmax}(\text{logits}) \]
\[ y = \text{labels} \]
\[ \text{cross}_\text{entropy} = - \sum_i y_i \log(\hat{y}_i) \]
\[ \text{loss} = \frac{1}{n} \sum_n \text{cross}_\text{entropy} \]
# Choose the optimizer

```python
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
```
# Train operation
Train_op = optimizer.minimize(loss)
# Initialize the variables (i.e. assign their default value)

```
init = tf.global_variables_initializer()
```
# 'Saver' op to save and restore all the variables

saver = tf.train.Saver()
Step 10

```python
with tf.Session() as sess:
    # Run the initializer
    sess.run(init)
```
for epoch in range(3):
    avg_loss = 0.
    total_batch = int(mnist.train.num_examples/batch_size)
    # Loop over all batches
    for i in range(total_batch):
        batch_x, batch_y = mnist.train.next_batch(batch_size)

        _, c = sess.run([Train_op, loss], feed_dict={x: batch_x,
                                                   y: batch_y})

        # Compute average loss
        avg_loss += c / total_batch
# Display logs per epoch step

```python
if epoch % display_step == 0:
    print("Epoch:", '%04d' % (epoch+1), "loss=", \
          '{:.9f}'.format(avg_loss))
print("First Optimization Finished!")
```
Step 11, cont

# Test model
correct_prediction = tf.equal(tf.argmax(tf.nn.softmax(pred), 1),
                         tf.argmax(y, 1))
# Calculate accuracy
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
print("Accuracy:", accuracy.eval({x: mnist.test.images, y: mnist.test.labels})))
Step 11, save

# Save model weights to disk
save_path = saver.save(sess, model_path)
print("Model saved in file: %s" % save_path)
Assignment 1

- Run the code and write down the accuracy

Change the code so that:

- Define a placeholder to enter the learning rate
- Add a 3rd fully connected layer with 128 neurons
  - Weights are initialized with tf.truncated_normal initializer with stddev=0.5
  - Biases are initialized with tf.constant(0.1, shape=[128])

- Run the code and write down the accuracy

  - Add a session, and restore the current checkpoint and run 7 more epochs
    - saver.restore(sess, model_path) will help restore the checkpoint

- Run the code and write down the accuracy
Add Relu To The network

# Create model
def multilayer_perceptron(x, weights, biases):
    # Hidden layer with RELU activation
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    layer_1 = tf.nn.relu(layer_1)
    # Hidden layer with RELU activation
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    layer_2 = tf.nn.relu(layer_2)
    # Output layer with linear activation
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    return out_layer
Run the code with Relu and write down the accuracy
Change the code in assignment 1 and add Relu
Run the new code and write down the accuracy
**Algorithm 1:** Batch Normalizing Transform, applied to activation $x$ over a mini-batch.
def multilayer_perceptron(x, weights, biases):
    # Hidden layer with RELU activation and BN
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    batch_mean1, batch_var1 = tf.nn.moments(layer_1, [0])
    scale1 = tf.Variable(tf.ones([n_hidden_1]))
    beta1 = tf.Variable(tf.zeros([n_hidden_1]))
    BN1 = tf.nn.batch_normalization(layer_1, batch_mean1, batch_var1, beta1, scale1, epsilon)
    layer_1f = tf.nn.relu(BN1)
    # Hidden layer with RELU activation and BN
    layer_2 = tf.add(tf.matmul(layer_1f, weights['h2']), biases['b2'])
    batch_mean2, batch_var2 = tf.nn.moments(layer_2, [0])
    scale2 = tf.Variable(tf.ones([n_hidden_2]))
    beta2 = tf.Variable(tf.zeros([n_hidden_2]))
    BN2 = tf.nn.batch_normalization(layer_2, batch_mean2, batch_var2, beta2, scale2, epsilon)
    layer_2f = tf.nn.relu(BN2)
    # Output layer with linear activation
    out_layer = tf.matmul(layer_2f, weights['out']) + biases['out']
    return out_layer
Assignment 3

- Run the code with Relu and BN and write down the accuracy
- Change the code in assignment 2 and add BN
- Run the new code and write down the accuracy
PART 2:
CNN for MNIST Classification
# Create some wrappers for simplicity

def conv2d(x, W, b, strides=1):
    # Conv2D wrapper, with bias and relu activation
    x = tf.nn.conv2d(x, W, strides=[1, strides, strides, 1], padding='SAME')
    x = tf.nn.bias_add(x, b)
    return tf.nn.relu(x)
Max Pooling

```python
def maxpool2d(x, k=2):
    # MaxPool2D wrapper
    return tf.nn.max_pool(x, ksize=[1, k, k, 1], strides=[1, k, k, 1],
                          padding='SAME')
```
# Store layers weights & biases
weights = {
    # 5x5 conv, 1 input, 32 outputs
    'wc1': tf.Variable(tf.random_normal([5, 5, 1, 32])),
    # 5x5 conv, 32 inputs, 64 outputs
    'wc2': tf.Variable(tf.random_normal([5, 5, 32, 64])),
    # fully connected, 7*7*64 inputs, 1024 outputs
    'wd1': tf.Variable(tf.random_normal([7*7*64, 1024])),
    # 1024 inputs, 10 outputs (class prediction)
    'out': tf.Variable(tf.random_normal([1024, n_classes]))
}

biases = {
    'bc1': tf.Variable(tf.random_normal([32])),
    'bc2': tf.Variable(tf.random_normal([64])),
    'bd1': tf.Variable(tf.random_normal([1024])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}
**CNN Model**

\[ Conv1 = \text{Relu}(W_{c1} ** x + b_{c1}) \]

\[ Conv1 = \text{maxpool}(\text{Conv1}) \]

\[ Conv2 = \text{Relu}(W_{c2} ** \text{Conv1} + b_{c2}) \]

\[ Conv2 = \text{maxpool}(\text{Conv2}) \]

\[ fc1 = \text{Relu}(W_{d1} * \text{Conv2}(:) + b_{d1}) \]

\[ fc1 = \text{dropout}(fc1) \]

\[ out = W_{out} * fc1 + b_{out} \]
Figure 1: Dropout Neural Net Model. **Left:** A standard neural net with 2 hidden layers. **Right:** An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.
# Create model
def conv_net(x, weights, biases, dropout):
    # dropout: With probability keep_prob, outputs the input element scaled up by
    # 1 /keep_prob, otherwise outputs 0. The scaling is so that the expected sum is
    # unchanged. (to reduce overfitting)
    # MNIST data input is a 1-D vector of 784 features (28*28 pixels)
    # Reshape to match picture format [Height x Width x Channel]
    # Tensor input become 4-D: [Batch Size, Height, Width, Channel]
    x = tf.reshape(x, shape=[-1, 28, 28, 1])  # -1 is used to infer the shape (here number of
    # batches)
    # Convolution Layer
    conv1 = conv2d(x, weights['wc1'], biases['bc1'])
    # Max Pooling (down-sampling)
    conv1 = maxpool2d(conv1, k=2)
    # Convolution Layer
    conv2 = conv2d(conv1, weights['wc2'], biases['bc2'])
    # Max Pooling (down-sampling)
    conv2 = maxpool2d(conv2, k=2)
    # Fully connected layer
    # Reshape conv2 output to fit fully connected layer input
    fc1 = tf.reshape(conv2, [-1, weights['wd1'].get_shape().as_list()[0]])
    fc1 = tf.add(tf.matmul(fc1, weights['wd1']), biases['bd1'])
    fc1 = tf.nn.relu(fc1)
    # Apply Dropout
    fc1 = tf.nn.dropout(fc1, dropout)
    # Output, class prediction
    out = tf.add(tf.matmul(fc1, weights['out']), biases['out'])
    return out
Run the code and note the accuracy
  A. Add Batch Normalization (BN)

Run the code and note the accuracy
  B. Add a convolutional layer and BN together

Run the code and note the accuracy
Resources

- https://www.tensorflow.org/
- Github
  - https://github.com/aymericdamien/TensorFlow-Examples
  - And many others
- Stanford Tensorflow for Deep Learning Research
  - http://web.stanford.edu/class/cs20si/
- Youtube
  - https://www.youtube.com/watch?v=yX8KuPZCAMo
  - And many others!
- Kaggle.com
Bonus
# Copyright 2015 The TensorFlow Authors. All Rights Reserved.

# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
# http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
# """Functions for downloading and reading MNIST data."""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

# pylint: disable=unused-import
import gzip
import os
import tempfile

import numpy
from six.moves import urllib
from six.moves import xrange  # pylint: disable=redefined-builtin
import tensorflow as tf
from tensorflow.contrib.learn.python.learn.datasets.mnist import read_data_sets
# pylint: enable=unused-import