TensorFlow APIs for Image Classification

TensorFlow is a powerful open-source library for Deep Learning, developed at Google. It became available to the general public in late 2015, and includes such features as:

- common neural network elements such as backpropagation, convolutional layers, ReLU, softmax, etc.
- utilization of GPU for high-speed processing.
- visualization tools for monitoring learning progress.
- support for a variety of languages such as Python, C++, Java, and Rust.

TensorFlow has already enjoyed widespread use in a variety of machine learning applications such as image recognition, real-time OCR and translation, face-tagging (Facebook), self-driving cars, voice recognition, computers learning to play video games, cancer detection, sentiment analysis, etc.

TensorFlow programming involves the definition and use of “tensors”, which can be thought of as generalized vectors or matrices. Since most computational effort in neural networks involves large numbers of matrix operations (such as dot products), TensorFlow is designed to enable specifying a wide variety of programmer-defined paths through layers of matrix operations. The learning curve on raw TensorFlow coding is a bit steep.

As such, a number of high-level APIs have been developed to facilitate using TensorFlow in common scenarios. Also, several existing machine learning APIs have been integrated to also work with TensorFlow. So, there is a wide array of options for those wanting to use TensorFlow. Three widely-used APIs are:

- Keras
- TFLearn
- TFSlim

This document describes setting up TFLearn to learn an image classification task.

Installing Tensorflow and TFLearn

Complete instructions for installing Python and TensorFlow on a number of operating systems can be found on the TensorFlow website: https://www.tensorflow.org/install/

This handout will assume the use of Python 3.6 and TensorFlow 1.3 on Windows.

You may (or may not) need to add TensorFlow to your PYTHONPATH. If you do, you’ll need to find where TensorFlow is installed on your machine. On my Windows laptop, I added the following to my PYTHONPATH:

;C:\Users\gordonvs\AppData\Local\Programs\Python\Python36\Lib\site-packages\tensorflow

After installing TensorFlow, it is recommended to test it using the MNIST dataset, as described in the first two or three “getting started” tutorials on the TensorFlow website: https://www.tensorflow.org/get_started/

MNIST is a huge database containing the digits 0-9 handwritten in thousands of different ways. MNIST is often used to demonstrate/test an algorithm’s ability to learn to recognize handwritten characters.

Installing TFLearn is really easy if you have already installed Python and TensorFlow, especially if your Python installation includes “pip”: pip install tflearn

Complete installation instructions, and other TFLearn details, are on the TFLearn website: http://tflearn.org/

This TFLearn walkthrough assumes that the reader understands the previous notes 11A that describe convolutional neural networks. It is also advised to watch this Andrej Karpathy (Stanford/Tesla) lecture: https://www.youtube.com/watch?v=GYGYnspV230
The Task ---

Our goal in this example is to teach a convolutional neural network to differentiate between fishes and horses. In particular, we want to be able to show the network a picture, and have it correctly tell us whether that is a picture of a fish or a horse. Our steps are as follows:

1. collect images to serve as training data, and additional images to serves as validation (testing) data.
2. prepare/standardize the images so they can be used to train (and test) a CNN.
3. prepare a TFlearn Python program to learn our training data.
4. run the training program, which causes TensorFlow to build and train a CNN on our training data.
5. prepare a TFlearn Python program to test our trained network.
6. run the testing program, which causes TensorFlow to report on the network’s ability to generalize.

(additional) use TensorBoard to visualize the progress of learning, either during or after training.

Step 1 – collecting images

Since we will be using supervised learning, we need training data. For this task, that means obtaining lots of JPG images of fishes, and lots of JPG images of horses. Most of these images will be used as training data, but we will also set some of them aside for use as testing data.

In this example, 24 pictures of fishes and 24 pictures of horses were downloaded from Google Images. They were of various resolutions and aspect ratios. Some examples looked like this:

![Fish Images](image1.png) ![Horse Images](image2.png)

40 images will be used for training (20 fishes, 20 horses), and 8 will be used for testing (4 fishes, 4 horses). Note that this isn't likely to be enough training cases for effective generalization. But it will allow us to demonstrate the steps and the framework.

Step 2 – Preparing the images for training

Convolutional neural networks are generally designed to work with images that are perfectly square, and they all must have the same resolution. There are many tools for doing this. In this example, a web-based tool at www.resizemypicture.com was used, that can resize up to 5 images at once. Using this tool, we converted each of our pictures to 100x100 square images. The four images above then looked like this:

![Resized Images](image3.png)

The 20 training examples of fish were put in a folder called: train/SUBFOLDER_0
The 20 training examples of horses were put in a folder called: train/SUBFOLDER_1
The 4 testing examples of fish were put in a folder called: validate/SUBFOLDER_0
The 4 testing examples of horses were put in a folder called: validate/SUBFOLDER_1
Step 3 – Preparing the TFLearn Python training program

Suppose we wish to use the following AlexNET-style CNN architecture to learn our training data:

The following Python program ("deepneuralnet.py") defines the network and expected input format:

```python
import tflearn
from tflearn.layers.core import input_data, dropout, fully_connected
from tflearn.layers.conv import conv_2d, max_pool_2d
from tflearn.layers.estimator import regression
from tflearn.metrics import Accuracy

acc = Accuracy()
network = input_data(shape=[None, 100, 100, 3])

# Conv layers ------------------------------------
network = conv_2d(network, 64, 3, strides=1, activation='relu')
network = max_pool_2d(network, 2, strides=2)
network = conv_2d(network, 64, 3, strides=1, activation='relu')
network = max_pool_2d(network, 2, strides=2)
network = conv_2d(network, 64, 3, strides=1, activation='relu')
network = conv_2d(network, 64, 3, strides=1, activation='relu')
network = conv_2d(network, 64, 3, strides=1, activation='relu')
network = max_pool_2d(network, 2, strides=2)

# Fully Connected Layers -------------------------
network = fully_connected(network, 1024, activation='tanh')
network = dropout(network, 0.5)
network = fully_connected(network, 1024, activation='tanh')
network = dropout(network, 0.5)
network = fully_connected(network, 2, activation='softmax')

network = regression(network, optimizer='momentum',
                      loss='categorical_crossentropy',
                      learning_rate=0.001, metric=acc)
model = tflearn.DNN(network)
```

The following Python program ("train.py") reads the training and validation data, and initiates the training:

```python
import deepneuralnet as net
import numpy as np
from tflearn.data_util import image_preloader
model = net.model

X, Y = image_preloader(target_path='./train', image_shape=(100, 100),
                        mode='folder', grayscale=False, categorical_labels=True, normalize=True)
X = np.reshape(X, (-1, 100, 100, 3))

W, Z = image_preloader(target_path='./validate', image_shape=(100, 100),
                        mode='folder', grayscale=False, categorical_labels=True, normalize=True)
W = np.reshape(W, (-1, 100, 100, 3))

model.fit(X, Y, n_epoch=250, validation_set=(W, Z), show_metric=True)
model.save('./ZtrainedNet/final-model.tfl')
```
Step 4 – run the training program

We run the training program by opening a command window and typing:  
```
python train.py
```

As training proceeds, TFLearn gives a progress report similar to the following:

```
Training Step: 1  loss: 0.00000 - acc: 0.0000 | val_loss: 0.69784 - val_acc: 0.5000
Training Step: 2  loss: 0.63188 - acc: 0.3937 | val_loss: 0.69763 - val_acc: 0.5000
Training Step: 3  loss: 0.68900 - acc: 0.4551 | val_loss: 0.69734 - val_acc: 0.5000
... 
Training Step: 248 loss: 0.57213 - acc: 0.7379 | val_loss: 0.50747 - val_acc: 0.7500
Training Step: 249 loss: 0.56288 - acc: 0.7391 | val_loss: 0.50415 - val_acc: 0.8750
Training Step: 250 loss: 0.58919 - acc: 0.7058 | val_loss: 0.50524 - val_acc: 0.7500
```

We hope that the “loss” fields (representing the total error) decrease over time.
We hope that the “acc” fields (representing the accuracy) increase over time.

The fields marked “loss” and “acc” refer to the error and accuracy thus far on the training data.
The fields preceded by “val_” refer to the aggregate performance thus far on the testing (validation) data.

At the conclusion of training, the trained network is stored in  
```
./ZtrainedNet/final-model.tfl
```

Step 5 – prepare the TFLearn Python testing program

We can now check the performance of the trained network on each of the training cases.
The program (“predict.py”) reads the validation (testing) data and tests each one using the trained network:

```
import deepneuralnet as net
import numpy as np
from tflearn.data_utils import image_preloader

model = net.model
path_to_model = './ZtrainedNet/final-model.tfl'
model.load(path_to_model)

X, Y = image_preloader(target_path='./validate', image_shape=(100,100), mode='folder', grayscale=False, categorical_labels=True, normalize=True)
X = np.reshape(X, (-1, 100, 100, 3))

for i in range(0, len(X)):
    iimage = X[i]
    icateg = Y[i]
    result = model.predict([iimage])[0]
    prediction = result.tolist().index(max(result))
    reality = icateg.tolist().index(max(icateg))
    if prediction == reality:
        print("image %d CORRECT  " % i, end=''
    else:
        print("image %d WRONG    " % i, end=''
print(result)
```

Step 6 – run the testing program

We run the training program by typing:  
```
python predict.py
```

On our testing data images, we get the result shown at the right:
The trained network failed to correctly classify two of the fishes.
Note that it appears confident in 5 of the 8 test cases.
✅ (additional) – using TensorBoard

Installing TensorFlow also installs TensorBoard, a graphing tool for visualizing various statistics during (or after) neural network learning. TensorBoard can also be used when using TFLearn.

To use TensorBoard, we have to modify the last line in our “deepneuralnet.py” file as follows. Change the line that says:

```python
model = tflearn.DNN(network)
```
to:

```python
model = tflearn.DNN(network, tensorboard_verbose=3, tensorboard_dir="logs")
```

This causes TensorFlow to create and maintain a log file during training, in the directory called “logs”. TensorBoard will use the log information to generate graphs that can be accessed in a browser.

To run TensorBoard, you’ll need to open a second command window, navigate to the same folder from which you are running the training program, and type:

```bash
tensorboard --logdir=logs
```

This tells tensorboard to use the logs found in the directory called “logs” specified in the model, above. TensorBoard responds with a message specifying an html address, something like:

TensorBoard 0.1.8 at http://MYCOMP:6006 (Press CTRL+C to quit)

TensorBoard is now generating graphs that can be accessed in a browser, using the indicated address: