CS 5600/6600/7890: Intelligent Systems
Assignment 5

Vladimir Kulyukin
Department of Computer Science
Utah State University

September 29, 2018

Learning Objectives

1. Decision Trees
2. Random Forests
3. Entropy
4. Training and testing ConvNets with TFLearn

Introduction

In this assignment, we'll compare the performance of decision trees, random forests, and ConvNets trained with tflearn on MNIST.

Exercise 1 (1/2 pt)

Draw a decision tree for $A \vee (B \wedge C)$.

Exercise 2 (1/2 pt)

Suppose that we have a set of samples S such that $|S| = 15$. Of the 15 samples, 10 are classified as C_1 and 5 are classified as C_2. Consider a feature F such that $Values(F) = \{v_1, v_2\}$. Let $|S_{v_1}| = 9$ so that 5 samples in S_{v_1} are classified as C_1 and 4 as C_2. Let $|S_{v_2}| = 6$ so that 3 samples in S_{v_2} are classified as C_1 and 3 as C_2. What is $Gain(S, F)$?

Problem 1 (2 pts)

The file `mnist_digits_decision_tree.py` has the function `test_dtr` that takes a numpy validation data array and prints a classification report with precision and recall percentages, as we discussed in Lecture 10. To run it, you'll have to install `sklearn` from `scikit-learn.org`. To safe you time with data conversion from the mnist format to the sklearn format, I defined a bunch of functions to convert the mnist data to sklearn format. Use `test_dtr` to train 10 decision trees and test each of them on `mnist_valid_data_dc`. Compute and record the average recall of your decision trees and the physical time it took you to train your trees. Here is a sample run of fitting a decision tree against the mnist validation data.

```
>>> test_dtr(mnist_valid_data_dc)
Training completed...
             precision    recall  f1-score   support

          0       0.92      0.94      0.93       980
          1       0.94      0.95      0.95      1135
          2       0.88      0.84      0.86      1032
          3       0.82      0.86      0.84      1010
          4       0.88      0.86      0.87       982
          5       0.81      0.82      0.81       892
          6       0.89      0.88      0.89       958
          7       0.89      0.90      0.89      1028
          8       0.81      0.78      0.79       974
          9       0.84      0.85      0.84      1009

avg / total       0.87      0.87      0.87     10000
```

The file `mnist_digits_random_forests.py` has the function `test_rf(num_trees)`. This function takes a number of trees in a random forest, fits the random forest to the training and testing data and targets (i.e., `mnist_train_data_dc` and `mnist_train_target_dc`), tests the trained forest on the mnist validation data in `mnist_valid_data_dc`, and prints a classification report. Again, to safe you time with data conversion from the mnist format to the sklearn format, I defined a bunch of functions to convert the mnist data to sklearn format. Use this function to train 10 random forest by varying the number of trees from 10 to 100 in increments of 10. Compute and record the average recall of your random forests and the physical time it took you to train your random forests. Here is a sample run.

```
>>> test_rf(10)
Training completed...
             precision    recall  f1-score   support

          0       0.95      0.99      0.97       980
          1       0.98      0.99      0.99      1135
          2       0.94      0.94      0.94      1032
          3       0.91      0.94      0.92      1010
          4       0.95      0.95      0.95       982
          5       0.93      0.91      0.92       892
          6       0.97      0.96      0.96       958
          7       0.96      0.95      0.95      1028
          8       0.94      0.92      0.93       974
          9       0.94      0.91      0.93      1009

avg / total       0.95      0.95      0.95     10000
```

Now let's proceed to a slightly more challenging part. Go to tflearn.org and install tflearn. Use tflearn to train and persist 5 convnets. Each of your convnets should have 2 or 3 convolutional layers. You can use the code in `mnist_convnet.py` and `mnist_convnet_load.py` discussed in Lecture 9 (both files are attached to the Lecture 9 announcement on Canvas) to see how you can build, train, and persist a tflearn network. To safe you some typing, I created the following Py files with some starter code:

1. `mnist_convnet_1.py`;

2. mnist_convnet_2.py;
3. mnist_convnet_3.py;
4. mnist_convnet_4.py;
5. mnist_convnet_5.py.

Each of the above files has the function build_tflearn_convnet_i(), where $1 \leq i \leq 5$. This is the only function that you should modify in each file. When you build your network correctly and run this file, you should see the following output.

```
Training Step: 5000  | total loss: 0.14771 | time: 63.192s
| SGD | epoch: 001 | loss: 0.14771 - acc: 0.9585 | val_loss: 0.12067 - val_acc:
0.9638 -- iter: 50000/50000
--
Training Step: 10000  | total loss: 0.06692 | time: 63.713s
| SGD | epoch: 002 | loss: 0.06692 - acc: 0.9917 | val_loss: 0.06600 - val_acc:
0.9791 -- iter: 50000/50000
```

Record the physical time it took you to train all five networks. You don't have to be exact. An approximate number will do. After your training is completed, TFLearn persists the trained model in a directory of your choice by creating three files: data, index, and meta. For example, if you persist your first network in /home/vladimir/nets/hw05_my_net_01.tfl, you will the following files created in it after the net is persisted:

1. hw05_my_net_01.tfl.data-00000-of-00001;
2. hw05_my_net_01.tfl.index;
3. hw05_my_net_01.tfl.meta.

Now go to test_mnist_convnets.py and define the five loading functions that will load each of your persisted convnets. Modify the variables net_path_1, net_path_2, net_path_3, net_path_4, net_path_5 accordingly. Here is how you can load and test a persisted ConvNet. The function test_convnet_model is implemented for you. Here is how you can load a trained network and test it against the mnist validation data (validX) and target (validY).

```
>>> net_model = load_mnist_convnet_1(net_path_1)
>>> test_convnet_model(net_model, validX, validY)
0.999
```

Of course, your accuracy may be different. I encourage you to experiment with different activation functions in your layers. For example, here is how you can feed an 28x28 input layer to a convolutional layer of 20 features with a 5x5 local receptive field and ReLU neurons.

```
input_layer = input_data(shape=[None, 28, 28, 1])
conv_layer_1  = conv_2d(input_layer,
                        nb_filter=20,
                        filter_size=5,
                        activation='relu',
                        name='conv_layer_1')
```

Here is how you can activate with tanhs and sigmoids.

```
conv_layer_1  = conv_2d(input_layer,
                        nb_filter=20,
                        filter_size=5,
                        activation='tanh',
                        name='conv_layer_1')

conv_layer_1  = conv_2d(input_layer,
                        nb_filter=20,
                        filter_size=5,
                        activation='sigmoid',
                        name='conv_layer_1')
```

Use the function `test_convnet_model` to compute the average accuracy of your ConvNet on the validation data, i.e., on `validX` and `validY`. Write a 1-page report where you compare the average performance of your decision trees, random forests, and convnets and the physical times it took you to train these classifiers. You can put all your observations in one table. So, are ConvNets worth it? Briefly argue why or why not based on your experimental discoveries.

What to Submit

1. `mnist_convnet_1.py`;
2. `mnist_convnet_2.py`;
3. `mnist_convnet_3.py`;
4. `mnist_convnet_4.py`;
5. `mnist_convnet_5.py`;
6. `test_mnist_convnets.py`;
7. `ai_f18_hw04_report.pdf`;
8. the tflearn persisted files for 5 trained convnets.

Happy Hacking!