CS 5600/6600: F20: Intelligent Systems
Assignment 5

Vladimir Kulyukin
Department of Computer Science
Utah State University

October 4, 2020

Learning Objectives

1. Building, Training, Testing, Validating TFLearn Nets
2. Dropout

Introduction

This assignment will give you an opportunity to build, train, test, and validate ConvNets and ANNs with tflearn on MNIST. My objective is to make sure that you have a working tflearn version installed on your machine to work on Project 1. You may want to review lectures 8 and 9 before working on this assignment. If you go to tflearn.org, you'll find out that tflearn requires an installation of tensorflow. I recommend that you work with tensorflow 1.13 or 1.14. I run 1.13.1 on my laptop and 1.14 on my GPU (both run Ubuntu 18.04 LTS). They are more stable than the latest 2.X versions.

Paper Analysis (2 points)

Read and write a one-page analysis on "Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images" by Nguyen, Yosinski, and Clune. I've included the pdf in the zip.

This paper was published in 2015 and caught immediate attention, because its findings poured some ice cold water on the DL hype. These researchers showed that it is possible to automatically generate images that human observers cannot recognize at all yet deep neural networks classify, with a very high degree of certainty, as familiar objects on which they were trained. Any serious student of DL should study this article very carefully and take heed of its findings.

Save your writeup in `cs5600_6600_F20_hw05_paper.pdf` and submit it in Canvas.

Problem 2 (3 pts)

For this problem you'll write and save your code in `cs5600_6600_f20_hw05.py` and test it with `cs5600_6600_f20_hw05_uts.py` where I wrote seven unit tests (UTs) for you. As usual, my recommendation is to go through the UTs one by one as you work on the functions below.

1) Write the function `make_tfl_convnet_slide22()` that builds and returns a ConvNet model defined on slide 22 in lecture 9 (DL and ConvNets: Part 2). The model takes MNIST 28x28x1 images as input, puts them through a convolutional layer with 20 filters (feature maps) with 5x5 local receptive fields, and activates its neurons with the sigmoid function. The convolutional layer is followed by a max pooler with a stride of 2. The pooler is fed into a fully connected layer of 100 sigmoid neurons. The fully connected layer is coupled to another fully connected layer of 10 sigmoid neurons (the output layer) with the softmax activation function. The network uses SGD as its weight optimizer, categorical crossentropy as its loss (i.e., cost) and the learning rate η of 0.1. The function returns a tflearn model object built with `tflearn.DNN()`. Here's a quick test.

```
>>> model_slide22 = make_tfl_convnet_slide22()
>>> model_slide22
<tflearn.models.dnn.DNN object at 0x7f9c16bf70f0>
>>> assert model_slide22 is not None
```

2) Write the function

```
fit_tfl_model(model, trainX, trainY,
                     testX, testY,
                     model_name, net_path,
                     n_epoch=1, mbs=10)
```

that takes a tflearn model object `model` and uses the tflearn methods `model.fit()` and `model.save()` to train and test (i.e., fit) the model on the datasets `trainX`, `trainY`, `testX`, `testY` and save the fit model under `model_name` in the folder `net_path`. The last two keyword parameters specify the number of training epochs and the mini batch size.

Use `test_ut01()` in `cs5600_6600_f20_hw05_uts.py` to test your `make_make_tfl_convnet_slide22()` and `fit_tfl_model()`. This UT creates a model, fits it on the MNIST training and testing sets for 5 epochs with a mini batch size of 10 and saves it under the name of `ConvNet_Slide22.tfl` in the directory specified in `NET_PATH`. Change `NET_PATH` accordingly. Here's my output from running `test_ut01()` on my Linux laptop (no GPU!).

```
Training samples: 50000
Validation samples: 10000
--
Training Step: 5000  | total loss: 0.24374 | time: 47.314s
| SGD | epoch: 001 | loss: 0.24374 - acc: 0.9535 |
                     val_loss: 0.20645 - val_acc: 0.9342 -- iter: 50000/50000
Training Step: 10000  | total loss: 0.25055 | time: 46.079s
| SGD | epoch: 002 | loss: 0.25055 - acc: 0.9239 |
                     val_loss: 0.12400 - val_acc: 0.9632 -- iter: 50000/50000
Training Step: 15000  | total loss: 0.16997 | time: 46.014s
| SGD | epoch: 003 | loss: 0.16997 - acc: 0.9470 |
                     val_loss: 0.09636 - val_acc: 0.9720 -- iter: 50000/50000
Training Step: 20000  | total loss: 0.04772 | time: 45.967s
| SGD | epoch: 004 | loss: 0.04772 - acc: 0.9844 |
                     val_loss: 0.06675 - val_acc: 0.9784 -- iter: 50000/50000
Training Step: 25000  | total loss: 0.07263 | time: 46.124s
| SGD | epoch: 005 | loss: 0.07263 - acc: 0.9698 |
                     val_loss: 0.05960 - val_acc: 0.9808 -- iter: 50000/50000
```

The net's validation accuracy rose from 0.9342 in epoch 001 to 0.9808 in epoch 005 while the validation loss fell from 0.20645 in epoch 001 to 0.05960 in epoch 005. Not bad at all and that's what we should be looking for in general: rising validation accuracy (the higher the better) and falling validation loss (the lower the better).

3) Write the function `load_tfl_convnet_slide22(model_path)` that takes a path to a saved tflearn model, uses the tflearn method `model.load()` to load the model, and returns the loaded model object. Slide 23 in lecture 9 gives you an example of how a tflearn model can be loaded.

Remember that a tflearn loader must know the architecture of the saved network. Essentially, it first builds the model's topology and then uses `model.load()` to load the persisted model into it. Use `test_ut02()` to load your saved model and use it to predict the target of a randomly chosen MNIST image. Here's my output (I'm skipping some deprication warnings that tflearn may print out).

```
raw prediction   = [[2.8279987e-07 2.3854611e-06 6.4301241e-07 7.0408198e-05 2.7615696e-03
                    2.7981298e-05 4.5003605e-07 9.1810634e-06 8.3719038e-05 9.9704343e-01]]
raw ground truth = [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
ground truth = 9
prediction   = 9
True
```

4) Write the function `test_tfl_model(model, X, Y)` that tests the accuracy of the model `model` on the set of examples `X` and the corresponding set of targets (i.e., ground truth) `Y`. The accuracy is computed as the percentage of the targets accurately predicted by the model. Use `test_ut03()` to test your function. This UT loads the trained slide 22 model and tests it on the MNIST validation dataset and its targets. Here's my output.

```
ConvNet Slide 22 Acc = 0.9836
```

5) Let's dive deeper. Write the function `make_deeper_tfl_convnet()` to build a tflearn ConvNet model that takes 28x28x1 input images. The input is fed into a convolutional layer of 20 filters with a local receptive field of 5x5 and sigmoid activation. The convolutional layer is connected to a max pooler with a stride of 2. The pooler is connected to the second convolutional layer of 40 filters with a local receptive field of 5x5 and ReLU activation. Here's how you can define it.

```
conv_layer2 = conv_2d(pool_layer, nb_filter=40,
                      filter_size=5,
                      activation='relu',
                      name='conv_layer_2')
```

The second convolutional layer is coupled to the second max pooler with a stride of 2. The second max pooler is connected to the first fully connected layer of 100 ReLU neurons and a dropout of 0.5. Here's how you can define the first fully connected layer.

```
fc_layer_1 = fully_connected(pool_layer2, 100,
                             activation='relu',
                             name='fc_layer_2')
fc_layer_1 = dropout(fc_layer_1, 0.5)
```

The first fully connected layer is connected to the second fully connected layer of 200 sigmoid neurons and a dropout of 0.5. The second fully connected layer feeds into the output layer, also fully connected, with 10 softmax neurons. The network uses categorical crossentropy as its loss, SGD as its weight optimizer, and the learning rate of 0.1.

Use `test_ut04()` in `cs5600_6600_f20_hw05_uts.py` to test your implementation of `make_deeper_tfl_convnet`. This UT creates the deeper model, fits it on the MNIST training and testing sets for 5 epochs with a mini batch size of 10 and saves it under the name of `Deeper_ConvNet.tfl` in the directory specified in `NET_PATH`. Here's my output from running `test_ut04()` on my Linux laptop (no GPU!).

```
Training samples: 50000
Validation samples: 10000
--
Training Step: 5000  | total loss: 2.30659 | time: 82.482s
| SGD | epoch: 001 | loss: 2.30659 - acc: 0.0838 |
                     val_loss: 2.30774 - val_acc: 0.0974 -- iter: 50000/50000
Training Step: 10000  | total loss: 2.31123 | time: 82.227s
| SGD | epoch: 002 | loss: 2.31123 - acc: 0.0851 |
                     val_loss: 2.30242 - val_acc: 0.1135 -- iter: 50000/50000
Training Step: 15000  | total loss: 0.19282 | time: 82.884s
| SGD | epoch: 003 | loss: 0.19282 - acc: 0.9434 |
                     val_loss: 0.13023 - val_acc: 0.9602 -- iter: 50000/50000
Training Step: 20000  | total loss: 0.07037 | time: 82.798s
| SGD | epoch: 004 | loss: 0.07037 - acc: 0.9768 |
                     val_loss: 0.09596 - val_acc: 0.9700 -- iter: 50000/50000
Training Step: 25000  | total loss: 0.10975 | time: 82.845s
| SGD | epoch: 005 | loss: 0.10975 - acc: 0.9752 |
                     val_loss: 0.07211 - val_acc: 0.9794 -- iter: 50000/50000
```

The validation accuracy rises from 0.0974 in epoch 001 (yes, just a touch below 10%) to 0.9794 in epoch 005. The validation loss falls from 2.30774 in epoch 001 to 0.07211 in epoch 005. That's pretty good. However, note that the first model's validation accuracy was 0.9808 in epoch 005 and its validation loss – 0.05960. So, if we confine ourselves to 5 training/testing epochs, we don't seem to gain much by diving deeper.

Before moving on, let's see what the validation accuracy of the deeper model is by running `test_ut05()`. Here's my output.

```
Deeper ConvNet Acc = 0.9842
```

Fine, but the first ConvNet model's accuracy was 0.9836 (see above). So, bottom line – we haven't gained much.

6) Choke me in the shallow water before I get too deep. So, let's get really shallow. Write the function `make_shallow_tfl_ann()` that defines a network whose input layer accepts 28x28x1 images. The input layer is connected to a fully connected layer of 20 sigmoid neurons. This hidden layer is connected to a fully connected output layer of 10 softmax neurons. The network uses SGD as its weight optimizer, categorical crossentropy as its loss function, and the learning rate of 0.1. Use `test_ut06()` to build, fit, and save this model. Here's my output.

```
Training samples: 50000
Validation samples: 10000
--
Training Step: 5000  | total loss: 0.26368 | time: 10.197s
| SGD | epoch: 001 | loss: 0.26368 - acc: 0.9322 |
                     val_loss: 0.27009 - val_acc: 0.9218 -- iter: 50000/50000
Training Step: 10000  | total loss: 0.22014 | time: 10.199s
| SGD | epoch: 002 | loss: 0.22014 - acc: 0.9282 |
                     val_loss: 0.22303 - val_acc: 0.9360 -- iter: 50000/50000
Training Step: 15000  | total loss: 0.20456 | time: 10.251s
| SGD | epoch: 003 | loss: 0.20456 - acc: 0.9563 |
                     val_loss: 0.20062 - val_acc: 0.9407 -- iter: 50000/50000
Training Step: 20000  | total loss: 0.13526 | time: 10.156s
| SGD | epoch: 004 | loss: 0.13526 - acc: 0.9664 |
                     val_loss: 0.18675 - val_acc: 0.9444 -- iter: 50000/50000
Training Step: 25000  | total loss: 0.25814 | time: 10.191s
| SGD | epoch: 005 | loss: 0.25814 - acc: 0.9422 |
                     val_loss: 0.20071 - val_acc: 0.9385 -- iter: 50000/50000
```

Let's take stock of what's happened. The validation loss falls from 0.27 in epoch 001 to 0.2. Nothing to boast about. The validation accuracy rises from 0.9218 in epoch 001 to 0.9385 in epoch 005. Not impressive, to say the least. Let's run `test_ut07()` to see what the this net's accuracy is on the MNIST validation data. Here's my output.

```
Shallow ConvNet Acc = 0.9538
```

The shallow net's accuracy is lower than either deeper net's. So, if we confine ourselves to 5 epochs, staying shallow has little payoff. It pays to dive deeper but not too deep.

What to Submit

1. Your one page paper analysis saved as `cs5600_6600_F20_hw05_paper.pdf`
2. `cs5600_6600_f20_hw05.py` with your code: the function stubs are in there.
3. The `nets` folder with your 3 saved tfl nets: `ConvNet_Slide22.tfl`, `Deeper_ConvNet.tfl`, and `Shallow_ANN.tfl`.
4. Zip components 1, 2, 3 into `hw05.zip` ans submit it in Canvas.

Happy Thinking and Hacking!