CS 5600/6600: F21: Intelligent Systems
Assignment 5

Vladimir Kulyukin
Department of Computer Science
Utah State University

October 2, 2021

Learning Objectives

1. Building, Training, Testing, Validating TFLearn Nets
2. Reflections on Deep Learning (DL)

Introduction

This assignment is slightly skewed toward reading and writing, because we're coming to the end of the data-driven intelligence part of our course and I want you to reflect on what we've learned about NNs and DL. The two papers I'm asking you to read and write about are also a response to some of your questions and comments in your writeups for Assignments 1 and 3.

The programming component of this assignment (Problem 3) will give you an opportunity to build, train, test, and validate CNs and ANNs with TFLearn on MNIST. My objective is simple – to make sure that you have a working TFLearn version installed on your computer for Project 1. To solve Problem 3, I used TFLearn 0.3.2 (an older but more stable version) and Python 3.6.8. To install TFLearn, I followed, more or less closely, the instructions at tflearn.org/installation/. There're numerous online resources if you find the latter instructions hard to follow. The installation of TFLearn requires the installation of tensorflow. TFLearn, like Keras, is just a layer on top of tensorflow.

Don't be shy to share your installation experiences/recommendations/shortcuts with the rest of the class. Help each other. This is not a rat race.

Paper Analyses (3 points)

The reading and writing part of this assignment includes 2 recent publications: 1) "Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images" by Nguyen, Yosinski, and Clune and 2) "The Computational Limits of Deep Learning" by Neil C. Thompson, Kristjan Greenewald, Keeheon Lee, Gabriel F. Manso. The PDFs are in the zip.

Read both papers and write a one-page analysis for each paper. Save your analyses in `cs5600_6600_F21_hw05_papers.pdf`.

"Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images" 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, but deep NNs 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 and possible implications.

Several months ago a student who took a class from me sent me a hey-dr-k-did-you-see-this email with a link to "The Computational Limits of Deep Learning" by Neil C. Thompson, Kristjan Greenewald, Keeheon Lee, Gabriel F. Manso. That paper made my semester! This is what the university environment is all about –

open knowledge sharing and open, respectful discussion. I learn as much, and often more, from my students, as, I hope, they learn from me.

I swallowed this 2020 paper in half an hour and then re-read it more carefully to enjoy it even more. I wish I had sufficient time to re-read it a couple dozen times. I found every paragraph to be packed without thought and careful deliberation. It also resonated deeply with me because of my own thoughts and reflections over the past decade. In several of my publications, I made cautious statements, not so much *against* DL as *about* it, based on my observations from my electronic beehive monitoring research: the cost of data storage and curation and of training DL models for embedded systems becomes prohibititely expensive to justify better accuracy. The promise of learning transfer is just that – a promise. Sure, I can train a better DL model by running it on my GPU for a month on several TB of data stored on a few external hard drives to obtain a ≈5 percent improvement over a random forest that trains in a day. But, is it worth i? More importantly, sometimes I cannot even deploy a trained DL model on an embedded computer (e.g., a raspberry pi/arduino) due to the latter's RAM limitations. In a couple of my recent research talks, I made a conceptual argument – DL approaches don't even attempt to address (insomuch as they play the radical version of Turing's Immitation Game – more on this later in the course) the fundamental questions posed by cognitive science: 1) how is our memory organized?; 2) how do we retrieve knowledge relevant to what we're doing and apply it?; 3) how do we represent and share knowledge we have with others through natural language (NL); 4) how and when do we modify our knowledge?; 5) how and why do we forget?. I can go on, but I hope you get the gist of what I'm trying to say.

"The Computational Limits of Deep Learning" takes a different approach. It makes no appeal to cognitive science or brain science. Its arguments, based on mathematics, statistics, and economics, are focused on the ever growing computational requirements of DL and some logical implications of those requirements. If one, rather cautiously, extrapolates forward the reliance of bulldozing on ever growing GPU farms, one logically concludes that progress along these lines may well be economically, technically, and environmentally unsustainable.

I know, I know — hopium is a powerful drug and we're neurolinguistically conditioned to believe in perpetual linear scientific progress (although History whispers us many different, cautionary tales – e.g. "The Collapse of Complex Societies" by J. Tainter). Be it as it may, we cannot take these researchers arguments and findings lightly or brush them away as pessimistic nonsence, because we're dealing with a comprehensive study: the authors have analyzed 1,058 research papers on various DL methods and applications in five different domains (image classification, object detection, question answering, named entity recognition, and machine translation) touted by DL proponents as the best testimony to the validity, correctness, and promise of bulldozing.

For your analysis of the second paper, read only the first 18 pages. You don't have to read, unless you want to or have sufficient time, the supplementary materials. This paper offers not only a great review of data-driven intelligence research but also a superb list of references, many of which are available online.

You don't have to agree with my views. I accept and welcome alternative views (so long as they're well articulated and argued). I always look forward to your analyses and enjoy reading and commenting on them.

Problem 2 (2 pts)

For this problem you'll write and save your code in `cs5600_6600_f21_hw05.py` and test it with `CS5600_6600_f21_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.

I wrote two functions for you to model in `cs5600_6600_f21_hw05.py`. The first function is `make_tfl_mnist_convnet()`.

```
def make_tfl_mnist_convnet():
    input_layer = input_data(shape=[None, 28, 28, 1])
    conv_layer = conv_2d(input_layer, nb_filter=20,
                         filter_size=5,
                         activation='sigmoid',
                         name='conv_layer_1')
    pool_layer = max_pool_2d(conv_layer, 2, name='pool_layer_1')
```

```
    fc_layer_1 = fully_connected(pool_layer, 100,
                                 activation='sigmoid',
                                 name='fc_layer_1')
    fc_layer_2 = fully_connected(fc_layer_1, 10,
                                 activation='softmax',
                                 name='fc_layer_2')
    network = regression(fc_layer_2, optimizer='sgd',
                         loss='categorical_crossentropy',
                         learning_rate=0.1)
    model = tflearn.DNN(network)
    return model
```

This function builds and returns a TFLearn CN model. 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 (stochastic gradient descent) 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()`.

The function `fit_tfl_model()` in `cs5600_6600_f21_hw05.py` takes a TFLearn model `model` object 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 fitted 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_tfl_mnist_convnet()` and `fit_tfl_model()`. This unit test (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 `my_tfl_mnist_convnet` in the directory specified in `NET_PATH`. Change `NET_PATH` as necessary. Here's my output from running `test_ut01()` on my Linux laptop.

```
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
```

This 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! 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).

I also wrote the function `load_tfl_mnist_convnet()` 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.

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 TFLearn warnings).

```
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
```

I wrote 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 model and tests it on the MNIST validation dataset and its targets. Here's my output.

```
tfl mnist model acc = 0.9842
```

Let's dive deeper. Write the function `make_deeper_tfl_convnet()` to build a TFLearn CN 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 the convolutional layer.

```
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 and its dropout. Note that the second max pooler (i.e., `pool_layer2`) feeds into it.

```
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, defined above, 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_f21_hw05_uts.py` to test your implementation of `make_deeper_tfl_mnist_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 `my_deeper_tfl_mnist_model` in the directory specified in `NET_PATH`. Here's my output from running `test_ut04()` on my laptop (warnings are skipped).

```
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 a whopping 0.9794 in epoch 005. The validation loss falls from 2.30774 in epoch 001 to 0.07211 in epoch 005. This is 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 epochs, we don't seem to have gained 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.

```
tfl mnist deeper convnet acc = 0.9886
.
----------------------------------------------------------------------
Ran 1 test in 12.625s

OK
```

The first CN's accuracy was 0.9842. This CN's accuracy is 0.9886). Bottom line – we haven't gained much by way of accuracy.

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

```
shallow tfl mnist ann acc = 0.9596
.
----------------------------------------------------------------------
Ran 1 test in 1.425s
OK
```

The shallow net's accuracy is lower than either deeper net's. So, if we confine ourselves to 5 epochs, staying shallow didn't pay off. It may be worth it to dive deeper or change the topology of the shallow net, activation functions, learning rate, etc.

What to Submit

1. Your two paper analyses (2 pages) saved as `cs5600_6600_F21_hw05_papers.pdf`
2. `cs5600_6600_f21_hw05.py` with your code: the function stubs are all in there.
3. The `nets` folder with your 2 saved trained TFLearn nets: `my_deeper_tfl_mnist_model` and `my_shallow_tfl_mnist_ann`. I leave it up to you (and your hardwdare) to choose the number of epochs to train each net with. Try to do 5 or more.
4. Zip components 1, 2, 3 into `hw05.zip` and submit it in Canvas.

Happy Reading, Writing, and Hacking!