AI Programming (IT-3105) Module 1: Deep Learning with Tensorflow

Due Date: The 1st demo session (see course web page for further details)

1 Purpose

1. Learn to solve classification tasks using feed-forward neural networks with backpropagation.
2. Gain hands-on experience with Google's Tensorflow, a system for configuring, training and testing neural networks.
3. Learn to build a general interface for Tensorflow such that a wide variety of network architectures can be quickly and easily tested on a diverse array of datasets.
4. Learn to generate some standard visualizations of neural networks.

2 Assignment Overview

You will use Tensorflow to classify data from a wide variety of sources, including the classic MNIST benchmark of digit images. This requires good familiarity with the Tensorflow API along with clever insights into the proper topologies and parameter settings for artificial neural networks (ANNs) used as classifiers. Your grade for this project module will be based on several factors:

- the ability to design an interface to Tensorflow that facilitates easy configuration and running of neural nets of widely varying topologies (a.k.a. architectures) on a diversity of datasets, and visualizing the results.
- the ability to design appropriate ANN architectures for particular classification tasks,
- the successful performance of these ANNs on those tasks, and
- the competence to explain the behavior of an ANN on a dataset by visualizing layer activation patterns, weight matrices and other useful relationships (such as dendrograms, described in accompanying documents).

Your system must be object-oriented and must provide the same type of modularity and building-block composition as the classes Gann (General Artificial Neural Network) and Gannmodule in file tutor3.py. You are free to build the system from scratch or to use tutor3.py as a starting point. Otherwise, you are **not**

permitted to use other people's code (e.g. from github) for this project, unless explicitly specified in this or related documents. You are also **not** permitted to use Keras or other high-level abstractions of Tensorflow, since these will not give the desired hands-on experience with Tensorflow. In fact, you will be building something akin to Keras, from scratch.

Your system must accept any of a wide range of user-specified scenarios, including a variety of network architectures, and it must do so **without requiring a recompilation of the system**. Systems that require rebuidling for each scenario will lose a substantial number of points during grading.

3 Introduction

Artificial Neural Networks (ANNs) are one of a standard set of tools for solving classification problems. Recently, they have begun to show exceptional performance on speech and image classification tasks. Human-level competence on these particular tasks has eluded artificial intelligence for decades, but in a few short years, the algorithms have gone from neophyte to genius, often out-performing human experts.

Systems such as Theano, Cafe and (most recently) Tensorflow have helped bring Deep Learning to the masses of students and professionals. As recently as 2010, it was difficult to build effective multi-layered ANNs for solving complex classification tasks, since deep-learning research papers rarely provided all of the essential implementation details. Today, Tensorflow and its army of users (many of whom post their results on the web) makes the job a whole lot easier. Though neural networks have performed very well on many tasks for several decades, the recent quantum leap has greatly invigorated the field. It stems from many factors, including a deeper understanding of the backpropagation process, which have enabled researchers to build truly deep nets, often consisting of 10 (or even 100 or more) hidden layers. However, neither exceptionally deep networks nor the (somewhat complex) techniques that have sparked recent breakthroughs are the centerpiece of this project. Rather, we will focus on one tool, TensorFlow, which serves as the computational backbone for many of these advances, but in a context that does not require very deep networks. A few hidden layers with relatively conventional dynamics (i.e., activation functions) will suffice.

By learning and using TensorFlow, you will open many possibilities for further research and development using deep neural networks, whether on a master's or PhD project, or out in industry. As detailed in the lecture notes for this module, the big wins with Tensorflow are both its speed in handling matrix operations, a cornerstone of Deep ANNs, and its automatic calculation of partial derivatives, often involving variables that are quite distant from one another in a sequence of mathematical expressions, e.g., the weight of an interior ANN connection and the network?s final output. Together, these Tensorflow features allow users to piece together fast ANNs using only a few lines of code.

The datasets for this assignment should not require extremely deep networks, but anywhere from one to three hidden layers may work optimally, depending upon your choice of the other parameters. Typical activation functions with which to experiment include: sigmoid, hyperbolic tangent (tanh), and rectified linear units (RELUs), all of which are primitives in Tensorflow. Typical error functions are the mean sum of squared errors (MSE) and cross-entropy, also provided as Tensorflow primitives.

A primary goal of this assignment is to streamline this trial-and-error process so that many different networks can be experimented with in a short period of time. To this end, you will build an interface to Tensorflow that facilitates these experiments.

4 Essential Project Requirements

This section provides a long list of mandatory requirements for your system. Read it carefully. For an in-depth explanation of several critical components of this system, see the document *deepnet_iface_details.pdf* provided on the course web pages. You are responsible for the contents of both documents.

4.1 The Complete List of Scenario Parameters

What follows is the minimum set of scenario-defining input parameters that your system must accept and then use to build and run the ANN. Although these parameters can certainly be entered via an elaborate graphical user interface (GUI), that is not a requirement for the assignment. It is fine to enter them as arguments to a **single**, high-level function or to enter them via a **simple**, scenario-defining script, i.e., a script whose sole purpose is to supply parameters to your system. It is **not** acceptable if these parameters are spread throughout your source code and must be modified by bouncing around within and between files.

If entered via a GUI, any combination of menus, submenus, windows, sub-windows, etc. is fine, but if entered by more primitive means (i.e, function arguments or a script), then all data should be entered as the arguments to a single function or as a single script. In general, the specification of a scenario should be very straightforward and thus facilitate a wide variety of user-specified runs during a short 10-15 minute demonstration period.

The scenario-defining parameters are:

1. Network Dimensions - the number of layers in the network along with the size of each layer.
2. Hidden Activation Function - that function to be used for all hidden layers (i.e., all layers except the input and output).
3. Output Activation Function - this is often different from the hidden-layer activation function; for example, it is common to use *softmax* for classification problems, but only in the final layer.
4. Cost Function - (a.k.a. loss function) defines the quantity to be minimized, such as mean-squared error or cross-entropy.
5. Learning Rate - the same rate can be applied to each set of weights and biases throughout the network and throughout the training phase. More complex schemes, for example those that reduce the learning rate throughout training, are possible but not required.
6. Initial Weight Range - two real numbers, an upper and lower bound, to be used when randomly initializing all weights (including those from bias nodes) in the network. Optionally, this parameter may take a value such as the word *scaled*, indicating that the weight range will be calculated dynamically, based on the total number of neurons in the upstream layer of each weight matrix.
7. Optimizer - the choices must include basic gradient descent, RMSprop, Adagrad and ADAM.
8. Data Source - specfied as either:
    - a data file along with any functions needed to read that particular file.
    - a function name (such as one of those in *tflowtools.py*) along with the parameters to that function, such as the number and length of data vectors to be generated, the density range (i.e. upper and lower bound on the fraction of 1's in the feature vectors), etc.

9. Case Fraction - Some data sources (such as MNIST) are very large, so it makes sense to only use a fraction of the total set for the combination of training, validation and testing. This should default to 1.0, but much lower values can come in handy for huge data files.
10. Validation Fraction - the fraction of data cases to be used for validation testing.
11. Validation Interval (vint) - number of training minibatches between each validation test.
12. Test Fraction - the fraction of the data cases to be used for standard testing (i.e. after training has finished).
13. Minibatch Size - the number of training cases in a minibatch
14. Map Batch Size - the number of training cases to be used for a *map test* (described below). A value of zero indicates that no map test will be performed.
15. Steps - the total number of minibatches to be run through the system during training.
16. Map Layers - the layers to be visualized during the map test.
17. Map Dendrograms - list of the layers whose activation patterns (during the map test) will be used to produce dendrograms, one per specified layer. See below for more details on dendrograms.
18. Display Weights - list of the weight arrays to be visualized at the end of the run.
19. Display Biases - list of the bias vectors to be visualized at the end of the run.

4.2 Basic Scenario Results

The mandatory information generated by each run of your system is:

1. A plot of the progression of the training-set error from start to end of training. Each data point is the average (per case) error for a single mini-batch.
2. A plot of the progression of the validation-set error from start to end of training.
3. A listing of the error percentage for the training and test sets, as evaluated after training has finished.

The plots of training-set and validation-set error must be on the same graph. This allows you to easily detect whether the network has been overtrained, and where overtraining begins.

4.3 Visualization

In addition, the scenario specification may call for additional visualizations; your system must provide the following basic functionalities:

1. Mapping - This involves taking a small sample of data cases (e.g. 10-20 examples) and running them through the network, with learning turned off. The activation levels of a user-chosen set of layers are then displayed for each case.

2. Dendrograms - For any given network layer, a comparison of the different activation vectors (across all cases of a mapping) can then serve as the basis for a dendrogram, a convenient graphic indicator of the network's general ability to partition data into relevant groups.

3. Weight and Bias Viewing - These are simple graphic views of the weights and/or biases associated with the connections between any user-chosen pairs of layers. Although you may want to display weights and biases intermittently during a run, this assignment only requires their visualization at the end of a run.

Users of your system must be able to turn these options on and off, as well as specify more details about them, such as **which hidden layer** to use as the basis of a dendrogram, or **which weights and biases** to display at the end of a run, etc.

These additional (but still mandatory) features are described in more detail in the document *deepnet_iface_details.pdf*. The combination of visualized weights, biases and mappings (along with dendrograms) supports thorough investigations into the behavior of a neural network, thus allowing the user to break open the *black box* and reduce some of the mystery associated with this powerful, but not particularly transparent, AI tool.

4.4 Data Sets

At the demonstration session, you will be required to use some or all of the data sets described below. Some are generated dynamically by calls to functions in *tflowtools.py*, while others are provided as simple text files. Those generated by functions are large lists consisting of pairs: feature vector, target vector or label. Data files typically house one case per row, with items separated by commas, and the feature vector consisting of all but the last element of a row, while that last element constiitutes the class label. It is safe to assume this format for most data files but to then write a special reader function for any file that violates that assumption.

Accompanying each dataset below is a quantitative definition of *reasonably good*. It indicates the percentage of correct answers that your network should achieve on the **training set** when training has completed. Some datasets are harder than others, so the numbers vary. For each case, it is mandatory that your training set is a randomly-selected group of cases that constitute 80% or more of the total number of cases in the data set. For example, you might use 10% for validation, 10% for testing and 80% for training.

Some of the data sets listed below are not part of the performance test, but your system still needs to be able to run them and produce sensible results: they should not crash your system or give unintelligible results such as a negative error rate.

It is important to note that during training, you will probably want to use one of the standard error/loss functions such as mean-square error or cross-entropy. These give useful gradients for modifying the network's weights and gradually reducing error. However, when training has finished and you want to compute the number of training cases that the network correctly classifies, you should switch to a simpler error function: one that merely tests whether or not the output node with the highest activation level corresponds to the target value. Tensorflow's primitive function **in_top_k(predictions,targets,k)** (with k = 1) is very helpful in this regard. You can use it for checking the network's accuracy on the training, validation and test sets, but you do not want to use it during actual training, since it may not provide useful gradients for learning.

Machine-Learing (ML) researchers strongly emphasize performance on both the training and test sets, since the latter indicates the network's ability to generalize from (and not merely memorize) the training data. Conversely, the focus of this project is on the implementation of a general-purpose neural-net simulator, and

much less on the actual performance of any given network on a particular data set. Hence, the performance requirements of individual runs are far less stringent than one encounters in ML conferences, journals and competitions.

4.4.1 Parity

The function **gen_all_parity_cases** in *tflowtools.py* generates all bit vectors of a specified length and then packs them into cases consisting of a vector along with a target of either 1 or 0, indicating the parity of the vector: 1 (0) means that the vector contains an odd (even) number of 1's.

Alternatively, if the *double* flag is True (the default), targets will be either [1,0] indicating even or [0,1] denoting odd.

REASONABLY GOOD: 95% (using the set of all 10-bit vectors as the dataset and using a random 80% of them for training)

4.4.2 Symmetry

The function **gen_symvect_cases** in *tflowtools.py* produces bit vectors of a specified length that are either symmetric (or not) about their middle – for odd-length vectors, the value of the middle bit makes no difference. Thus, the following are symmetric length-8 vectors: 00111100, 11011011, 00100100, and these are symmetric length-9 vectors: 000010000, 101101101, 110010011. Cases consist of a vector plus one extra bit whose value denotes symmetric (1) or not (0), and the **gen_symvect_cases** generates an equal (or nearly so) number of symmetric and non-symmetric cases.

REASONABLY GOOD: 99% (using 2000 vectors of length 101 as the dataset)

4.4.3 Autoencoder

Cases in these data sets consist of binary feature vectors whose targets are identical to the feature vectors. In *tflowtools.py*, two functions generate autoencoder data sets:

- **gen_all_one_hot_cases** produces all one-hot feature vectors of a specified length, e.g. all 8 one-hot vectors of length 8.
- **gen_dense_autoencoder_cases** produces vectors of a specified length with a specified density range. For example, a density range of (0.4, 0.7) entails that all vectors will consist of anywhere from 40% to 70% 1's.

Both of these functions produce cases consisting of the feature vector and its copy (the target).

REASONABLY GOOD: You will not be asked to run a performance test on an autoencoder at the demo session, but you may choose an autoencoder as the network that you explain in detail.

4.4.4 Bit Counter

This data consists of bit vectors whose class is simply the count of the number of 1's in the vector. To produce these data sets, simply call the function **gen_vector_count_cases** in *tflowtools.py*.

REASONABLY GOOD: 97.5% (on a set of 500 randomly-generated cases of length 15)

4.4.5 Segment Counter

These data cases consist of a bit vector as the features and a number (represented as a one-hot vector) denoting the number of 1's segments in the vector. For example the feature vector 111110001010000111 is classified as a "4", since there are 4 groups of 1's (separated by 0's) in the vector. To generate these segment-vector cases, use the function **gen_segmented_vector_cases** in *tflowtools.py*. This function takes the following arguments:

- **size**, length (in bits) of each input/feature vector,
- **count**, the number of cases to produce,
- **minsegs**, the minimum number of segments in a vector,
- **maxsegs**, the maximum number of segments in a vector,
- **poptargs**, a flag indicating whether or not one-hot vectors (also known as population-coded vectors) are being used as targets. The default is True.

For example, the call **gen_segmented_vector_cases(25,10,0,5)** will produce 10 cases, each employing a vector consisting of 25 bits and housing anywhere from zero to five segments. The returned cases are in the normal format: a pair of vectors, the features and the one-hot target. Note that a vector with zero segments will have a one-hot coding of [1,0,....] while a vector with one segment has [0,1,0,....]. Do not worry about the fact that this function may occasionally produce duplicate cases.

REASONABLY GOOD: 95% (on a set of 1000 randomly-generated cases of length 25 with 0 - 8 segments in each case)

4.4.6 MNIST

This is a classic machine-learning benchmark consisting of all 10 digit classes (0 - 9) represented by 28 x 28 pixel arrays. The original data files contain tens of thousands of cases in a rather complex format, but by importing the file mnist_basics.py, you gain access to several simple functions for generating standard data sets (consisting of one-dimensional feature vectors and integer class labels), such as **load_all_flat_cases**. For more important details on installing and using the MNIST files and accessors, read tflow-mnist.pdf.

For this dataset, it is fine to use only a random subset (S) containing about 10%, of the complete dataset. Then divide S into training, validation and test sets.

For MNIST, you will probably want to scale all pixel values to numbers in the range [0,1] before feeding them into the input layer of your network.

REASONABLY GOOD: 95% (on the training fraction of subset S)

4.4.7 Popular Datasets from UC Irvine

All of the following datasets are available from the UC Irvine Machine-Learning Repository, which provides supporting information about each set. The raw datasets (without explanations) can also be found on the "Projects" web page for this course.

Some of these datasets have features whose ranges vary dramatically. For example, feature 3 may have values in the range (0,1) whereas those for feature 5 might be in range (0, 100). If these values are fed directly into the input layer of a neural network, feature 5 could easily dominate other features, making classification extremely difficult.

One solution to this problem is to scale all features by their average and standard deviation. Hence, feature j for case c ($f_{c,j}$) is scaled to $I_{c,j}$ (the value of input neuron j for case c) as follows:

$$I_{c,j} \leftarrow \frac{f_{c,j} - \mu_j}{\sigma_j} \tag{1}$$

where μ_j and σ_j are, respectively, the mean and standard deviation of feature j as calculated across all the entire case set.

Alternatively, to insure an input value in the closed range [0,1], you can scale by $f_{j,min}$ and $f_{j,max}$, the minimum and maximum values of feature j across the case set. Then:

$$I_{c,j} \leftarrow \frac{f_{c,j} - f_{j,min}}{f_{j,max} - f_{j,min}} \tag{2}$$

These or other similar functions are worth considering when working with the Irvine data sets.

Wine Quality This dataset consists of 1599 cases, each of which has 11 real (or integer) features and one of SIX integer classes. If downloading this set from UC Irvine, note that it contains two files, one for red and one for white wine. The file provided on the course webpage is the one for red wine (which involves 6 classes). The white-wine file involves 7 classes. You are only responsible for the handling the red-wine file. Note: In both files, the elements of each case are separted by semicolons, not commas.

REASONABLY GOOD: 95%

Glass This dataset contains 214 cases, each of which has 9 real-valued features and one of SIX integer classes. Note: For some odd reason, the class labels are 1-7 but with **no examples of class 4**. So a one-hot bit vector of length 6 can be used as a target.

REASONABLY GOOD: 95%

Yeast There are 1482 cases in this set. Each case has 8 real-valued features and one of 10 integer labels.

REASONABLY GOOD: 90%

Hacker's Choice Choose any additional data set (consisting of at least 100 cases) from the UC Irvine Machine-Learning Repository. **You must have at least one such data set available to your system.** As with all of the datasets listed above, it should be possible to run scenarios involving a hacker's-choice dataset at the demo session.

REASONABLY GOOD: This dataset will NOT be part of the performance test, since success rates can vary dramatically, depending upon the UC Irvine dataset that you choose. Some are much harder than others. Your system simply needs to be able to run this data set and produce sensible training, validation and test results.

It must be possible to run any of these data sets through your ANN system without the need to recompile your code between datasets.

5 The Demonstration Session

At the demo for this project, you will be asked to run your system on any of the datasets described above, and using a diversity of network architectures and parameter settings. This process must occur without the need to re-compile or otherwise re-make your system. Your system must easily process the scenario specifications for each new run and display the appropriate graphics (such as weight arrays, dendrograms, etc.). **Failure to satisfy these basic conditions will incur significant point loss.**

In addition, the performance of your system on different datasets will be checked. Perfection is not expected but reasonably good results are. During a performance test, you will be given a dataset and then be expected to choose an ANN architecture and parameter settings that yield a good result. It is important to prepare ahead of time for this evaluation by knowing the datasets and what network architectures and parameter settings work best for each one.

Finally, you will be asked to explain the behavior of an ANN on a particular dataset. You can choose any of the data sets described above, with the exception of the bit-counter. Your explanation must include the use of **all** of the following graphic tools: mappings, weight-and-bias visualizations and dendrograms. You should generate all of these diagrams prior to the demo and then use them as part of your verbal explanation during the demo. Your explanation need not be a complete mathematical proof, but it should shed considerable light on the activity under the hood that enables your network to successfully perform the given task.

The point breakdown for the demonstration is as follows:

1. The working system that you can fully explain and that is capable of handling any of the datasets and a wide selection of architectures and parameter settings. (20 points).
2. Completion of all performance tests with reasonably good results, where *reasonably good* is quantified above (15 points).
3. The detailed explanation of a successful network (5 points).

Demonstrations will be given **individually**, even if the code was written as a group. Each individual is therefore responsible for understanding and explaining every aspect of the code and results. Failure to clearly explain any aspect of the code can result in a significant point loss, so each group member should understand ALL of the code.

There is no written report for this project module.

6 Other Practical Information

A zip file containing the commented code (for the entire project) must be uploaded to BLACKBOARD prior to the demo session in which this project module is evaluated. You will not get explicit credit for the code, but it is crucial that we have the code online in the event that you decide to register a formal complaint about your grade (for the entire course). Even if you work in a group, **both** group members must upload the same code to their individual BLACKBOARD accounts. The 40 total points for this module are 40 of the 100 points that are available for the entire semester.