COMS 4705 – Natural Language Processing – Fall 2015
Assignment 3
Recursive Neural Networks with Dependency Parsing

(version 1.2, March 23, 2016)

Due: April 8, 2016 (11:59:59 PM)

Introduction

In this assignment, we will:

1. Go through the basics of Theano, a numerical computation library for python;
2. Develop and evaluate a logistic regression model;
3. Develop and evaluate a recursive neural network .

This document includes the following sections:

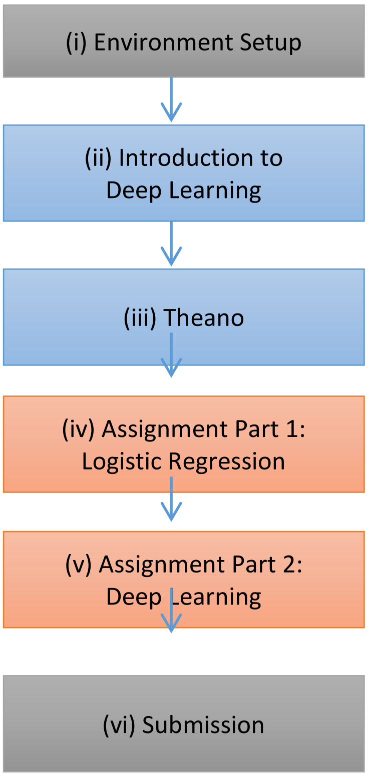
Section A: Background

- a) Introduction to deep learning and Theano
- b) Implementing logistic regression using Theano

Section B: Implementation of Deep Learning for Question Answering

- c) Implementing a dependency parse recursive neural network
- d) Experiment with hyperparameter settings and different features and evaluate results

Structure of this document:



(i) Environment Setup

Setup everything to get you ready for this assignment

(ii) Introduction to Deep Learning

A Quick introduction to Deep Learning, a popular framework for machine learning

(iii) Theano

Introduction to and tutorial on Theano.

(iv) Assignment Part 1: Logistic Regression

Implement logistic regression using Theano.

(v) Assignment Part 2: Deep Learning

Implement a recursive neural network using dependency parses. Improve the performance by tuning parameters.

(vi) Submission

Follow the instructions to submit. Be sure to submit everything needed and set the right permissions.

(i) Environment setup

We assume that you have already got your CLIC account and set up the hidden directory during the first assignment. For this assignment, you can acquire the provided code by running the following commands:

```
cd ~/hidden/$HIDDENNUMBER/
mkdir Homework3
cd Homework3
cp -r ~coms4705/Homework3/* .
```

Before running code for this assignment, make sure to run this command (or add it to your .bashrc):

```
source envsetup.sh
```

to set your PYTHONPATH correctly.

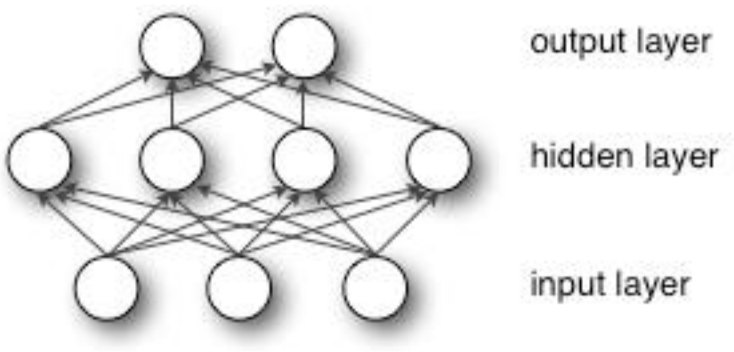
BACKGROUND INFORMATION

(ii) Introduction to Deep Learning

Deep learning is currently a very popular topic in NLP. Deep learning approaches have very recently obtained state-of-the-art performance across many different NLP tasks. The appeal of deep learning is enhanced by the idea that tasks can be modeled without extensive feature engineering[1]. The term "deep learning" often refers to neural networks with many hidden layers. However, many people use the term for any neural network with non-linear transformations.

Neural networks have been used for tasks such as language modeling (Bengio et al, 2003, Mikolov et al, 2010), word representation (Mikolov et al, 2013, Pennington et al, 2014), parsing (Socher et al, 2013), sentiment analysis (Socher et al, 2013), and question answering (Iyyer et al, 2014). For this course, you will implement the dependency recursive neural network for factoid question answering developed by Iyyer.

The standard multi-layer neural network (also called a multi-layer perceptron or MLP) looks something like this[2]:



In this example, the input layer x is of size $d_0 = 3$, the hidden layer h is of size $d_1 = 4$, and the output layer $f(x)$ is of size $d_2 = 2$. In matrix notation:

$$f(x) = g_2(b_2 + W_2 \cdot g_1(b_1 + W_1 \cdot x))$$

[1] This is true for many tasks. However, the most successful methods for some tasks include a deep learning component as well as manually derived features.

[2] http://deeplearning.net/tutorial/mlp.html (We will cover Theano later in this homework)

where $W_n \in R^{d_n \times d_{n-1}}$ and $b_n \in R^{d_n}$ are the **parameters** of the network for layer n and g_n is an **activation function** for layer n. We learn the parameters by minimizing a **cost function** using some **optimization method**. The **hyperparameters** of the model are the dimensionality of the input and hidden layers and number of hidden nodes[3].

Determining the network structure, activation function, cost function, and optimization method is part of the process of modeling a neural network. If the cost function is differentiable, we can take the gradient of the cost function with respect to the parameters using **backpropagation** and apply a gradient-based optimization technique. For this course, we will not be deriving gradients but rather using a library for automatic differentiation[4]. Alternatively, there are many search-based algorithms for learning neural network parameters without differentiation.

There are many gradient-based optimization algorithms (popular methods include Adagrad, Adam, and LBFGS)[5]. Adagrad is the method used for this project and is an improved form of gradient descent. If you have taken machine learning, you are probably familiar with gradient descent (if you haven't, it doesn't matter for this course). Gradient descent is a basic optimization algorithm for finding the local minimum of a function.

In order to actually model a network, we still need to select an activation function and a cost function. An activation function is a non-linear function that allows the MLP to approximate any function. Common activation functions include the sigmoid, hyperbolic tangent, and rectified linear unit (ReLU or rectifier). The cost function is very task-specific. We may wish to minimize squared error or another difference metric. We may also want a probabilistic interpretation. In this framework we want to minimize the negative log likelihood of the training data. One common probabilistic cost function is the **softmax function.**

Two common network structures in NLP are **recurrent** and **recursive** neural networks. Recurrent neural networks are used for sequential input or time series. In the NLP case, these networks have been used successfully for part-of-speech tagging and named entity recognition. Recursive neural networks are used for tree structures and have had success at parsing and sentiment analysis. The difference between these networks and the MLP is that they model "hidden states" at the current time or node that are a composition of all previous states. For recurrent and recursive neural networks, the gradient-based techniques are backpropagation through time (BPTT) and backpropagation through structure (BPTS).

Here is one example of a recurrent neural network:

$$h_t = g_h(x_t \cdot W_x + h_{t-1} \cdot W_h + b_h)$$
$$f(x_t) = g_o(h_t \cdot W_o + b_o)$$

Each state h_t is a composition of the previous state and x_t, the feature vector for element *x* at time *t*. The parameters to be learned are the weight matrices W, the biases b, and an initial state h_0. For a specific application, consider that the elements x_t are words and the outputs $f(x_t)$ are part-of-speech tags. We may choose the sigmoid function for g_h and the softmax function for g_o which will give us a probability distribution for a word at time *t*.

In many NLP applications, the input vector will be a **word embedding**, a continuous representation of a word. These word embeddings may be pre-trained using methods such as GloVe or Word2vec and then fed into the model. During training, there are several variations on how we treat word

[3] For a full exploration of deep learning, check out Richard Socher's class. This class will only cover some of the higher level points of modeling and training neural networks. There are many different architectures, activation functions, cost functions, and optimization methods that will help you to tailor a network to your task if you are familiar with them.

[4] For a *deep* understanding of deep learning, you should derive and implement backpropagation rather than using an autograd library.

[5] Optimization is a subject with a vast literature of its own.

embeddings. The embeddings may be fixed and not changed during training or we may initialize them with pre-trained embeddings and learn them as parameters (possibly also adding a constraint that they are not allowed to vary too much). Alternatively, we may not use pre-trained embeddings at all and learn them as parameters with random initialization. Generally, the ad hoc wisdom is that for small-ish datasets it is better to use fixed embeddings.

For this project, you will be implementing a recursive neural network using dependency parses for factoid question answering (QANTA): https://cs.umd.edu/~miyyer/qblearn/ (see the paper for more details). The goal of the task is to answer questions from the Quiz Bowl contest, which means there are a limited set of possible answers.

From the last assignment, you should be familiar with dependency parsing. The data has already been parsed using the Stanford parser with the basic, typed dependencies where each node has only one parent so that they form a tree. The data has also been preprocessed so that each node in the tree is either a word or a multi-word entity (such as "Battle of Midway") and each answer is a multi-word entity as well.

The recursive neural network is defined as follows, for a particular node n:

$$h_n = f(W_v \cdot x_n + b + \sum_{k \in K(n)} W_{R(n,k)} \cdot h_k)$$

where x_n is the word representation of node n, K(n) are the children of n, and R(n,k) is the dependency relationship between node n and child k. The parameters to be learned are the matrix W_v, *r* matrices W_R (one for each possible dependency relation), and the bias b. (In their work, they used word2vec to create word embeddings trained on the preprocessed question text. For this assignment, you will experiment with different options).

The intuition behind the model is that the questions should be close to their correct answers in vector space and far away from incorrect answers. (For their experiments, they also incorporated an additional weighting known as WARP). The **contrastive max-margin** objective function is as follows:

$$C(\theta) = \frac{1}{N} \sum_{t \in T} \sum_{s \in s_t} \sum_{z \in Z_t} \max(0, 1 - x_c \cdot h_s + x_z \cdot h_s)$$

where N is the total number of all nodes in all trees in the training data, T is the set of trees in training, s_t is the set of all nodes in the tree t, and Z_t is a set of randomly sampled incorrect answers for each t. h_s is the hidden vector for node s, x_c is the vector for the correct answer, and x_z is the vector for an incorrect answer. We want this number to be as small as possible: if the dot product with the correct answer $x_c \cdot h_s$ is larger than the dot product with an incorrect answer $x_z \cdot h_s + 1$, then there is no penalty.

The activation function f is the normalized hyperbolic tangent:

$$f(x) = \frac{\tanh(x)}{\|\tanh(x)\|}$$

Once the question and answer embeddings have been learned, the embeddings are then fed into a logistic regression classifier and trained to predict the answer as a multi-class learning problem.

IMPLEMENTATION

Overview

You will be implementing QANTA using Theano, a numerical computation library in python.

We recommend starting this assignment early as the algorithm may take a few hours to converge and the CLIC machines may get crowded. Alternatively, you can work on your own machine and install the necessary packages with `virtualenv` and `pip`.
(http://docs.python-guide.org/en/latest/dev/virtualenvs/)

The software depends on the following 4 libraries (which are installed on CLIC but which you would need to install if you want to work somewhere else):
numpy
gensim
theano
TSNE[6] (in version 0.17 of scikit-learn or downloadable from here: https://lvdmaaten.github.io/tsne/)

If you really want to understand theano, you should work through the following tutorial:
http://deeplearning.net/software/theano/tutorial/
For this assignment, we will cover what you need to know.

Here is an outline of the assignment:

1. Implement multi-class logistic regression using theano – 1 point
2. Implement the function to convert dependency trees to ordered lists of indices (in other words a topologically sorted list) – 5 points
3. Implement the activation function – 2 points
4. Implement the recurrence relation and cost function for QANTA using theano – 10 points
5. Experiment with different options for embeddings – 2 points
   a. Embeddings trained on question data using word2vec
   b. Randomly initialized embeddings
   c. Embeddings trained on question data using word2vec and fixed
6. TSNE visualization of answers – 5 points

Provided data
The raw data is in the file `hist_split.json`. The data is split into 4 parts: train, dev, devtest, and test. In the file `corpus.py` there are several functions you may find useful. `get_train, get_dev, get_devtest, and get_test`, respectively

Each datum is as follows:

```
[
 [word_string, relation_string, parent_index],
 …
],
answer_string
```

The word_string and answer_string belong to the same vocabulary. The idea is to learn words and answers jointly. Each word and answer may be a multi-word phrase such as "henry_clay" or "battle_of_midway". The relation_string refers to the dependency parse relation and the parent_index is the index into the list of its parent node.

You may also want to pretty print the file to see some examples:

```
python -m json.tool hist_split.json | less
```

ASSIGNMENT INSTRUCTIONS

For this assignment, we have provided the following files:

```
envsetup.sh
hist_split.json
adagrad.py
corpus.py
```

[6] Pronounced tee snee, apparently

```
dependencyData.py
dependencyRNN.py
evaluation.py
featureExtractor.py
logisticRegression.py
main.py
test.py
```

1) Logistic regression (1 point)
As a warmup, you will implement logistic regression using Theano.

In the file `logisticRegression.py`, you will implement the __init__ method for the LogisticRegression class. This method takes in one parameter: a dimensionality. In this method, you must compile two theano functions: `train` and `predict`. (Make sure to set them to `self.train` and `self.predict`).

You will find these tutorials very useful:
[http://deeplearning.net/software/theano/tutorial/examples.html - a-real-example-logistic-regression](http://deeplearning.net/software/theano/tutorial/examples.html) (for binomial logistic regression)
http://deeplearning.net/tutorial/logreg.html (this is pretty much exactly what you are implementing)

The main difference from the first tutorial is that you will be implementing multinomial logistic regression rather than the standard binomial logistic regression. For your convenience, you may use the function `theano.tensor.nnet.softmax`. You can test your results by running the script `test.py --lr`.

If you have implemented this correctly, your results will look like this:

```
cost at epoch 97: 0.0334393380655
cost at epoch 98: 0.0331787297654
cost at epoch 99: 0.0329229579185
classification report: precision    recall  f1-score   support

          0       0.99      0.97      0.98        88
          1       0.94      0.86      0.90        91
          2       1.00      0.93      0.96        86
          3       0.90      0.86      0.88        91
          4       0.93      0.89      0.91        92
          5       0.91      0.95      0.92        91
          6       0.91      0.99      0.95        91
          7       0.93      0.93      0.93        89
          8       0.92      0.90      0.91        88
          9       0.82      0.95      0.88        92

avg / total       0.92      0.92      0.92       899
```

There are some issues to be aware of when using Theano:
a. Theano is often more like writing math equations than programming. You create a bunch of symbolic variables and then compile them into a function.
b. Theano uses symbolic representation, so most matrices and vectors will not have a shape until runtime. When you compile a theano function, even if it compiles successfully, at runtime there may be an issue in something like matrix multiplication where the matrices are the wrong shape.

2) Input handling – 5 points

The next step will be implementing the function `sort_datum` in the file dependencyData.py.

This function will take one input: a data point in the format described in the data section above. The output is a list of indices.

The dependency RNN training requires that the array be topologically sorted left to right, with the root node as the leftmost node. Data should be further sorted numerically ascending.

For example, given this datapoint:

```
[[u'ROOT', None, None],
 [u'majority', u'nn', 2],
 [u'opinion', u'nsubj', 6],
 [u'in', u'prep', 2],
 [u'this', u'det', 5],
 [u'case', u'pobj', 3],
 [u'notes', u'root', 0],
 [u'that', u'mark', 17],
 [u'california', u'poss', 11],
 [u"'s", u'possessive', 8],
 [u'continual', u'amod', 11],
 [u'invocation', u'nsubjpass', 17],
 [u'of', u'prep', 11],
 [u'worthless', u'amod', 14],
 [u'remedies', u'pobj', 12],
 [u'is', u'auxpass', 17],
 [None, None, None],
 [u'buttressed', u'ccomp', 6],
 [u'by', u'prep', 17],
 [u'the', u'det', 20],
 [u'experience', u'pobj', 18],
 [u'of', u'prep', 20],
 [u'other', u'amod', 23],
 [u'states', u'pobj', 21]]
```

This function should return (organized by depth):

```
[0,
 6,
 2, 17,
 1, 3, 7, 11, 15, 18,
 5, 8, 10, 12, 20,
 4, 9, 14, 19, 21,
 13, 23,
  22]
```

3) Activation function – 2 points

Implement the normalized_tanh function in the file dependencyRNN.py

The functions you may find useful are theano.tensor.tanh and theano.tensor.sqrt.

4) Recurrence relation and cost function – 10 points

Implement the function recurrence in the file dependencyRNN.py.

This function calculates the cost and hidden state for a specific node. Recall that the hidden nodes are calculated as $h_n = f(W_v \cdot x_n + b + \sum_{k \in K(n)} W_{R(n,k)} \cdot h_k)$ and the cost of a specific node is $\sum_{z \in Z_t} \max(0, 1 - x_c \cdot h_s + x_z \cdot h_s)$

The functions you will need are theano.scan, theano.tensor.dot, theano.tensor.maximum, theano.tensor.set_subtensor, and theano.tensor.as_tensor_variable.

It is very important to understand the function theano.scan. This is the theano equivalent to a for loop. To use scan, you need to specify a function to apply at each iteration. Theano will supply to the function the next item(s) in the sequence (specified by sequences), the output(s) from the previous execution of the function (initialized by outputs_info), and any other variables it needs (specified by non_sequences). See here for more details: http://deeplearning.net/software/theano/library/scan.html

Run main.py --save random_init.npz when you are finished. In your README.txt report the first 10 lines for epochs 0, 10, and 25 and the train and validation accuracy.

5) Options for embeddings – 2 points

a) Write a script called buildWord2Vec.py that takes in the file hist_split.json and uses gensim to train a model as follows:

```
model = Word2Vec(size=100, window=5, min_count=1)
model.build_vocab(sentences)
alpha, min_alpha, passes = (0.025, 0.001, 20)
alpha_delta = (alpha - min_alpha) / passes

for epoch in range(passes):
    model.alpha, model.min_alpha = alpha, alpha
    model.train(sentences)

    print('completed pass %i at alpha %f' % (epoch + 1, alpha))
    alpha -= alpha_delta

    np.random.shuffle(sentences)
```

sentences is a list of lists of words, which you will extract from the training data. Gensim is an open-source tool for creating word embeddings using the word2vec method (Mikolov et al, 2013).

You should use the method model.save on the resulting word embeddings and name the file whatever you want.

Remember: you are only allowed to use the training set for creating embeddings and training. For ease of implementation we are assuming that we know how many total words and answers there are but not the embeddings for all of them.

b) In dependencyRNN.py, modify self.params to not include self.We (this will hold the word embeddings fixed during training)

For a and b, run `main.py -We <word2vec_file> --save word2vec_init.npz` and `main.py -We <word2vec_file> --save word2vec_init_fixed.npz`. In your README.txt report the first 10 lines for epochs 0, 10, and 25 and the accuracy on training and development.

6) TSNE visualization of answers – 5 points

TSNE is a dimension reduction technique useful for visualizing high dimensional data in low dimensions (usually 2). For this part of the assignment, you will create visualizations for the answers produced in rand_init.npz

Write a script to make a scatterplot of the answer embeddings in 2D space. You can load the answer embeddings by calling DependencyRNN.load(filename) and then the answer method will produce a dictionary of answers to their embeddings, which you will need to convert to a matrix. Then run TSNE on the answer matrix.

If using the tsne python library:
X_reduced = tsne(X, no_dims = 2, initial_dims = 100, perplexity = 30.0)

If using scikit-learn:
tsne = sklearn.manifold.TSNE(n_components=2, perplexity=30.0)
X_reduced = tsne.fit_transform(X)

Then plot the results using a plotting library such as matplotlib with each point represented on the graph as its word.

You have a few options for creating the plots:

1) Set up an X11 server (same as in the previous assignment)
2) Create the graphs from your local machine (the files are small so this should be fine)
3) Use ipython notebook server:
   http://jupyter-notebook.readthedocs.org/en/latest/public_server.html

After you have created this visualization, save it to a file called tsne_visualization.png

Deliverables

You should have a copy of each of these files in your directory when you finish your assignment, e.g. in

HW3_ROOT=/home/$USER/hidden/$HIDDENNUMBER/Homework3/

where $USER is your UNI and $HIDDENNUMBER is the number that Prof. Radev emailed you.

Please follow this folder structure exactly; otherwise you may lose points for otherwise correct submissions!

1. Logistic regression code
   a. $HW3_ROOT/logisticRegression.py
2. Input handling code
   a. $HW3_ROOT/dependencyData.py

3. RNN code
    a. `$HW3_ROOT/dependencyRNN.py`
4. Script to generate word embeddings
    a. `$HW3_ROOT/buildWord2Vec.py`
5. TSNE visualization of answers
    a. `$HW3_ROOT/tsne_visualization.png`
6. README file containing results and discussions
    a. `$HW3_ROOT/README.txt`

As a final step, run a script that will setup the permissions to your homework files, so we can access and run your code to grade it:

```
~coms4705/hw_set_permissions.sh 3 <YOUR_PIN>
```

Please ignore the error message: “cannot access ~/hidden/<PIN>/output” since you are not asked to write on ‘output’ directory for this assignment.