CPSC 477/577 Natural Language Processing

Homework 3: Recursive Neural Networks with Dependency Parsing

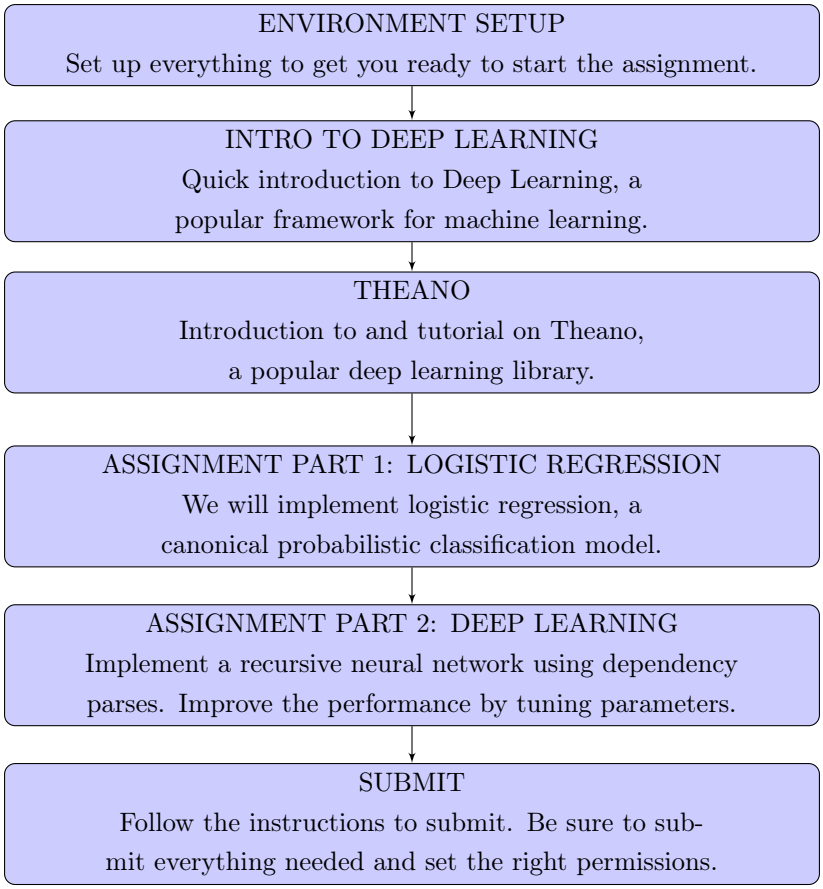
Instructor: Dragomir Radev *Due: April 10th, 2017*

I. 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 is structured in the following sequence:



II. Environment Setup

We assume that you have already got your 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:

```
$ cp -r /home/classes/cs477/assignments/Homework3 ~/hidden/<YOUR_PIN>/
```

In this assignment, we will be using Scikit-Learn, a machine learning library, and Theano, a deep learning library. These have both been installed on the Zoo cluster, but if you are working locally on your own computer, you have to install them yourself. If you have Anaconda installed (which is highly recommended – installation instructions are easily available online), simply enter the following in Terminal:

```
$ conda install scikit-learn=0.15.2
$ conda install Theano
```

III. Introduction to Deep Learning

Have you noticed that of late, everybody is talking about Artificial Intelligence and the existential threat it poses? This is largely because of advances in Deep Learning, which have lead to advancements in a host of fields ranging from computer vision to NLP. Deep Learning is (partially) why cars can now drive themselves.

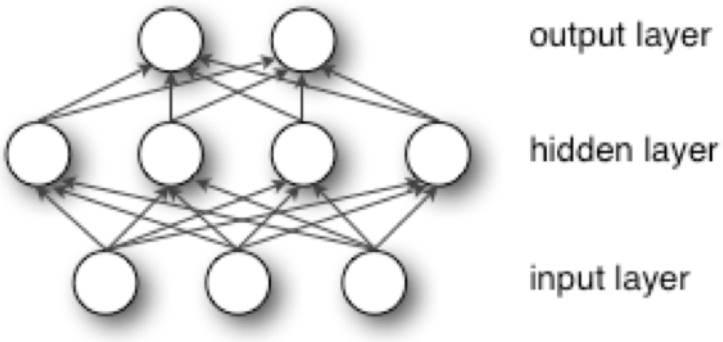
Deep learning allows us to model tasks without extensive feature engineering. This is important because there are tons of problems that are easy for humans to do but hard for us to describe with enough detail to specify which features we should train our machine learning algorithm on. For example, suppose we want to detect different objects in pictures – is there a person in this picture? a car? a dog? This problem is quite hard for us to select features for. How, after all, do we describe how a human looks or differentiate humans from other objects? An image is composed of pixels, but clearly the pixels by themselves aren't very helpful to tell what is a human or not. This suggests that to make the problem tractable for a machine learning algorithm, we need to represent our data in a useful way. This task is called Representation Learning. A form of representation learning that has had amazing success is Deep Learning. In Deep Learning, a neural network learns the optimal representation of the problem by itself – we don't have to specify the features. Instead, the neural network builds its own representation from other simpler representations. For example, given the input representation of a picture as pixels, a deep learning network would represent the pixels as edges, the edges as corners and contours, the corners and contours as object parts, and the object parts as object identity. Please see the introductory chapter (more specifically, page 6) in Deep Learning by Goodfellow et al (2016) for more (you can access the book at `http://www.deeplearningbook.org`).

More specific to NLP, Deep Learning has obtained state-of-the-art performance for language modeling, word representation, sentiment analysis, and question answering. This project will involve the last of these examples, question answering, specifically implementing the dependency recursive neural network for factoid question answering developed by Iyyer et al (2014).

Multilayer Perceptrons

We will now discuss the quintessential example of a deep learning model, called the multi-layer perceptron (MLP) or a feedforward neural network. In short, a MLP approximates a function f^* for which $y = f^*(\vec{x})$ by defining a mapping $y = f(\vec{x}; \vec{\theta})$, and learning the value of $\vec{\theta}$, the vector of weight and bias parameters. Our approximating function is a composition of other functions: for example, we could define $f(\vec{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\vec{x})))$, and $f(\vec{x}; \vec{\theta}) = f^{(3)}(f^{(2)}(f^{(1)}(\vec{x}, \vec{\theta_1}); \vec{\theta_2}); \vec{\theta_3})$ where $f^{(n)}(\vec{x}; W_n, b_n) = W_n \cdot \vec{x} + b_n$. Here, W_n is called the weight matrix, and b_n is called the bias. We would call $f^{(1)}$ the first layer, and $f^{(2)}$ the second layer, and so on and so forth. Note that these "layers" correspond precisely to the representations we discussed previously.

An example of a MLP is below:



Here, we have a three layer MLP, where the input (first) layer x is of size $d_1 = 3$, the hidden (second) layer h is of size $d_2 = 4$, and the output (third) layer $f(x)$ is of size $d_3 = 2$. From before, we see that $f(\vec{x}) = f^{(2)}(f^{(1)}(\vec{x}; \vec{\theta}); \vec{\theta})$. Expanding this out, we calculate the output layer as follows:

$$f(\vec{x}) = f^{(2)}(b_2 + W_2 \cdot f^{(1)}(b_1 + W_1 \cdot \vec{x}))$$

where $W_n \in \mathbb{R}^{d_n \times d_n - 1}$ and $b_n \in \mathbb{R}^{d_n}$ are the weight and bias parameters of the network for layer n and f_n is the *activation function* for layer n. We learn the parameters (from before, $\vec{\theta}$) by minimizing a cost function using some optimization method. The hyperparameters of the model are the dimensionality of the input and hidden layers, and the number of hidden layers. Finally, note that all the arrows in this computational graph (which describe how the functions are composed together) point in one direction. This is why a MLP is alternately known as a feedforward network. Finally, every MLP computational graph will be a directed acyclic graph (DAG), which you should recall from CPSC 202.

An MLP "learns" by taking the gradient of the cost function with respect to the parameters $\vec{\theta}$ using a process called backpropagation and then applying a gradient-based optimization technique. The backward propagation of errors, commonly known as backpropagation, involves propagating an input vector through each layer of the MLP. The output of the MLP is compared against the desired output, and the cost function is used to calculate an error value for each neuron in the output layer. We then backpropagate to previous layers, calculating for each neuron the associated error value which represents its contribution to the error in the output. These error values are used to calculate the gradient of the cost function with respect to the parameters. For a more in-depth look into backpropagation, consult the Deep Learning book.

After backpropagation, the MLP applies a gradient-based optimization technique, traditionally a variant of stochastic gradient descent (SGD). Classical gradient descent simply finds a minima by taking steps proportional to the negative of the gradient, which is calculated for all examples. SGD is an online version of classical gradient descent, which follows a similar procedure, but using the gradient of a single

or a few examples at a time. A detailed explanation of SGD falls beyond the scope of this discussion (though once again, the Deep Learning book has excellent coverage of the topic). The variant of SGD we use in this problem set is called Adagrad.

To specify a network, we still need to select an activation function and cost function. An activation function is a nonlinear differentiable function that allows the MLP to approximate any function. Common activation functions are the sigmoid $S(t) = \frac{1}{1+e^{-t}}$, the rectified linear unit (ReLU) $f(x) = \max(0, x)$[1] and the normalized hyperbolic tangent

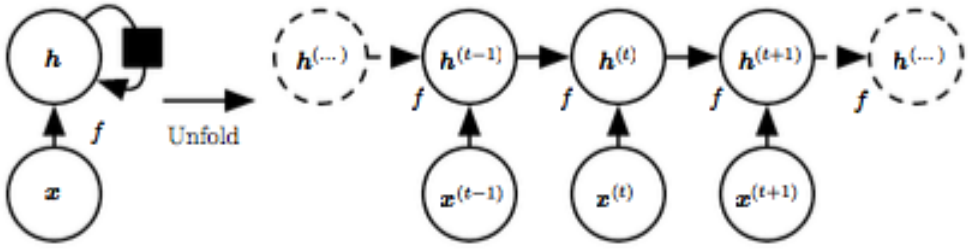
$$f(x) = \frac{\tanh(x)}{||\tanh(x)||} \text{ (where } ||x|| \text{ is the norm of } x)$$

We will be using the normalized hyperbolic tangent as our activation function in this problem set. A cost function is dependent on what problem we are trying to solve. For example, in some cases a good cost function is squared error or another difference metric. In the case of this homework, we want a probabilistic cost function: we wish to minimize the negative log likelihood of the training data. We use a common probabilistic cost function called softmax, or the normalized exponential function. It is defined as:

$$\sigma(\vec{z})_j = \frac{e^{z_j}}{\sum_{k=1} K e^{z_k}} \text{ for } j = 1, ..., K$$

Recurrent and Recursive Neural Networks

Recurrent Neural Networks (RNNs) are neural networks that can model temporally-dynamic systems, which is to say *sequences indexed by time*. This is useful in NLP for tasks ranging from handwriting and speech recognition to part-of-speech tagging (one of the TAs, Jungo, actually worked on the latter). Crucially, RNNs allow us to share the same parameter weights across several time steps, something that MLPs cannot do. Consider part-of-speech tagging, where the neural network is fed tagged sentences (for simplicity, of fixed length) and tries to learn the rules of POS tagging. An MLP would need to learn all of these POS rules separately at each position in the sentence, whereas an RNN, by virtue of sharing weights among several "time" steps (here, positions in the sentence), can just learn them once. The RNN can, in addition, learn dependencies between the various time steps. To make RNNs more clear, consider the following example (Figure 10.2 in Sec 10.1 of the Deep Learning book):



where x is the input state and h is the hidden state. This recurrent network does not have an output layer: all it does is process information from the input and then incorporate it into the state h which is passed forward through time. At every step, we calculate the state h at time t as a function of the previous state (which encodes information about the whole past sequence) and the new information (which comes from x). This should be clear from the unfolded version of the network presented to the right. We write this network out mathematically as follows:

$$\vec{h}^{(t)} = f(\vec{h}^{(t-1)}, \vec{x}^{(t)}; \vec{\theta})$$

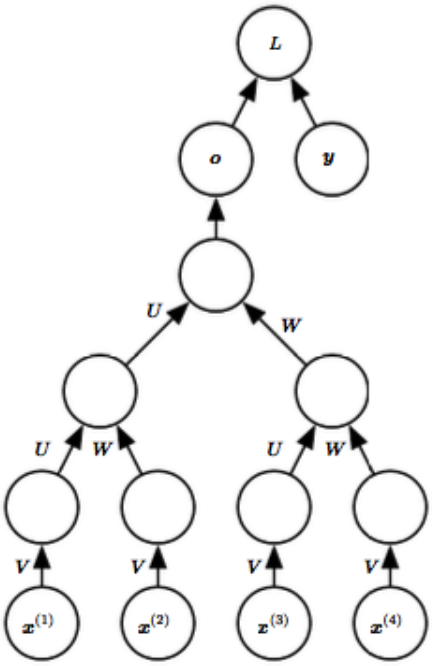
[1] This function is, strictly speaking, not differentiable, so in practice we use the smooth approximation $f(x) = \log(1 = e^x)$

where $\vec{h}, \vec{x}$ are time-indexed vectors representing the hidden and input state respectively, and $\vec{\theta}$ is, as before, the vector of parameters that we seek to learn. Note that f is a single model that can operate on all time steps, rather than needing a separate model for each possible time step, which as we previously mentioned is the distinguishing characteristic between a RNN and a MLP. Learning a single model allows us to make generalizations that did not appear in the training set, and allows us to make good models with far fewer training examples. Expanding out our equation as we did previously, we see that

$$h^{\vec{(t)}} = f(\vec{h}^{(t-1)} \cdot W_h + \vec{x}^{(t)} \cdot W_x + b_h)$$

An RNN learns in a manner similar to a MLP, through a procedure called back-propagation through time (BPTT). Essentially, we "unfold" the network through time (like in the right part of the above figure), and then do regular back-propagation. We optimize in the same way as before, using some variant of SGD. For this project, we use the normalized hyperbolic tangent as our activation function, and softmax as our cost function.

We end this brief discussion of RNNs now, and turn to Recursive Neural Networks, which can be thought of as a generalization of RNNs. A Recursive NN operates on any hierarchical (tree-like) structure, whereas a Recurrent NN operates on a specific structure – namely, a linear chain. Recursive NNs have had notable success in modeling relationships between concepts where the concepts are represented by embeddings (we will explain what this is later). In fact, this is what we will be using them for! Below is an example of a recursive network from the Deep Learning book (Figure 10.14, Section 10.6), which you should compare and contrast to the previously-shown recurrent network.



Here, x is a variable-sized input sequence that we map to a fixed-size representation o with parameters U, V, W. Do not worry about the specifics of this example, just note that we operate on a tree-like structure as opposed to a linear chain like in the case of an RNN. This is very helpful in, for example, parsing, where we can fix the tree structure of the Recursive NN to the parse tree of a sentence provided by a parser, and thus more easily (and reliably) understand the dependencies within the sentence than if we used a linear chain. In the case of this problem set, we use the *principle of semantic composition*, which states that the meaning of a phrase comes from the meaning of the subparts of the phrase and the syntax that glues the parts together ("the whole is the same as the sum of the parts"), to learn the

meaning of sentences through the use of a dependency-tree recursive neural network, which can represent any dependency parse of a sentence and thus understand its meaning through the principle of semantic composition.

QANTA

Now, we will talk about QANTA, the actual dependency-tree recursive NN (DTRNN) you will be implementing in this homework to answer Quiz Bowl questions. You can access the paper and related resources at `https://cs.umd.edu/~miyyer/qblearn/`. We *highly* recommend taking the time to read through and understand this paper.

To start, QANTA is a DTRNN that jointly learns answer and question representations in the same vector space. Jointly learning question and answer representations together allows us to model relationships between answers and discard the incorrect assumption that answers are independent. QANTA is short for *question answering neural network with trans-sentential averaging*, where the "trans-sentential averaging" part comes from the fact that we average the representations of each sentence in the question.

Before using QANTA, we need to convert words into a continuous representation called a *word embedding* which try to capture the linguistic context of words. We touched upon this in Homework 0. These word embeddings are pre-trained using a method like Word2Vec or GloVe, which take in a corpus of text and produce a vector space of hundreds of dimensions, where each word is represented by a vector and similar words (words that share many contexts) are closer together in the space. There are multiple approaches to how we treat word embeddings: we may fix the embeddings and not change them during training, or we may learn the pre-trained embeddings as parameters. Alternatively, we may not even used pretrained embeddings at all and instead randomly initialize them. Generally, it is a good idea for small-ish datasets to use fixed embeddings. We will experiment with these approaches in the problem set, but the approach QANTA uses is pretraining the word embeddings with Word2Vec and then learning them as parameters with the DTRNN. Given a sentence, QANTA uses these vector representations of each word in the sentence, and the dependency parse of the sentence, to build a vector representation of the sentence as a whole. The vector representations of each sentence in a question are averaged to find the vector representation of the entire question. We use a contrastive max-margin objective function to ensure that vectors of questions are near their correct sentences and far away from incorrect answers. We then feed the question and answer representations into a logistic regression classifier which predicts the answer as a multi-class learning problem.

Let us now formalize our above discussion in relation to building vector representations of each word into a representation of the entire sentence. Recall that a dependency parse of a sentence is a tree. Traversing recursively from the bottom of the dependency tree to the top, QANTA combines the current node's word vector with its children's hidden vector to form the current node's hidden vector. An example of this process is found on page 3 of Iyyer et al. To put this formally:

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

where $K(n)$ are the children of n, x_n is the word vector (provided by the embedding method), and $R(n,k)$ is the dependency relation between node n and child k in the parse tree. The parameters our NN attempts to learn are the matrix W_v, one matrix W_R for each possible dependency relation, and the bias b.

Our recursive NN uses the *contrastive max-margin* objective function to encourage the vectors of question sentences to be near their correct answers and far from incorrect answers. The contrastive max-margin

objective function is defined as:

$$C(\theta) = \sum_{s \in S} \sum_{z \in Z} L(rank(c, s, Z)) \cdot \max(0, 1 - x_c \cdot h_s + x_z \cdot h_s)$$

Noting that $rank(c, s, Z)$ is expensive to compute, we instead use an approximation (explained in the paper in depth) to find the form of the function we use, which is:

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

where N is the total number of 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. Additionally, h_s is the hidden vector for node s, x_c is the vector for the correct answers, and x_z is the vector for an incorrect answer.

As we mentioned before, we use the normalized hyperbolic tangent as our activation function, and after learning question and answer embeddings, we use a logistic regression classifier to predict answers as a multi-class learning problem – that is, every answer is a class, and we predict into which class each question is assigned to. Mathematically, we find

$$P(Y = i \mid x, W, b) = softmax_i(Wx + b) = \frac{e^{W_i x + b_i}}{\sum_j e^{W_j x + b_j}}$$

which finds the probability that a question x has the answer i. The answer we assign to the question is the one for which this probability is maximal, i.e.

$$\text{argmax}_i P(Y = i \mid x, W, b)$$

Summary of QANTA model

We initialize word embeddings using a method like Word2Vec or GloVe. We then use a dependency-tree recursive neural network to learn the embeddings, which is defined as followed:

- For a particular node n:

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

- The activation function f is:

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

- The objective function is:

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

After learning the embeddings, we use a multinomial logistic regression classifier to predict answers.

We are now ready to start the problem set!

IV. Implementation

Overview

You will implement 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 Zoo servers may get crowded. Alternatively, you can work on you own machine and instal the necessary packages with virtualenv and pip. See `http://docs.python-guide.org/en/latest/dev/virtualenvs/` for more detail.

Our implementation depends on the following 4 libraries, which are installed on the Zoo, but which need to be installed on your computer if you choose to work locally:

- NumPy
- Gensim
- Theano
- TSNE (which is in scikit-learn or can be downloaded from `https://lvdmaaten.github.io/tsne/`)

If you really want to learn/understand Theano, we recommend working through the following tutorial: `http://deeplearning.net/software/theano/tutorial/`.

The outline of this assignment is as follows:

1. Implement multi-class logistic regression using Theano — 1 point
2. Implement the function to convert dependency trees into a topologically sorted list — 5 points
3. Implement the activation function — 2 points
4. Implement the recurrence relation and objective function for QANTA using Theano — 10 points
5. Experiment with different options for embeddings — 2 points
    - Embeddings trained on question data using word2vec
    - Randomly initialized embeddings
    - 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 of the following form:

```
[
  [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 like "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
```

V. Assignment Instructions

For this assignment, we have provided the following files:

- hist_split.json
- adagrad.py
- corpus.py
- dependencyRNN.py
- evaluation.py
- logisticRegression.py
- main.py
- test.py

V.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.

Use the discussion we provided at the end of the Intro to Deep Learning section and the following two extremely useful tutorials to understand how to implement multinomial logistic regression:

- `http://deeplearning.net/software/theano/tutorial/examples.html-a-real-example-logistic-regressi` (for binomial logistic regression)
- `http://deeplearning.net/tutorial/logreg.html` (for multinomial logistic regression, which is exactly what we want to do)

For your convenience, you may use the function theano.tensor.nnet.softmax. You can test your results by running the script python test.py – lr.

If your implementation is correct, your results should 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 some issues to be aware of when using Theano:

- Theano is often more like writing math equations than programming. You create a bunch of symbolic variables and then compile them int a function.
- Theano uses symbolic representation, so most matrices and vectors will not have a shape until runtime. So when you compile a Theano function, even if it compiles successfully, at runtime there may be an issue with something like matrix multiplication where the matrices are the wrong shape.

V.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 recursive NN training requires that the array be *topologically sorted* left to right, with the root node as the leftmost node. The data should be further sorted numerically ascending.

For an 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]
```

V.3 Activation Function (2 points)

Now, implement the normalized_tanh function in the file dependencyRNN.py. Functions you will find useful are theano.tensor.tanh and theano.tensor.sqrt.

V.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:

$$\sum_{z \in Z} \max(0, 1 - x_c \cdot h_s + x_z \cdot h_s)$$

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 most important to understand the function theano.scan. This is the Theano equivalent of 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). For more details, please refer to `http://deeplearning.net/software/theano/library/scan.html`.

Run

```
python main.py -- save random_init.npz
```

when you are finished. In your README.txt report the first 10 lines for epochs 0, 10, and 29 as well as the train and validation accuracy.

V.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 = gensim.models.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.

You should use the method model.save on the resulting word embeddings and name the file whatever you want. We denote the filename as <word2vec_file> later.

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

Run

```
python main.py --We <word2vec_file> --save word2vec_init.npz
```

In your README.txt report the first 10 lines for epochs 0, 10, and 29, and the accuracy on training and development.

(b) In dependency RNN.py modify self.params to not include self.We. This will hold the word embeddings fixed during training. Run:

```
python main.py --We <word2vec_file> --save word2vec_init_fixed.
npz
```

In your README.txt report the first 10 lines for epochs 0, 10, and 29, as well as the accuracy on training and development.

IMPORTANT: for final submission, add self.We back to self.params

V.6 TSNE Visualization of Answers (5 points)

TSNE, or t-distributed stochastic neighbor embedding, is a dimension reduction technique developed by Geoff Hinton (a god in deep learning) and his students that is very useful for visualizing high dimensional data in 2-dimensions. If you are interested in how it works, put briefly the algorithm creates a probability distribution over pairs of high-dimensional objects in a way so that similar objects (as determined by Euclidean distance) have a high chance of being picked whereas dissimilar objects have a low chance of being picked. The algorithm then finds the probability distribution in 2-dimensions that is most similar to this distribution, i.e. minimizes the Kullback-Leibler divergence. To read more, go to `https://lvdmaaten.github.io/tsne/`. For this part of the homework, you will create visualization for the answers produced in random_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 answers method will produce a dictionary of answers and their embeddings, which you will need to convert to a matrix of dimensions num_words *times* dim. Then, run TSNE on the answer matrix as follows:

If using the TSNE Python library:

```
X_reduced = tsne(X, no_dims=2, initial_dims = 100, perplexity = 30.0)
```

And if using scikit-learn:

```
import sklearn.manifold
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 a X11 server (as we did in HW2)
2. Create graphs from your local machine (the files are small so this should be fine)
3. Use a 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.

VI. Submission

You should submit your assignment through your hidden directory on Zoo. Once you are done with the homework and have finalized the README.txt file, check once again that this is the path for your homework:

`~/hidden/<YOUR_PIN>/Homework3/`

Submit the following files. **Please follow this folder structure exactly; otherwise you may lose points for otherwise correct submissions!**

`~/hidden/<YOUR_PIN>/Homework3/logisticRegression.py`
`~/hidden/<YOUR_PIN>/Homework3/dependencyData.py`
`~/hidden/<YOUR_PIN>/Homework3/dependencyRNN.py`

```
~/hidden/<YOUR_PIN>/Homework3/buildWord2Vec.py
~/hidden/<YOUR_PIN>/Homework3/tsne_visualization.png
~/hidden/<YOUR_PIN>/Homework3/README.txt
```

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

```
/home/classes/cs477/bash_files/hw3_set_permissions.sh <YOUR_PIN>
```

Make sure the command above runs without errors, and **do not make any changes or run the code again**. If you do run the code again or make any changes, you need to run the permissions script again. Submissions without the correct permissions may incur some grading penalty.