- **Introduction**

In this assignment, we will:
1. Develop simple RNN based encoder decoder network for the task of machine translation from chinese to english.
2. Add attention mechanism to improve the network from part 1

This assignment is based on the following papers:

1. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation
2. Neural Machine Translation by jointly learning to align and translate

Paper 1 Link: https://arxiv.org/abs/1406.1078
Paper 2 Link: https://arxiv.org/abs/1409.0473

The equations for part 1 & 2 of the assignment are covered in the appendix of the above research papers.

- **Data**

The data for this assignment are sentences in chinese and english languages. The data folder contains the following files:

1006 dev.src
1006 dev.tgt
506 test.src
44016 train.src
44016 train.tgt

The first column is the number of sentences in each file. The format of each file is one sentence per line.

- **ASSIGNMENT INSTRUCTIONS**

The assignment directory contains the following files:

- bleu.py - computes the bleu score
- utils.py - utilities to read, write, create dictionaries from the corpus
- nmt_dynet.py - simple neural machine translation system (Part 1)
- nmt_dynet_attention.py - neural machine translation system with attention (Part 2)

In Part 1, you are required to implement the following methods:

__init__
encode
get_loss
generate

In Part 2, you are required to implement the following methods:

__init__
encode
get_loss
generate
attend

To run your implementation, use the following commands:

```
python nmt_dynet.py ../data/train.src ../data/train.tgt ../data/dev.src ../data/dev.tgt
../data/final_nmt_dynet --dynet-mem 5000
python nmt_dynet_attention.py ../data/train.src ../data/train.tgt ../data/dev.src ../data/dev.tgt
../data/final_nmt_dynet_attention --dynet-mem 5000
```

If you run out of memory, you can increase the dynet-mem from 5000Mb to some other value.

- **Neural machine translation**

Most of the theory for this assignment is covered in a series of blogs by one of the authors of the papers mentioned in the introduction. Thus we are not covering detailed theory in the write-up.

Thus we highly recommend everyone to read these blogs before you start to get your hands dirty with the actual implementation. See below the link to these blogs. We have also included a combined pdf of these blogs in case you would like to read it offline.

https://devblogs.nvidia.com/parallelforall/introduction-neural-machine-translation-with-gpus/
https://devblogs.nvidia.com/parallelforall/introduction-neural-machine-translation-gpus-part-2/
https://devblogs.nvidia.com/parallelforall/introduction-neural-machine-translation-gpus-part-3/

From here on, we assume that you have read the above three articles.

- **Part 1 of the Assignment - 40 points**

In this part of the assignment, you will be implementing the simple encoder decoder network covered in parts 1 & 2 of the above blogs and the first paper. There is a small difference between the paper and this part in that we are asking you to implement bidirectional RNN instead of unidirectional RNN. The encoder will be bidirectional RNN with GRU cells and the decoder will be an unidirectional RNN with GRU cells. The encoder will work the same while training and translation. The decoder will use the parallel corpus during the training part to reduce the loss function and during translation, it will predict the output sentence one word at a time. To obtain the final hidden states of the bidirectional RNN, you can concatenate the hidden states of the forward and backward RNNs as shown below:

$$h_i = \begin{bmatrix} \overrightarrow{h}_i \\ \overleftarrow{h}_i \end{bmatrix}$$

You need not implement the GRU cell from scratch and use the GRUBuilder available in Dynet.

- **Implementation Details for Part 1**

Dynet provides GRUBuilder whose initialization is similar to that of LSTMBuilder. Thus you can initialize GRUBuilder as follows:

`GRUBuilder`**(unsigned *layers*, unsigned *input_dim*, unsigned *hidden_dim*, Model &*model*)**

Next, you need to figure out how you will implement the encoding part. Dynet again provides **add_input()** function so you can call this method to process one word at a time.

You need to concatenate the hidden states of the forward and backward lstms and use the last concatenated hidden state as input to the decoder.

To compute the loss, you will use decoder’s previous state, output of the encoder and the previous word to predict the current word. The loss will then be computed based on the predicted and actual words in the target sentence.

For translation, you will again predict one word at a time till you encounter end of sentence marker.

- **Part 2 of the Assignment - 60 points**

In part 2 of the assignment, you will be implementing the attention mechanism introduced in the second paper and covered in the third part of the above blogs. If you remember part 1, the decoder just takes the final hidden state of the encoder and uses that to decode the target sentence. That is not the case with attention. In this case all the hidden states of the encoder are used to decode the target sentence.

The encoder in this case will again be a bidirectional RNN with GRU cells and the hidden states of all the GRU cells will be used by the decoder to decode the target sentence. The decoder will again be an unidirectional RNN with GRU cells.

You need not implement the GRU cell from scratch and use the GRUBuilder available in Dynet.

- **Implementation Details for Part 2**

The encoding in Part 2 is quite similar to part 1. During decoding, you will be using all the hidden states of the encoder to compute the context vector which will then be used along with the previous hidden state of the decoder and previous word to predict the current word.

Here again, you can leverage functions like **add_input()**, **concatenate()** etc provided by DyNet.

As mentioned earlier, the equations for both the parts are covered in the appendix of the research papers.

- **Grading scheme**

You should be able to achieve the following BLEU scores in part 1 and 2 of this assignment:

Part1	Score: 22
Part2	Score: 25

You will need to submit the saved models as well for each part. **Do not remove the code that writes the models to the file.**

- **Extra Credit**

In this assignment, we are giving you the opportunity to earn extra credit by trying the following improvements:

- Use pretrained embeddings. In the skeleton code provided to you, the embeddings are initialized to random values. Instead, you can find pre-trained embeddings for the source and target languages and use them to show improvement in the BLEU score - 10 points
- Use Beam Search instead of Greedy decoding. We have implemented Greedy decoding for you. To earn more extra credit, you can implement beam search and show improvement in the BLEU score - 30 points
- We have provided a small set of test sentences for which the corresponding sentences in target language are not provided. In this part, top 20 students will obtain extra 10 points. Your position will be determined on the basis of the final BLEU score achieved by your network after you submit the assignment. You will need to submit your final model as well in order to earn extra credits in this part

Introduction

In this assignment, we will:

1. Develop recurrent neural networks (RNNs) and multi-layer perceptrons (MLPs) for language modeling
2. Implement a graph-based parsing algorithm and use our previously developed neural networks for scoring possible parses and labels

This document includes the following sections:

Section A: Background

a) Introduction to deep learning

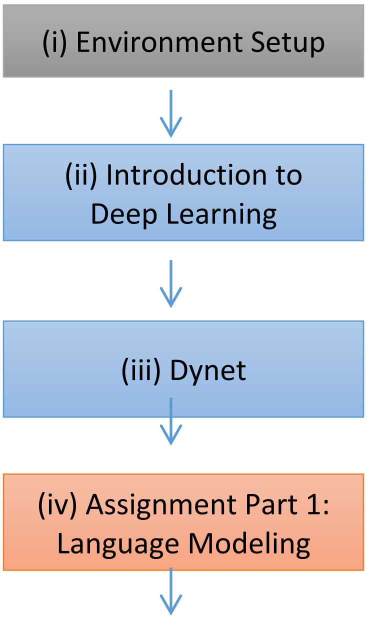
b) Implementing an RNN language model using Dynet

Section B: Implementation of Deep Learning for Dependency Parsing

c) Implementing a graph-based dependency parsing algorithm

d) Experimenting with hyperparameter settings and visualizing 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) Dynet

Introduction to and tutorial on Dynet.

(iv) Assignment Part 1: Language Modeling

Implement RNN language models using Dynet.

(v) Assignment Part 2: Dependency Parsing

Implement a recurrent neural network for dependency parsing. Improve the performance by tuning parameters.

(vi) Submission

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

BACKGROUND INFORMATION

(ii) Introduction to Deep Learning

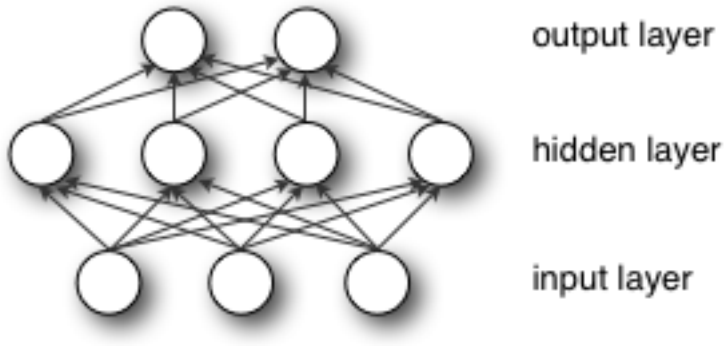
Deep learning is currently a very popular topic in NLP. Deep learning approaches have very recently obtained state-of-the-art performance across many different NLP tasks. The appeal of deep learning is enhanced by the idea that tasks can be modeled without extensive feature engineering[1]. The term "deep 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 graph-based dependency parsing approach used by Kiperwasser and Goldberg

[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.

(2016).

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



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:

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

where $W \in R^{d_n x d_{n-1}}$ and $b \in R^{d_n}$ 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[2].

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[3]. 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)[4]. Adam 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

[2] 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.

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

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

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. Previously in this class we studied HMMs, where the hidden states are discrete variables. In the recurrent neural network, the hidden state is a continuous variable.

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, we may allow the embeddings to be changed via backpropagation.

For this project, you will be implementing 1) a recurrent neural network for language modeling and 2) a graph-based dependency parsing algorithm: https://arxiv.org/pdf/1603.04351.pdf.

The high-level overview of the assignment is as follows:

1) Implement a language model
    a. Implement two versions of a recurrent neural network for character prediction and generation: a simple RNN and the Long Short-term Memory network (LSTM) – 15 points
    b. Implement a Multi-Layer Perceptron for predicting the next character – 5 points
2) Implement a graph-based dependency parser using an LSTM
    a. Implement the pre-processing of the input data, including normalizing and sampling text – 10 points
    b. Implement Eisner's algorithm for finding the MST of a dependency matrix – 50 points
    c. Report the differences with and without POS embeddings for English and Korean and graph the accuracy over time – 5 points
    d. Visualize the embeddings by performing dimensionality reduction using Principle Components Analysis (PCA) – 15 points

IMPLEMENTATION

Overview

You will be implementing parts 1 and 2 using Dynet, a numerical computation library in python.

The software depends on the following 4 libraries

numpy
dynet
scikit-learn
matplotlib

If you really want to understand Dynet, you should work through the following tutorial:
[http://dynet.readthedocs.io/en/latest/tutorial.html - python-tutorial](http://dynet.readthedocs.io/en/latest/tutorial.html#python-tutorial)
For this assignment, we will cover what you need to know.

Provided data

The data is the same as Homework 2. We will be running experiments on English, Korean, and Swedish for this assignment.

ASSIGNMENT INSTRUCTIONS

For this assignment, we have provided the following files and directories:

```
envsetup.sh
data/
decoder.py
eval.pl
graphParser.py
layers.py
output/
part1.py
part2.py
utils.py
```

1) Language modeling (20 points)
   As a warmup, you will implement a language model using Dynet.
   a) Implement a Multi-Layer Perceptron – 5 points

      In the file `layers.py`, you will implement the `get_output` method for the

`MLP` class. This method takes in one parameter: a vector. The output is the result of the MLP calculation.

Recall the formula for the MLP:

$$h = g(W_h \cdot x + b_h)$$

$$o = f(W_o \cdot h + b_o)$$

The hidden layer *h* is the output of a nonlinearity *g* after an affine transformation of the input *x*. The output *o* is then another nonlinearity *f* after a different affine transformation of *h*. This is a 1-layer MLP. Alternatively, we could add another hidden layer in between *h* and *o* (2-layer MLP) or remove the hidden layer and feed the input *x* to the output layer directly. (The *W* matrices and *b* vectors are the parameters of the network that we need to learn).

Your function should allow an arbitrary number of hidden layer (including 0) and any of the available activation functions. (A `nonlinearities` dictionary is defined within `layers.py` for your convenience).

There are some issues to be aware of when using Dynet:
a. If you are familiar with numpy, the API may be confusing because * is dot product and cmult is element-wise multiplication (see the API here for details: (http://dynet.readthedocs.io/en/latest/tutorials_notebooks/API.html)
b. Dynet builds a dynamic computation graph, which is different from other deep learning libraries you may be familiar with. It may not matter for this assignment but it could potentially give you some confusing error messages.

b) Implement 2 variations of a Recurrent Neural Network – 15 points

The next step will be implementing the function `get_output` for the `SimpleRNN` and `LSTM` class in `layers.py`. This function will take one required input: a matrix or list of vectors. The output is a list of lists, the items in each list is a list of the hidden states or other parameters at each time step.

A Simple RNN takes a hidden state vector (a history of previous inputs) and combines it with the current input to create a new hidden state vector. Formally, the equation is as follows:

$$h_t = f(W_h x_t + U_h h_{t-1} + b_h)$$

The model is not too different from the MLP, except we are now adding a hidden state term. The parameters of this model are the matrices *W* and *U* and the vectors *b* and *h_0*.

Implement a `get_output` method that **returns** all the hidden states for a sequence of inputs. Use **tanh** (hyperbolic tangent) as the activation function *f*. Your method should also allow for iterating over the input in **reverse.** Remember, you must **return** the list of states as a **singleton** list (it will become clear why in the next part of this section).

To test your MLP and Simple RNN, run the following command:

```
python part1.py simple_rnn
```

The output will look something like this (after a few training rounds, the text will begin to look coherent):

```
146.193786621 xeetvsn edfrv
95.9972305298 z tvxu qppwqea efeazp
60.8372612 n
34.6215171814 a  uiu  hoon ffputu emobzotde bqkagtiguioq box
15.531124115 bzqfn
5.22873735428 a quick brown fyv jumped over tve lazy aog
1.64150714874 a quick brown fox jumped over the lazy dog
0.928709506989 a quick brown fox jumped over the lazy dog
0.649986147881 a quiak brown fox jumped over the lazy dog
0.499105513096 a quick brogn fox jumped over the lazy dog
0.404530018568 a quick brown fox jumped over the lazy dog
0.339753985405 a qrick brown foxnjumped over the lazy dog
0.292638242245 a quick brown fox jumped over the lazy dog
0.25684723258 a quicj brown fox jumped over the lazy dog
0.22875213623 a quick brown fox jumped over the lazy dog
```

One application for RNNs is language modeling. Unlike our previous assignments with words, this model is trained to predict the next **character** in a sequence of text. At each timestep, we use the output of the RNN, and input into an affine transformation with a **softmax** activation function to predict the most likely character:

$$p(c) = soft\max(W_o h_t + b_o)$$

which gives us a probability vector over our character alphabet.

(This is actually where we would use beam search in practice! The probability of the next character depends on all previous hidden states so we don't have the Markov property that allows us to do Viterbi decoding)

However, the simple RNN is limited because it favors the most recent information so long-distance dependencies are a problem. This is where the Long short-term memory (LSTM) network comes in. Unlike the RNN, which just considers the previous state, we allow the model to store information about past states as follows:

$$
\begin{aligned}
f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \\
h_t &= o_t \circ \sigma_h(c_t)
\end{aligned}
$$

f_t is the ***forget*** gate (the weight of remembering old information), *i_t* is the ***input*** gate (the weight of new information), and *o_t* is the ***output*** gate (the weight of the information we decide to leak). *sigma_g* and *sigma_c* are the **sigmoid** and **tanh** activations, respectively. At each timestep, we calculate *sigma_c* applied to the hidden state update (note that this is essentially our entire hidden state update from before). A new cell value is calculated from a weighted combination of the previous cell and this new update, before we release some of the information from a non-linear function applied to the current cell, and store that as the current hidden state.

Implement a `get_output` method that **returns** all the hidden states and cells (in that order) for a sequence of inputs. Use **tanh** (hyperbolic tangent) as the activation function *sigma_h*. As before, your method should also allow for iterating over the input in **reverse.**

To test your LSTM, run the following command:

```
python part1.py lstm
```

The output will look something like this:

```
147.217041016 rykwztzdatzmbthbzuworqjeeriamyj  moxnuqghl
127.199874878 ofmfcjxwveezp qmn
115.247550964 paebc uaebrrf op
100.045349121 a vmatk  gwpoog uwu  oteq dol
84.2714157104 bpkr dzvrlrx bhziw mzc lvw ohee bk gove  tre tza d omdmne gollwr tkgth hco
68.0308990479 yodwg do
53.986743927  knc fir erwox ze z gb
42.1806297302 uqa ubu pia roog
30.3184890747 a uibron frowvee fox hel aver dover thhe lazz doo
20.4457950592 a q cickr oxn jox
14.6613349915  qjck broww ox tummped over laz dovpe dzoe drog
9.1230764389 a quickk brown fog
5.64252853394 a qrovk bfov fox jumjped over the lazy do
3.23011732101 a quick brown fox jumped mer the tahe lad dog
2.02552962303 a quick brown fox jumped over they lazy dog
1.51882219315 a quick brown fox jumped over the lazy dog
1.20694196224 a quick brown foxy jumped oghe lazy dog
0.99672627449 a juick brox fox jumped over tthe lazy dog
0.846131503582 a quick brown fox jumped over the lazy dog
0.733343243599 a quick down fox jumped over the layy dog
0.645937085152 a quick brrown fox jumped over the lazy dog
0.576351165771 a quiick brown fox jumped over the lazy dog
0.519733607769 a quick brown fox jumped over the lazy dog
0.472820192575 a quick brown fox jumped over the lazy dog
0.433358311653 a quick brown fox jumped over the lazy dog
0.39973038435 a qdick brown fox jumped over the lazy dog
0.37075433135 a quick brown fox jumped over the laz dd oved owe lazy dog
0.345535248518 a quick brown fox jumped over the lazy dog
0.323411881924 a quick bron fox jumped over the lazy dog
0.303843677044 a quick brown fox jumped over the lazy dog
0.286423802376 a quick brown fox jumped over the lazy dog
```

```
0.270822525024 a quick brown fox jumped over the lazy dog
0.256773948669 a quick brown fox jumped over the lazy dog
0.244056671858 a quick brown fox jxmped over the lazy dog
0.232499971986 a quckc brown fox jumped over the lzzy dog
0.221947789192 a quick brown fox jumped over the lazy dog
0.212278395891 a quick brown fox jumped over the lazy dog
0.203388243914 a quick brown fox jumped over the azy dog
0.195187672973 a quick brown fox jumped over the lazy dog
0.18759906292 a quick brown fox jumped over the lazy dog
```

The Simple RNN actually learns quicker than the LSTM. This is because the training data is very simple (one sentence with no repeated letter bigrams), so the model with fewer parameters can easily memorize the data. This RNN language modeling example was shamelessly stolen from here: http://dynet.readthedocs.io/en/latest/tutorials_notebooks/RNNs.html

2) Graph-based dependency parsing – 80 points

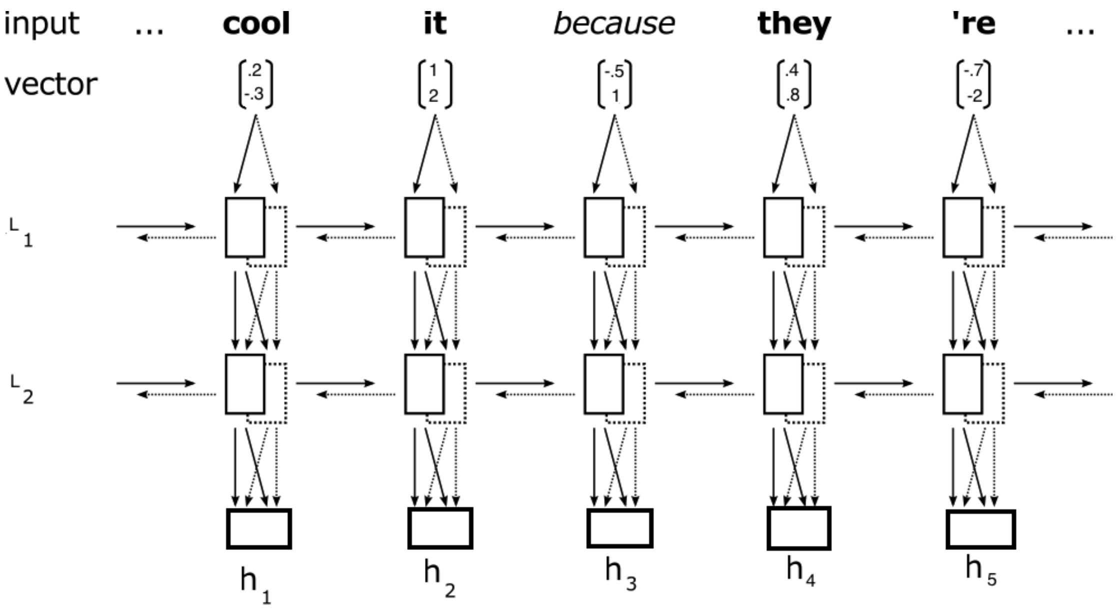
In the previous assignment, we worked on transition-based dependency parsing. An alternative to this approach is to create a score function that considers $n^2 + n$ possible arcs simultaneously. This approach considers all words in the sentence and the ROOT node and scores every pair of valid possible arcs. Then we use a dynamic programming algorithm to find the dependency tree with the highest sum of scores (a maximum spanning tree). Given the dependency tree, we can also create a score function that predicts dependency relations for each arc.

Remember that in Homework 2, we used a combination of contextual features (the parser configuration) and input features (words and POS)[5]. Without transition-based parsing, however, we don't have the parser configuration so we need to consider other contextual features.

Fortunately, the LSTM is such a model that provides context. The LSTM runs over the words in the sentence and the ROOT and gives us a feature vector (hidden state) at each time step. We can then concatenate the hidden states h_i and h_j and use an MLP to create a score for whether there should be an arc between the inputs at I and j. We can also use a separate MLP to predict the label of the dependency relation.

Similar to how we can layer an arbitrary number of hidden layers in an MLP, we can have additional layers for an LSTM. Furthermore, we are not limited to forward LSTMs, but can consider iterating over the input in the reverse direction. Consider the graphic that displays a 2-layer bi-directional LSTM:

[5] Note that these features are sparse and binary (present or not) and don't necessarily generalize well. The motivation behind using word embeddings and neural models is that words behave similarly so they should be near to one another in vector space rather than having all words at the maximum distance apart.



The input words are converted to **embeddings** and then passed into a forward and backward LSTM. Then the output from both the forward and backward LSTM at each time step is fed into another layer of a forward and backward LSTM, before we finally obtain a hidden state representation for each time step. Consider the flow of information through the computation graph. After the second LSTM layer, h_1 has seen the entire sentence in both directions.

a) Preprocessing – 10 points

At this stage, we need to convert the words, POS, and dependency relations to indices. Preprocessing also includes handling rare words, as we did in Homework 1. In order to handle problematic words, we will do the following:

i) Lower-case all words in the vocabulary

ii) Replace all numbers with the NUMBER token (you need to implement a `normalize` method that does i and ii)

iii) Replace words not in the vocabulary (`self.word2idx)` with the UNKNOWN token.

iv) Randomly replace words with the UNKNOWN token with probability $\frac{0.25}{0.25+\#w}$ where w is the count of word w (we do this because otherwise there will be significantly more UNKNOWN tokens in the test set than in the training set). If the parameter `deterministic` is set to true, don't do sampling.

Implement the preprocessing in the `process` method of the `Vocab` class in `utils.py`. This method takes in a list of lists, where the elements are of type ConllEntry. This method should return word indices, POS indices, arcs, and labels.

What are some other preprocessing steps you might want to take? **List at least 2 possible methods** in your README. (Hint: the methods we used were transformation-based (treating all numbers as the same and lower casing words) and count-based (down-sampling infrequent words).

b) Eisner's algorithm – 50 points

Eisner's algorithm for decoding is a dynamic programming algorithm for finding the optimal dependency parse, given scores for every possible arc (Described in depth here: http://www.cs.jhu.edu/~jason/papers/eisner.coling96.pdf)

Implement this algorithm in the `parse_proj` method of `decoder.py`. You may find this website useful, in particular the Pseudocode section at the bottom: http://curtis.ml.cmu.edu/w/courses/index.php/Eisner_algorithm, as well as the file Graph-based.pdf uploaded on the course website.

Modify the algorithm to do **loss augmented inference**. Given the gold standard arcs, we can encourage the parsing algorithm to select the correct parse and avoid overfitting by adding 1 to the scores of known **incorrect** arcs during training. More specifically, add 1 to scores[I,j] if there is an arc from j to i.

Discuss any practical concerns you needed to consider in your README. Alternatively, discuss any suggestions you have for improving this assignment (such as clearer instructions, but be specific).

c) Experiments and graphs for POS embeddings and languages – 5 points

Run experiments using this model:

```
python part2.py data/english/train.conll data/english/dev.conll output model
```

Report the LAS and UAS results after the final epoch in your README.

Run experiments using part-of-speech embeddings. In the basic model, the input to the bi-directional LSTM is the word embedding for the word at that time step. We can also concatenate an embedding for the part-of-speech for that word, and as we are using a very small dataset, this might give us even more generalization power.

Run the experiments as follows:

```
python part2.py data/english/train.conll data/english/dev.conll output model_pos --
pos_d 25
```

Report the LAS and UAS results after the final epoch in your README.

Also, **create a visualization** of the LAS and UAS scores over each epoch. Compare the results with and without POS embeddings, and save these files in `word_accuracy.png` and `pos_accuracy.png`. You may want to modify the `metrics` function in `utils.py` for this purpose.

Finally, report the results of experiments on Korean and Swedish:

```
python part2.py data/korean/train.conll data/korean/dev.conll output model_korean --pos_d 25
```

```
python part2.py data/swedish/train.conll data/swedish/dev.conll output model_swedish --pos_d 25
```

Add the **Korean and Swedish final epoch results** to your README.

d) PCA visualization of answers – 15 points

PCA is a statistical technique that can be used for dimension reduction (https://en.wikipedia.org/wiki/Principal_component_analysis). It is often useful for visualizing high dimensional data in low dimensions (usually 2). For this part of the assignment, you will create visualizations for the embeddings produced in `<model>_embeddings.json`

Write a script, `pca.py`, to make a scatterplot of the embeddings in 2D space. The embeddings are in JSON format, consisting of a dictionary. To access the word and POS embeddings, respectively, use 'word' and 'pos' as the key to the dictionary. This will produce another dictionary of answers to their embeddings, which you will need to convert to a matrix. Then run PCA on the embedding matrix (but make sure to keep track of which words match up with which embeddings):
http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

Then plot the results using a plotting library such as **matplotlib** to make a scatterplot with ***each point represented on the scatterplot as its corresponding 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

Create visualizations for the English **parts-of-speech** and **verbs** (a verb is any word that appears as a verb in the training data). Note that you should **run PCA on all the word embeddings**, but only plot the verbs. After you have created these visualizations, save them to files called `pos_visualization.png` and `verb_visualization.png`.

Deliverables

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

1. MLP and RNN/LSTM code:
   `$HW3_ROOT/layers.py`
2. Input handling code:
   `$HW3_ROOT/utils.py`
3. Eisner’s algorithm:
   `$HW3_ROOT/decoder.py`
4. Graphs of accuracy over time
   a. `$HW3_ROOT/word_accuracy.png`
   b. `$HW3_ROOT/pos_accuracy.png`
5. PCA visualization of answers
   a. `$HW3_ROOT/pca.py`
   b. `$HW3_ROOT/pos_visualization.png`
   c. `$HW3_ROOT/verb_visualization.png`
6. README file containing results and discussions
   `$HW3_ROOT/README.txt`