Assignment 3: Multi-Layer Perceptrons and Recurrent Networks with PyTorch

LING 380/780, Fall 2021
Neural Network Models of Linguistic Structure

Assigned: Monday, November 8
Due: Wednesday, November 17, 11:59 PM
Total Points: 100

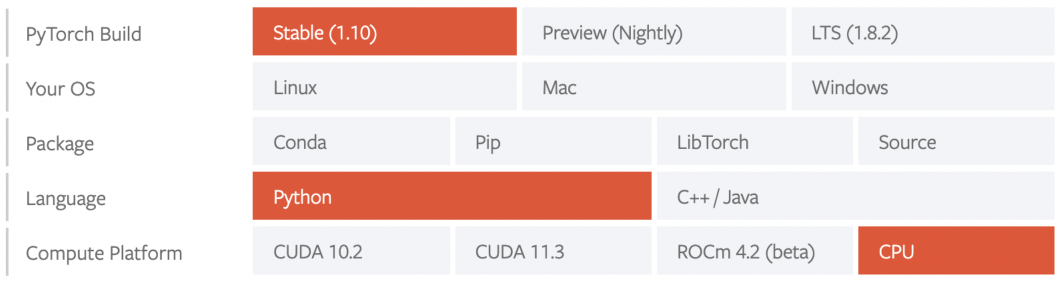
In this assignment, you will improve upon the part-of-speech (POS) tagging model from the previous assignment in two ways. Firstly, you will use a bidirectional recurrent architecture that, unlike the n-gram based multi-layer perceptron (MLP), can take into account a potentially unlimited amount of context on both sides of an input token. Secondly, you will implement your models using PyTorch, a widely-used and highly optimized library for implementing neural network models.

To start you off, we will have you do a number of basic PyTorch exercises that will teach you the lay of the land. Then, you will build two neural network models for POS tagging. The first of these replicates what you did on the previous assignment, but in PyTorch. The second model will use a recurrent network for the same task. This time, much of your work will involve writing code using the PyTorch package. As in the previous assignments, your answers and code need not be very long, but will require thought.

After the PyTorch crash course, the next part of the assignment involves the implementation of the models and training/testing procedures. In the last part, you will explore these models and their representations, and evaluate their effectiveness on a range of data.

1 Installation

For this assigment, you will be using a number of software packages.



PyTorch Build	[✓ Stable (1.10)]	Preview (Nightly)	LTS (1.8.2)	
Your OS	Linux	Mac	Windows	
Package	Conda	Pip	LibTorch	Source
Language	[✓ Python]	C++ / Java		
Compute Platform	CUDA 10.2	CUDA 11.3	ROCm 4.2 (beta)	[✓ CPU]

Figure 1: If you wish to do a custom installation of PyTorch using the instructions from the official website, please make sure the settings shown above for "PyTorch Build," "Language," and "CPU" are selected.

1.1 PyTorch

PyTorch is a Python library for creating, training, and running neural networks. Before attempting the assignment, please install the latest stable version of PyTorch (version 1.10). Your installation of PyTorch does not need to support parallel computation using graphics processing units (GPUs), and you do not need the torchvision or torchaudio packages.

If you have Anaconda installed on your computer, then you can install PyTorch using the `conda` package manager as follows.

```
# Windows and Linux Users
conda install pytorch cpuonly -c pytorch

# Mac OS Users
conda install pytorch -c pytorch
```

Windows and Mac OS users can also install PyTorch using the `pip` package manager.

```
# Windows and Mac OS Users
pip install torch
```

If none of these use cases applies to you (e.g., if you wish to install PyTorch on a Linux system using `pip`), please follow the installation instructions at `https://pytorch.org/get-started/locally/` using the settings shown in Figure 1.

1.2 Other Packages

In addition to PyTorch, you will need to have the software for the previous assignments installed, including NumPy and PyConll. You will also need the following packages.

- scikit-learn, a library that provides various data science and machine learning utilities. We will use scikit-learn to perform dimensionality reduction using PCA. If you have Anaconda installed, then you should already have scikit-learn installed. If you used miniconda instead, you can install scikit-learn as follows:

  ```
  conda install -c anaconda scikit-learn
  ```

  Otherwise, you can install scikit-learn using `pip` as follows:

  ```
  pip install scikit-learn
  ```

  or by following the instructions at https://scikit-learn.org/stable/install.html.

- The Natural Language Toolkit (NLTK), a library that provides various NLP and computational linguistics utilities. We will use a tokenizer from NLTK. If you have Anaconda installed, then you should already have NLTK installed. If you used miniconda, you can do the following:

  ```
  conda install -c anaconda nltk
  ```

  Otherwise, you can install NLTK using `pip` as follows:

  ```
  pip install nltk
  ```

  or by following the instructions at https://www.nltk.org/install.html.

  After NLTK has been installed, you will need to additionally download the Punkt tokenizer. To do this, open a Python console, Jupyter Notebook, or Python script and execute the following code.

  ```
  import nltk
  nltk.download("punkt")
  ```

2 PyTorch Exercises

The following exercises are designed to familiarize you with using PyTorch. You will be asked to describe the behavior of certain PyTorch functions and code snippets in plain English. You may choose to run the code snippets in the Python console, a Python script, or a Jupyter Notebook. You may also consult the PyTorch documentation at https://pytorch.org/docs/stable/index.html and the official tutorials at https://pytorch.org/tutorials/, as well as the course materials. Each code snippet assumes that all previous code snippets have already been run. Therefore, you must run the code snippets in the same order as they appear in the instructions.

To begin, please import NumPy and PyTorch, as follows.

```python
import numpy as np
import torch
```

2.1 Tensors

The basic object of PyTorch is the *tensor*. Please read the file `pytorch_backprop.pdf` in the `notebooks` folder of the “Files” section of Canvas. This file contains the outpout from the Jupyter Notebook presented during lecture on October 5.

Problem 1. What kind of object does a tensor represent? What do the `.grad` and `.requires_grad` properties of a tensor represent?

Problem 2. Please create some tensors using the following code.

```python
a = np.array([[[1, 2, 3], [4, 5, 6]],
              [[7, 8, 9], [10, 11, 12]]])
b = torch.Tensor(a)
c = torch.LongTensor(a)
d = torch.tensor(a)
e = torch.tensor(a, dtype=torch.float)
f = torch.Tensor(2, 3)
```

What is the difference between `b`, `c`, `d`, `e`, and `f`?

Problem 3. Please run the following code.

```python
print(torch.full((2, 3), 5))
print(torch.randn(2, 3))
```

What do `torch.full` and `torch.randn` do?

Problem 4. Please run the following code.

```python
b.requires_grad = True
c.requires_grad = True
```

One of these lines of code should work; the other should raise a RuntimeError. Why are PyTorch tensors designed this way?

Hints:

1. Think about what `.requires_grad` represents.
2. Recall that a function is differentiable only if it is *continuous*. That is, if f is differentiable, then for all x,
$$f(x) = \lim_{w \to x} f(w).$$

Problem 5. Many NumPy array operations will work on PyTorch tensors, such as `+`, `-`, `*`, `/`, `@`, and `.T`. However, there are some minor differences between array operations and tensor operations. Please run the following lines of code.

```python
print(a.sum(axis=-1))
print(a[:, np.newaxis].shape)
print(a.reshape(4, -1))
print(a.size)
```

What is the equivalent of the above code for PyTorch tensors? Please give your answer as a 4-line code snippet that applies to `b` the tensor operations that are analogous to the array operations shown above.

Problem 6. Please run the following code.

```python
b = torch.ones(2, 3)
c = torch.full((2, 4), 5)
d = torch.cat([b, c], dim=-1)
print(d)
```

What do `torch.ones`, `torch.full`, and `torch.cat` do?

Problem 7. Please run the following code.

```python
d.flatten()[:] = torch.arange(d.numel())
e = d.unfold(-1, 3, 1)
print(d, d.shape)
for i in range(e.size(1)):
    print(e[:, i])
```

What does `.unfold` do? Describe the tensor `e`.

2.2 Modules

PyTorch *modules* represent neural network layers as well as full neural network models. Modules are equivalent to the `Layer` class from Assignment 2. Like the `Layers` from Assignment 2, each module has a `.forward` and `.backward` function.

Modules are contained in the `torch.nn` package. For the following exercises, please import the `torch.nn` package as follows.

```python
import torch.nn as nn
```

By convention, the `torch.nn` is referred to using the alias `nn`.

Problem 8. Please run the following code.

```python
lin1 = nn.Linear(2, 3)
lin2 = nn.Linear(3, 4)
model = nn.Sequential(lin1, nn.Tanh(), lin2)
```

Describe `model`. What kind of neural network is it?

Problem 9. RNNs are implemented using the `nn.RNN`, `nn.LSTM`, and `nn.GRU` modules. Please run the following code.

```python
# Create an Embedding layer
embedding_layer = nn.Embedding(100, 20)

# Create an LSTM
lstm = nn.LSTM(input_size=20, hidden_size=9,
               bidirectional=True, batch_first=True)

# Create a fake input
x = torch.randint(100, (5, 7))

# Run the LSTM
embeddings = embedding_layer(x)
h, _ = lstm(embeddings)

print(x.shape)
print(embeddings.shape)
print(h.shape)
```

Describe `x`, `embeddings`, and `h`. What do each of their dimensions represent?

Problem 10. In Assignment 2, you needed to implement `.backward` for each layer. During training, you needed to call `.backward` for the `CrossEntropyLoss` and use its output as an argument for the `.backward` of `MultiLayerPerceptron`. You do not need to do any of this in PyTorch.

Please run the following code.

```python
# Create a fake input and output
x = torch.randn(5, 2)
y = torch.randint(4, (5,))

# Create a loss function
loss_function = nn.CrossEntropyLoss()

# Run the forward pass on model
```

```python
logits = model(x)
loss = loss_function(logits, y)
```

How would you run the backward pass for the mini-batch `(x, y)`?

Hints:

1. Your answer should consist of a single line of code.
2. After running your one line of code, the following loop should print the gradients of all of `model`'s parameters. None of the gradients should be `None`.

```python
for p in model.parameters():
    print(p.grad)
```

3 Model Architecture Definitions

In the next part of the assignment, you will implement two different kinds of neural networks for POS tagging: an n-gram-based MLP model and a bidirectional recurrent model. As in the previous assignment, we will make use the POS tag set that is utilized in the Universal Dependencies treebank:

ADJ: adjective	PART: particle
ADP: adposition	PRON: pronoun
ADV: adverb	PROPN: proper noun
AUX: auxiliary	PUNCT: punctuation
CCONJ: coordinating conjunction	SCONJ: subordinating conjunction
DET: determiner	SYM: symbol
INTJ: interjection	VERB: verb
NOUN: noun	X: other
NUM: numeral	

In the previous assignment, the inputs to the model were represented as integer-valued arrays of shape $(\text{batch size}, n\text{-gram length})$. Each row of the array represents one of the inputs in the batch, and the values in the array are numerical indices assigned to tokens by the vocabulary. This representation is not appropriate for this assignment, however, because the input to the RNN-based model is an entire sentence rather than an n-gram. We will instead represent inputs as arrays of shape $(\text{batch size}, \text{sentence length})$. When sentences of different lengths appear the same batch, the shorter sentences are *padded* on the right by special `[PAD]` tokens that you should think of as being equivalent to `[EOS]` tokens. To that end, we have added a `[PAD]` token to both the input vocabulary and the output vocabulary; the dataset assigns the `[PAD]` label to `[PAD]` tokens from the input, though we will not actually train our models to do this.

For the next few problems, you will be filling in the model definitions in the `models.py` module. The pattern for defining neural network models in PyTorch is largely similar to the pattern used in Assignment 2. To define a custom architecture, you need to create a subclass of `nn.Module`, which must implement a `forward` function. However, unlike the `Layer` class from Assignment 2, you do not need to implement the `backward` function. Instead, the backward pass is implemented automatically by the tensor data structure. This shows the benefits that automatic differentiation can provide for rapid prototyping.

3.1 Multi-Layer Perceptron

The MLP model is identical to the model from Assignment 2: it consists of an embedding layer and an output layer, as well as any number of hidden layers in between. However, because the input to the model is now in a different format, the MLP's `forward` function will need to "unfold" the input by dividing the sentences of the batch into n-grams.

For the next two problems, you will complete the definition of the class `MLPPosTagger`, which defines the MLP architecture.

Problem 11. In PyTorch, the *initializer* (i.e., the method `__init__`) of a module is responsible for defining all the layers comprising the model. In the initializer for `MLPPosTagger`, we have provided code that creates an `nn.Embedding` layer and loads pre-trained embeddings if they are provided as a keyword argument. Please complete the definition of `MLPPosTagger.__init__` by constructing the hidden and output layers of the network. The hidden and output layers should be combined into an `nn.Sequential` module, and saved to an instance variable called `self.layers`.

Problem 12. Implement the forward pass for the MLP model by filling in the function `MLPPosTagger.forward`.

Unlike the network from the previous assignment, this network object should take as input a batch of sentences; i.e., a tensor of shape $(\text{batch}, \text{sentence length})$. Before applying the relevant embedding, linear, and tanh layers, your `forward` method will need to perform the following operations on this tensor so that each sentence is broken into n-grams.

- First, you need to pad the input with `[BOS]` tokens on the left and `[PAD]` tokens on the right. The number of `[BOS]` and `[PAD]` tokens you need to add is determined by the following requirements:
    1. the first n columns of the padded input tensor must contain an n-gram with the first word of the sentence in the middle and `[BOS]` tokens to the left, and
    2. the last n columns of the padded input tensor must contain an n gram with the last word of the longest sentence in the middle (or `[PAD]` tokens for shorter sentences) and `[PAD]` tokens to the right.

- After the input tensor has been padded with the appropriate number of `[BOS]` and `[PAD]` tokens, it needs to be unfolded into a batch of n-grams. The unfolded input will be a 3D tensor of shape $(\text{batch size}, \text{sentence length}, n)$.

The embedding, hidden, and output layers can then be applied to this batch of n-grams. The `nn.Embedding` and `nn.Linear` layers can be applied to 3D inputs just as easily as 2D inputs: both layers only operate on the last dimension of their inputs. Thus, the embedding layer will turn the input into a 4D batch of embeddings of shape (batch size, sentence length, n, embedding size), which you will need to concatenate (i.e., use `torch.cat`) into a 3D array of shape $(\text{batch size}, \text{sentence length}, n \cdot \text{embedding size})$ before applying the hidden and output layers.

Hints for Problems 11 and 12:

1. Please read this tutorial if you confused by objected-oriented programming terms such as "initializers" or "instance variables": `https://www.tutorialspoint.com/python/python_classes_objects.htm`
2. For information about built-in PyTorch modules, please consult the documentation for the `torch.nn` package: `https://pytorch.org/docs/stable/nn.html`
3. Review Problems 1–10.

3.2 Bidirectional RNN

Now, you will implement the bidirectional RNN version of the POS tagging model. Your implementation will support using a simple recurrent network (SRN), long short-term memory network (LSTM), or gated recurrent unit (GRU) as the recurrent cell.

Problem 13. Complete the initializer for the `RNNPosTagger` class. Again, we provide code that constructs the embedding layer and optionally loads pre-trained embeddings. You are responsible for defining the RNN cell and a linear decoder, which are bound to the instance variables `self.rnn` and `self.decoder`, respectively. Your code should satisfy the following requirements.

1. Depending on the specification of the keyword parameter `rnn_type` (which you should allow to be either `"srn"`, `"lstm"`, or `"gru"`), you should construct a module of the appropriate sort and bind it to the variable `self.rnn`. Please use the modules `nn.RNN`, `nn.LSTM`, and `nn.GRU`, and not `nn.RNNCell`, `nn.LSTMCell`, or `nn.GRUCell`. Make sure to use the keyword argument `batch_first=True` when constructing the recurrent unit. And be sure to specify the relevant number of recurrent layers and a specification of whether the network is processing only unidirectionally or bidirectionally.
2. The final linear layer of the network, `self.decoder`, takes the output of the recurrent

network and maps it to a vector of size equal to the number of possible labels (POS tags or the `[PAD]` token). Make sure to think about how the size of this layer will change depending on whether the RNN is unidirectional or bidirectional.

Problem 14. Next, complete the `forward` method for this class. Its input is a tensor representing the current training batch of shape (batch size, sentence length), where each position is a vocabulary index. Your function should return the logits assigned to the possible labels for each input token. Additionally, you should save the hidden state vectors for the top recurrent layer computed after reading each input to the instance variable `self.rnn_hidden_states`. This is because, at the end of the assignment, you will do some analysis on the hidden state vectors generated by the RNN cell.

Hints for Problems 13 and 14:

1. Please study carefully the documentation for `nn.RNN`, `nn.LSTM`, and `nn.GRU`. You may find it helpful to experiment with these modules in the Python console or in a Jupyter Notebook.
2. Review Problems 1–10.

4 Training and Testing

In Assignment 2, our `Layer` base class had an `update` method that applied stochastic gradient descent to the model's parameters. PyTorch does not use this pattern. Instead, it provides *optimizers* through the `torch.optim` package, which implement a wide array of different optimization algorithms. These algorithms include stochastic gradient descent, which we learned about in class, as well as more advanced algorithms that have been empirically shown to work better for training neural networks. In this part of the assignment you will be training your POS tagging models using *Adam*, the most commonly used optimization algorithm for neural networks.

Before attempting the problems in this section, please read the documentation for PyTorch optimizers, available at: https://pytorch.org/docs/stable/optim.html.

Problem 15. Complete the `train_epoch` function in `train.py` script. This function should carry out a single epoch of training and report the results. Please consult the docstring for a description of the function's parameters. Below we provide some details about some of the parameters.

- `train_data` is a `Dataset` object that stores the training data. You can iterate through this data by using the `get_batches` method. In the starter code, this is done as follows:

```
batches = train_data.get_batches(batch_size)
```

```python
for i, (sentences, pos_tags) in enumerate(batches):
    ...
```

Here, `batches` is a Python *iterator*—an object that you can run a for-loop over. When you loop over the output of `Dataset.get_batches`, you get two things: (i) a `LongTensor` of shape (batch size, sentence length) containing a batch of sentences, and (ii) a `LongTensor` of the same shape containing integers representing the POS tag for each word. These are stored in the variables `sentences` and `pos_tags`, respectively.

- In the beginning of the `run_trial` function, you will find the following two lines of code:

```python
loss_function = nn.CrossEntropyLoss(
    ignore_index=pos_tag_pad_index,
    reduction="sum")
optimizer = optim.Adam(model.parameters(), lr=lr)
```

This code creates the objects that are supplied as arguments for the `loss_function` and `optimizer` parameters in `train_epoch`. The `ignore_index=pos_tag_pad_index` keyword argument causes logits with a corresponding label of `[PAD]` to be excluded from the batch loss, while the `reduction="sum"` setting causes the loss function to return the sum of losses incurred in the batch (as opposed to the average loss). The optimizer is an `optim.Adam` object linked to the parameters of `model`. (The optimizer needs to be represented by an object because the Adam algorithm keeps state.)

- You should not use the `pos_tag_pad_index` parameter in the `train_epoch` function. This parameter is used only by the starter code we have already provided.

Problem 16. Complete the `evaluate` function in `train.py`, which should evaluate the performance of your model on a given test, returning the overall POS accuracy and average loss per batch.

As in training, this function should loop through the provided dataset, which may be a dev set or a test set. Since the batch size has no effect on the evaluation results, we use a constant batch size of 100 for simplicity. During each iteration, you should compute the network outputs and total batch loss, saving these values to the variables `logits` and `batch_loss`, respectively. However, here you should not do a backward pass to compute gradients.

Additionally, your code should be able to optionally compute and print a *confusion matrix*, depending on the value of the flag `print_conf_matrix`. A confusion matrix is a matrix where each row and column represents one of the possible POS tags, and the entry in row i

and column j represents the number of tokens with POS tag i to which the model assigned POS tag j. At the bottom of the function, you should see the following code:

```python
if print_conf_matrix:
    print_cm(pred_list, target_list,
             model.pos_tag_vocab.forms[:-1])
```

This code prints a confusion matrix using the function `analysis.print_cm`, which we have provided in the `analysis.py` module. In order to make this code work, please populate the two lists `pred_list` and `target_list` with all the predicted POS tags from the entire test dataset. Be sure not to include any predictions for `[pad]` tokens when you construct these lists. (Note that you do not need to worry about ignoring these symbols for the loss or accuracy computations: this is already taken care of for you.)

This function's parameters are mostly the same as those of `train_epoch`, with the following caveats.

- You should not use the parameters `message` or `pos_tag_pad_index`. These parameters are used only by the starter code we have already provided.
- The flag `print_conf_matrix` specifies whether or not the user would like to print a confusion matrx at the end of the evaluation.

5 Hyperparameter Tuning

Now that you have completed the implementation of the two architectures as well as the code for training and testing, the final step of the development process is to find hyperparameter values that result in the best possible model for each architecture. In the next two problems, you will perform some basic hyperparameter tuning for the POS tagging model. The function `run_trial` in the `train.py` module conducts one *trial* of hyperparameter tuning—its parameters specify a particular hyperparameter configuration, and running the function will train a model using the specified hyperparameter values and report performance on the test set.

We provide two scripts for training a POS tagger: `train_mlp_model.py`, which constructs and trains an MLP model, and `train_rnn_model.py`, which constructs and trains an RNN model. You may use these scripts to train your model, though you are not required to. If you wish, you may choose to carefully study the two scripts and train your model using a custom script or a Jupyter Notebook.

Problem 17. Consider the following hyperparameters for the MLP model:

- the embedding size
- the hidden size (only relevant if the network has at least 1 hidden layer)

- the length of the n-grams used by the network
- the number of hidden layers in the MLP

and the following hyperparameters for Adam:

- the learning rate
- the batch size
- the number of epochs to train the model for.

Find at least 3 combinations of values for the above hyperparameters that yield an accuracy of at least 75% on the ***development*** set. Report the accuracy for each of your 3 (or more) hyperparameter tuning trials on the development set, and indicate which set of hyperparameters yields the best performance. Then, report the accuracy on the ***testing*** set of the model with the best accuracy on the development set.

Problem 18. Follow the same procedure as in the previous problem, but this time for the RNN model. Please tune the following hyperparameters for the RNN model:

- the embedding size
- the hidden size
- the type of RNN cell (SRN, LSTM, or GRU)
- the number of RNN layers
- whether or not the RNN operates bidirectionally

and the following hyperparameters for Adam:

- the learning rate
- the batch size
- the number of epochs to train the model for.

Hints for Problems 17 and 18:

1. Before settling on your 3 hyperparameter configurations, we recommend varying these parameters one at a time and exploring a wide range of values with a short training regimen (one epoch, say) to see which gives the best performance. Once you have found 3 hyperparameter configurations that perform well after a single epoch, you can perform a full hyperparameter tuning trial with a greater number of epochs for each of these configurations.

2. To obtain the best possible performance, please train your network until *convergence*—that is, please use enough epochs so that, at the end of training, the development set accuracy no longer improves from epoch to epoch.

6 Exploring the Trained RNN POS Tagger

In this final part of the assignment, you will explore the behavior and internal representations of the RNN POS tagger you have implemented. If you do not have much in the way of background in natural language syntax, you may find it helpful to review chapters 5 through 7 of Emily Bender's book *Linguistic Fundamentals for Natural Language Processing*.

Problem 19. Train a bidirectional RNN with the set of hyperparameters that gave you the best performance. Please train this network to convergence. Once you have such a network, use the `evaluate` function to compute performance on the test set, with the setting `conf_matrix=True`.

Inspect the confusion matrix for the network. If the network were behaving perfectly, all of the numbers off of the diagonal of the confusion matrix would be 0; this would mean that the network did not assign any POS tags incorrectly. Comment on the nature of the errors that the network is making. Why are some POS tags more confusable for the network than others?

Problem 20. Using the network from the previous question, examine its POS tagging performance on some specific examples, using the `analysis.eval_sent` function. This function will run your model on a single input. Here is an example of how the function may be used in the Python console.

```
>>> from analysis import eval_sent
>>> eval_sent(model, ["the virus disappeared quickly ."])
[['DET', 'NOUN', 'VERB', 'ADV', 'PUNCT']]}
```

For this question, you must explore the network's behavior on at least at the following two kinds of cases (but you are welcome to think of other comparisons that might be interesting): (i) sentences involving a word or words that exhibit POS ambiguity (e.g., *pad*, as in *I wrote on the pad* or *you should pad the matrix*), and (ii) sentences containing novel words that the network has not seen in its training data (e.g., sentences from Lewis Carroll's poem "Jabberwocky"). In both of these cases, the network must go beyond simply memorizing a single POS tag for each word, and must instead use context to help determine it. How well does the network perform in such cases? What kinds of context are sufficient for the network to correctly determine the POS tag for such cases?

Problem 21. For this problem, you should train a network that uses a unidirectional RNN;

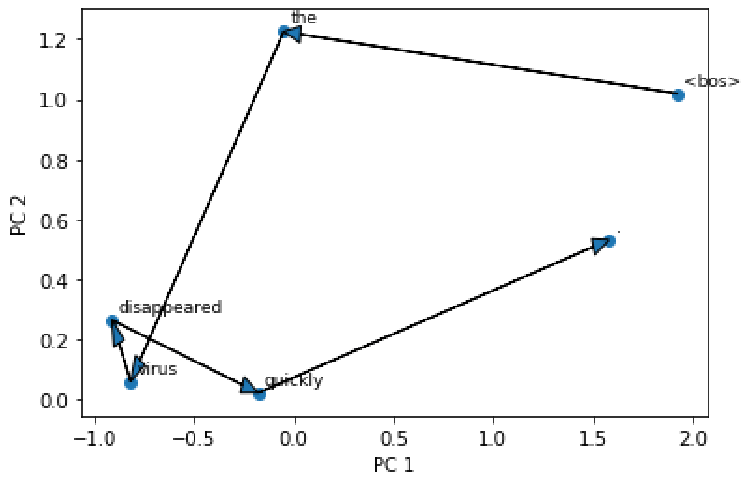
i.e., you should use the `bidirectional=False` flag when you create the `RNNPosTagger` instance. Use the best hyperparameters you can find for a unidirectional RNN POS tagger, and train it to convergence. What you will do next is perform PCA for all of the hidden unit vectors that are computed during the processing of the test set. You can do this using the `make_hidden_pca` function in the `analysis.py` module. This function takes your RNN POS tagger and a dataset (you should use your test data), and returns a PCA model fit to the hidden state vectors generated by the RNN. The function is used as follows.

```
from analysis import make_hidden_pca
pca = make_hidden_pca(model, test_data)
```

After the PCA model has been trained, please use it to plot the trajectories of the network's hidden unit representations for a sentence or list of sentences by using the provided function `plot_sentences` in `analysis.py`. For example, running the following code

```
from analysis import plot_sentences
plot_sentences(model, pca,
               ['The virus disappeared quickly.'])
```

should produce a plot such as the following:



The plot shows each of the hidden state vectors produced by the model, as expressed by their first two principal components.

Use this function to examine the trajectories of phrases of the following sorts:

1. Simple noun phrases (NPs), such as names (*Alice*) or definite and indefinite descriptions (*the/a student*)
2. NPs with adjectival modifiers, such as *the tall student*, *the tall happy student*, etc.

3. NPs with prepositional phrase (PP) modifiers, such as *the student with a dog*, *the student with a dog with a tail*, etc.

4. Sentences with intransitive verbs, such as *Alice is sleeping*, *the student laughed*

5. Sentences with transitive verbs, such as *The dog loves the cat*, *the student wanted an extension*

Try to construct phrases and sentences that involve minimal variation in the choice of words, so that you can see the impact of each factor on the trajectory. So, for example, you might compare the trajectory of a sentence like *the dog is sleeping* with that of *Fido is sleeping* to see how the network's response to descriptions compares to its response to names in the input, and their relative impact on the network's "state" for the remainder of the sentence. You may also identify other grammatically interesting phrase types and explore the network's behavior on them.

Before plotting trajectories, you should confirm that the network correctly assigns the POS to each word in your sentences. Then, use the `plot_sentences` function to create the trajectories, and explore effects of the manipulation you are performing. As was done in Elman's original paper on simple recurrent networks, you may look at trajectories determined by principle components other than the first two, by providing `plot_sentences` with values for its optional parameters `i` and `j`, which are by default set to 0 and 1.

In examining these trajectories, your goal is to get some idea of what the network is representing about the word sequence as it is processing. In your answer, discuss the degree to which it represent word sequences in a way that reflects the regularities of English syntax. For example, does the network treat all NPs the same or does it treat some NPs differently? Are there patterns in terms of which NPs are treated the same? What impact does modification by a PP or an adjective (or a sequence of them) have on the network's state? How does the trajectory of a subject NP different from that of an object NP?

7 Submission Instructions

To submit your completed assignment, please upload the following files to CodePost. Please ensure that your files have the same filenames as indicated below. **Failure to submit your assignment correctly will result in a deduction of 5 points.**

- Your modified versions of the following Python files: `model.py` and `train.py`
- A Markdown document called `assignment3.md` containing your solutions to Problems 1–10 (PyTorch exercises), 17–18 (hyperparameter tuning results), and 19–21 (analysis questions)