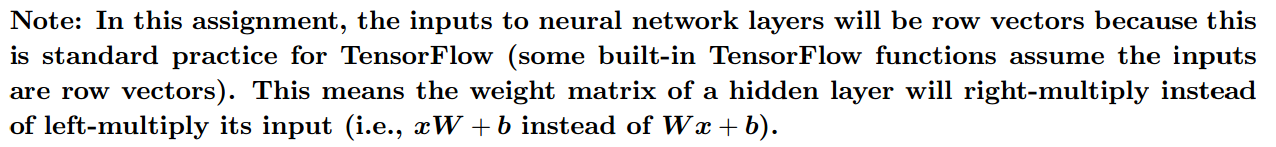
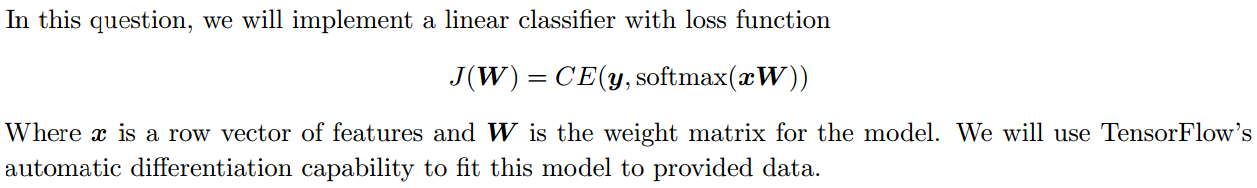
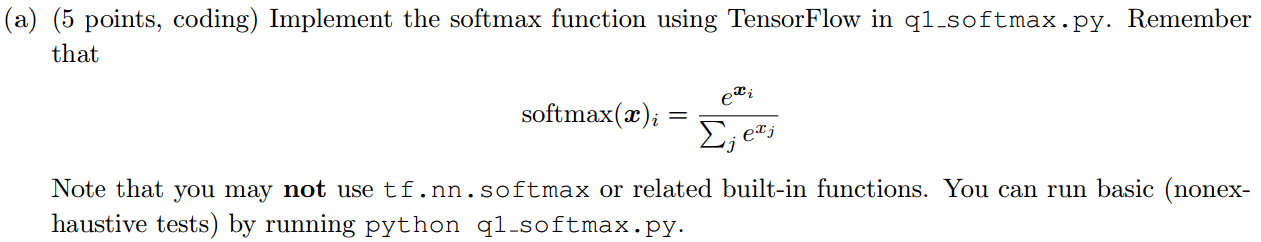
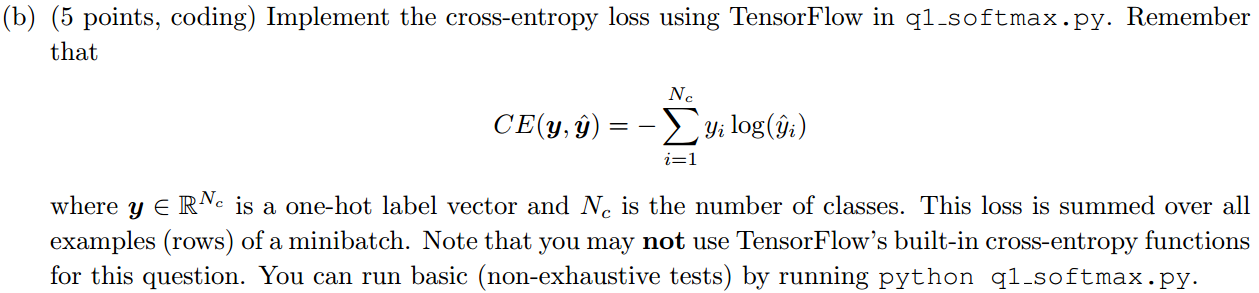
CS 224n: Assignment #2

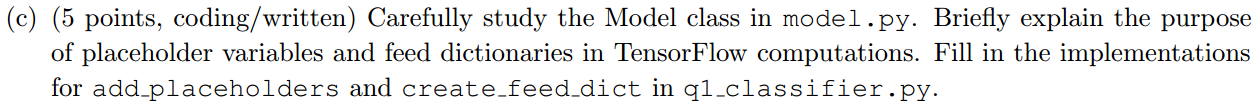


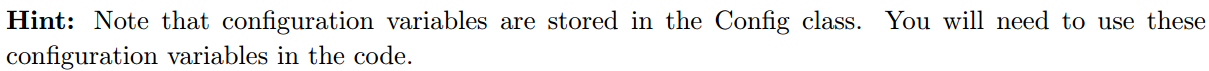
## 1 Tensorflow Softmax (25 points)

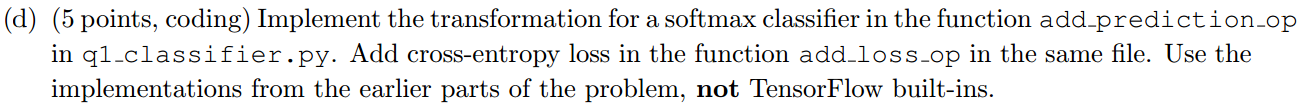


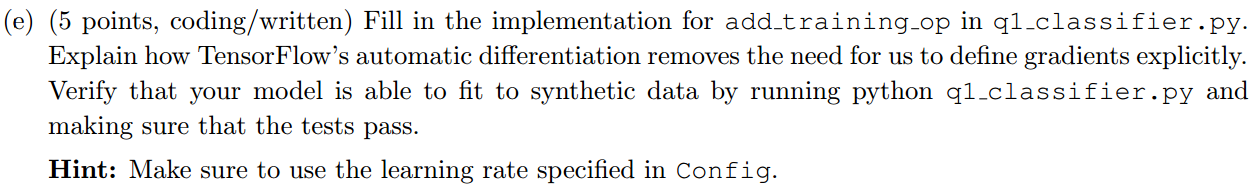












## 2 Neural Transition-Based Dependency Parsing (50 points)

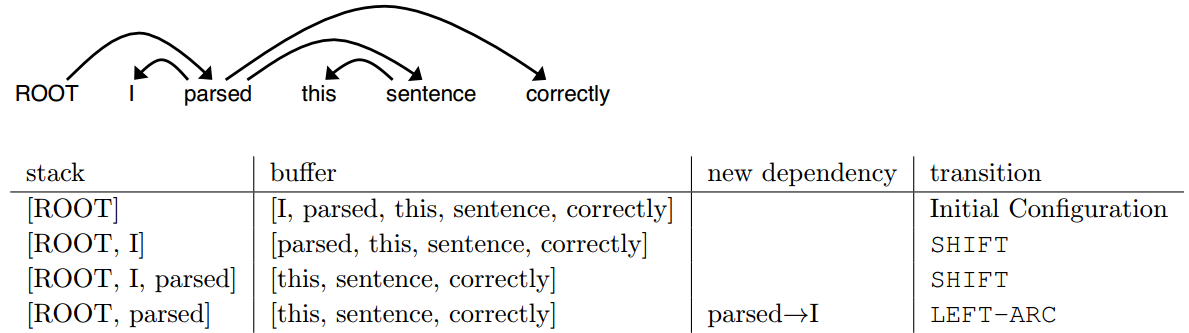
In this section, you’ll be implementing a neural-network based dependency parser. A dependency parser analyzes the grammatical structure of a sentence, establishing relationships between “head” words and words which modify those heads. Your implementation will be a transition-based parser, which incrementally builds up a parse one step at a time. At every step it maintains a partial parse, which is represented as follows:

*•* A *stack* of words that are currently being processed.  
*•* A *buffer* of words yet to be processed.  
*•* A list of *dependencies* predicted by the parser.

Initially, the stack only contains ROOT, the dependencies lists is empty, and the buffer contains all words  
of the sentence in order. At each step, the parse applies a *transition* to the partial parse until its buffer is  
empty and the stack is of size 1. The following transitions can be applied:  
*•* SHIFT: removes the first word from the buffer and pushes it onto the stack.  
*•* LEFT-ARC: marks the second (second most recently added) item on the stack as a dependent of the  
first item and removes the second item from the stack.  
*•* RIGHT-ARC: marks the first (most recently added) item on the stack as a dependent of the second  
item and removes the first item from the stack.

Your parser will decide among transitions at each state using a neural network classifier. First, you will  
implement the partial parse representation and transition functions.

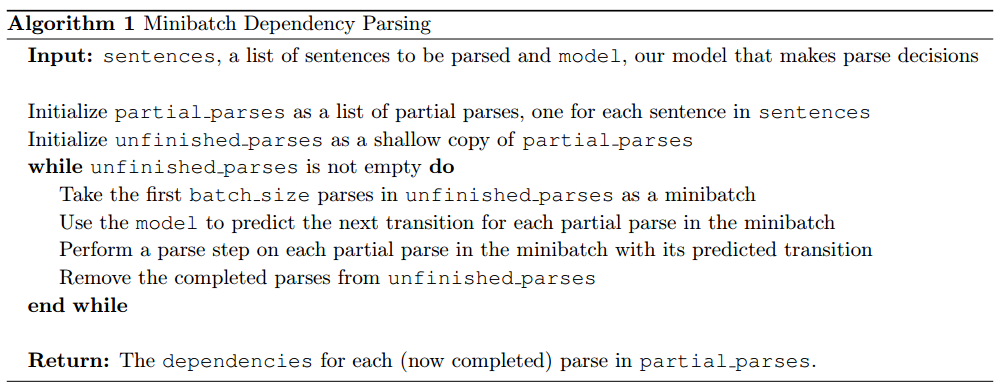
(a) (6 points, written) Go through the sequence of transitions needed for parsing the sentence “*I parsed  
this sentence correctly”*. The dependency tree for the sentence is shown below. At each step, give the  
configuration of the stack and buffer, as well as what transition was applied this step and what new  
dependency was added (if any). The first three steps are provided below as an example.



(b) (2 points, written) A sentence containing *n* words will be parsed in how many steps (in terms of *n*)?  
Briefly explain why.

(c) (6 points, coding) Implement the \_\_init\_\_ and parse step functions in the PartialParse class in  
q2 parser transitions.py. This implements the transition mechanics your parser will use. You  
can run basic (not-exhaustive) tests by running python q2 parser transitions.py.

(d) (6 points, coding) Our network will predict which transition should be applied next to a partial parse.  
We could use it to parse a single sentence by applying predicted transitions until the parse is complete.  
However, neural networks run much more efficiently when making predictions about batches of data at  
a time (i.e., predicting the next transition for a many different partial parses simultaneously). We can  
parse sentences in mini-batches with the following algorithm.



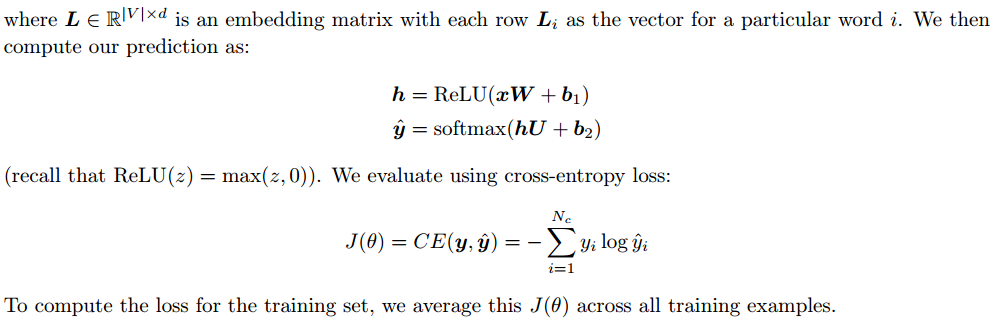
Implement this algorithm in the mini-batch parse function in q2 parser transitions.py. You can run basic (not-exhaustive) tests by running python q2 parser transitions.py.  
*Note: You will need mini-batch parse to be correctly implemented to evaluate the model you will build  
in part (h). However, you do not need it to train the model, so you should be able to complete most of  
part (h) even if mini-batch parse is not implemented yet.*

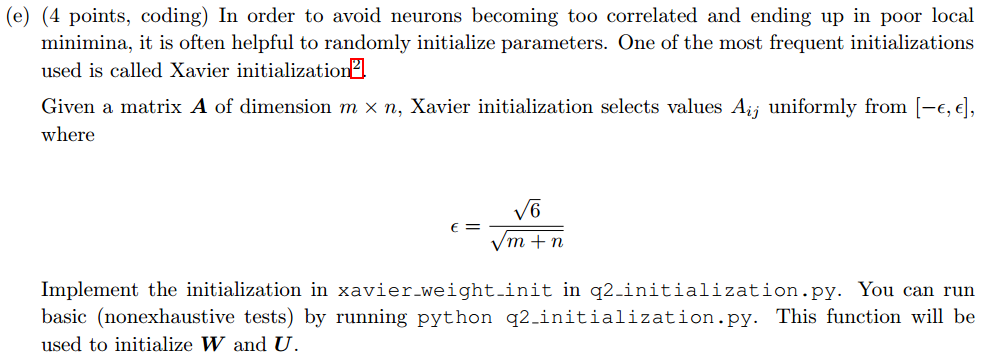
We are now going to train a neural network to predict, given the state of the stack, buffer, and dependencies, which transition should be applied next. First, the model extracts a feature vector representing the current state. We will be using the feature set presented in the original neural dependency parsing paper: *A Fast and Accurate Dependency Parser using Neural Networks*1. The function extracting these features has been implemented for you in parser utils. This feature vector consists of a list of tokens (e.g., the last word in the stack, first word in the buffer, dependent of the second-to-last word in the stack if there is one, etc.). They can be represented as a list of integers

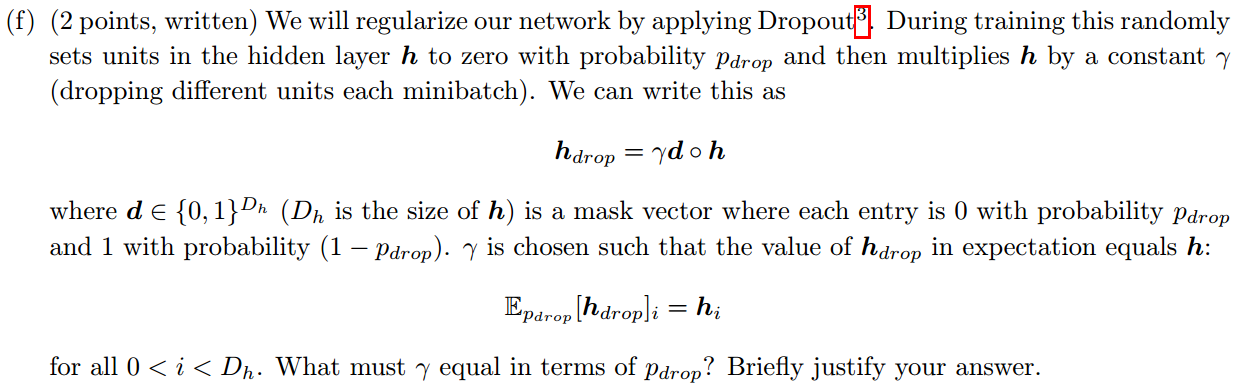


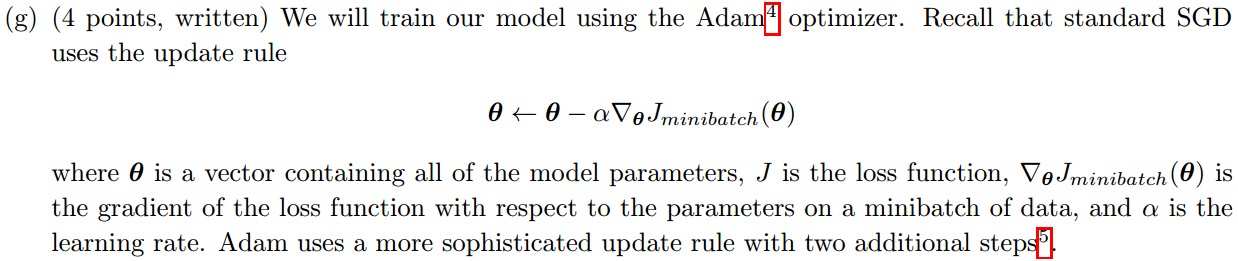
where *m* is the number of features and each is the index of a token in the vocabulary ( is the vocabulary size). First our network looks up an embedding for each word and concatenates them into a single input vector:

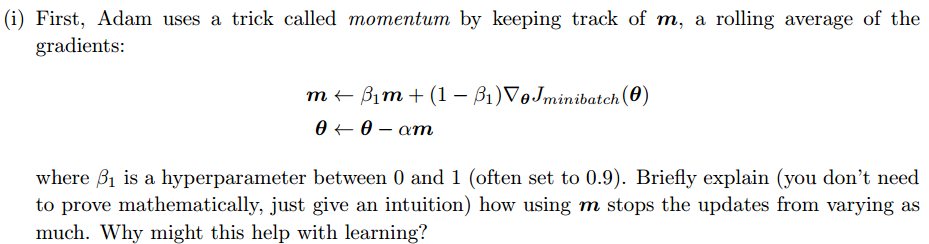


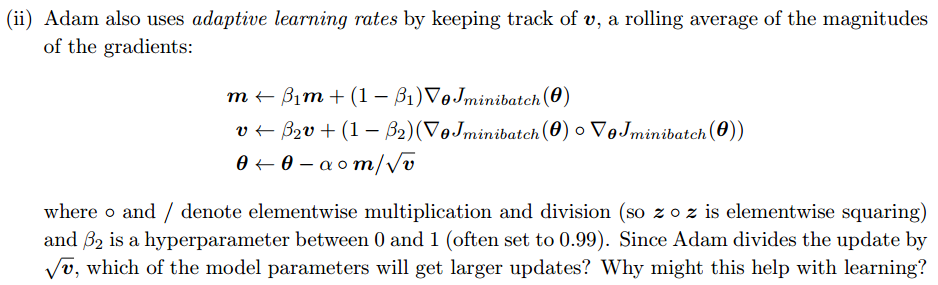


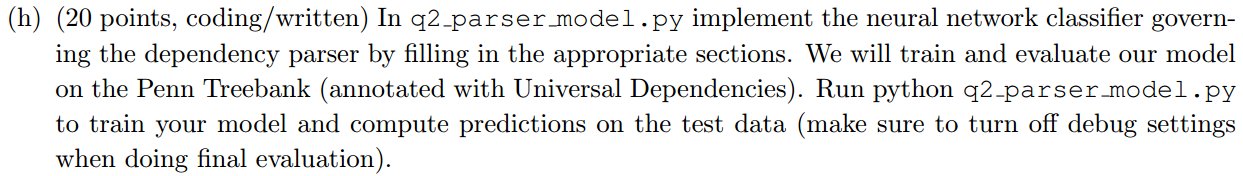


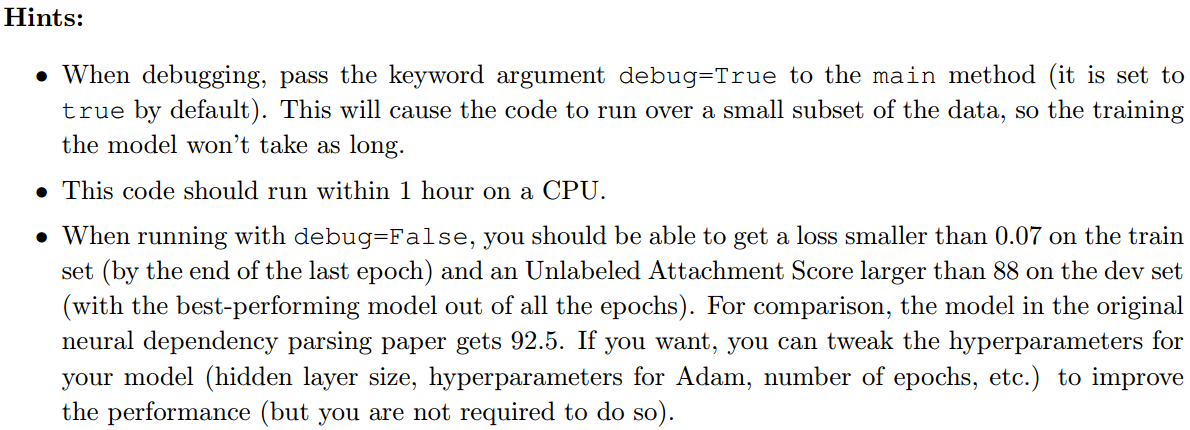


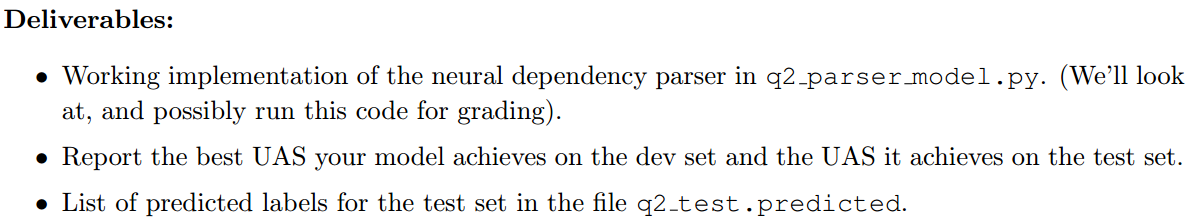


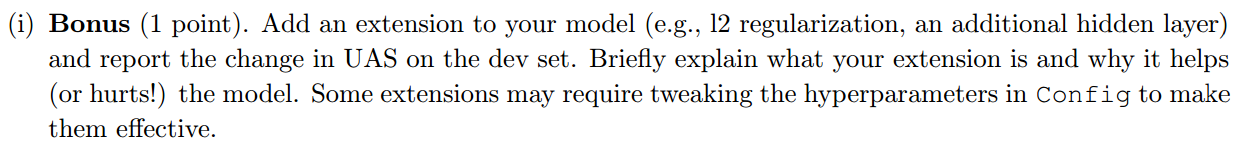












## 3 Recurrent Neural Networks: Language Modeling (25 points)

In this section, you’ll compute the gradients of a recurrent neural network (RNN) for language modeling.

