Sequence Analysis

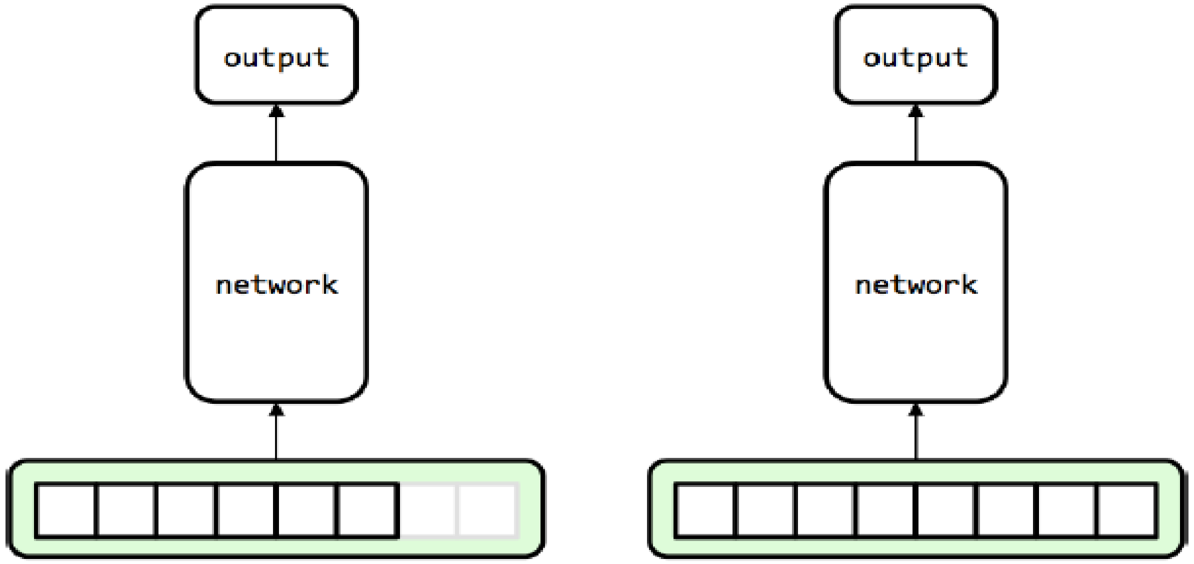
Dr. Sanjay Chatterji

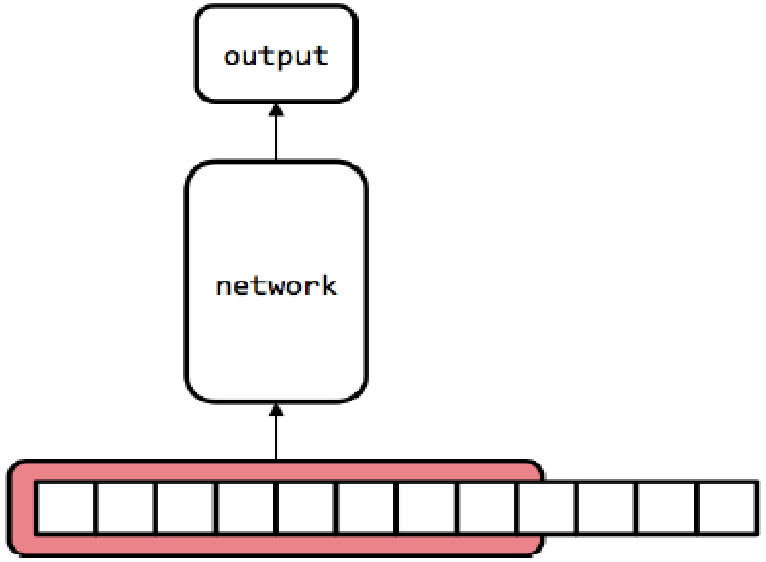
Learning Lower-Dimensional Representation

❑We've only worked with data with fixed sizes: images from MNIST, CIFAR-10, and ImageNet.

❑Variable Length Inputs (sequences): whether it's reading the morning newspaper, making a bowl of cereal, listening to the radio, watching a presentation, or deciding to execute a trade on the stock market.

❑We'll have to be a little bit more clever about how we approach designing deep learning models for Sequences.





Feed-forward networks thrive on fixed input size problems.

Feed-forward neural networks for analyzing sequences

- Feed-forward neural networks break when analyzing sequences.
- If the sequence is the same size as the input layer, the model can perform as we expect it to.
- It’s even possible to deal with smaller inputs by padding zeros to the end of the input until it’s the appropriate length.
- However, if the input exceeds the size of the input layer, naively using the feedforward network no longer works.

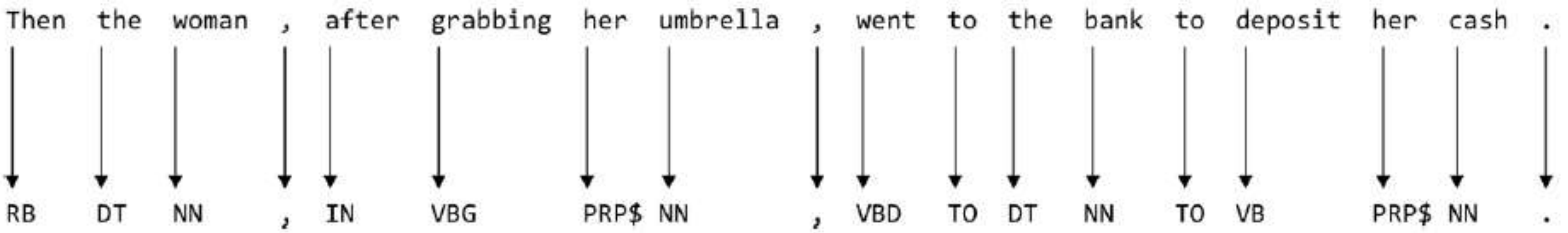
All hope is lost !!!!!   ....   NO!!!!

- In the following sections:
- We’ll explore strategies to “hack” feedfoward networks to handle sequences.
- We’ll analyze the limitations of these hacks and discuss new architectures to address them.
- We’ll discuss some of the most advanced architectures explored to date to tackle some of the most difficult challenges in replicating human-level logical reasoning and cognition over sequences.

Tackling seq2seq with Neural N-Grams

- We’ll explore a feed-forward neural network architecture that can process a body of text and produce a sequence of part-of-speech (POS) tags.
- It’s a solid first step toward developing an algorithm that can understand the meaning of how words are used in a sentence.
- This problem is also interesting because it is an instance of a class of problems known as seq2seq.
- Other famous seq2seq problems include translating text between languages, text summarization, and transcribing speech to text.

An example of an accurate POS parse of an English sentence

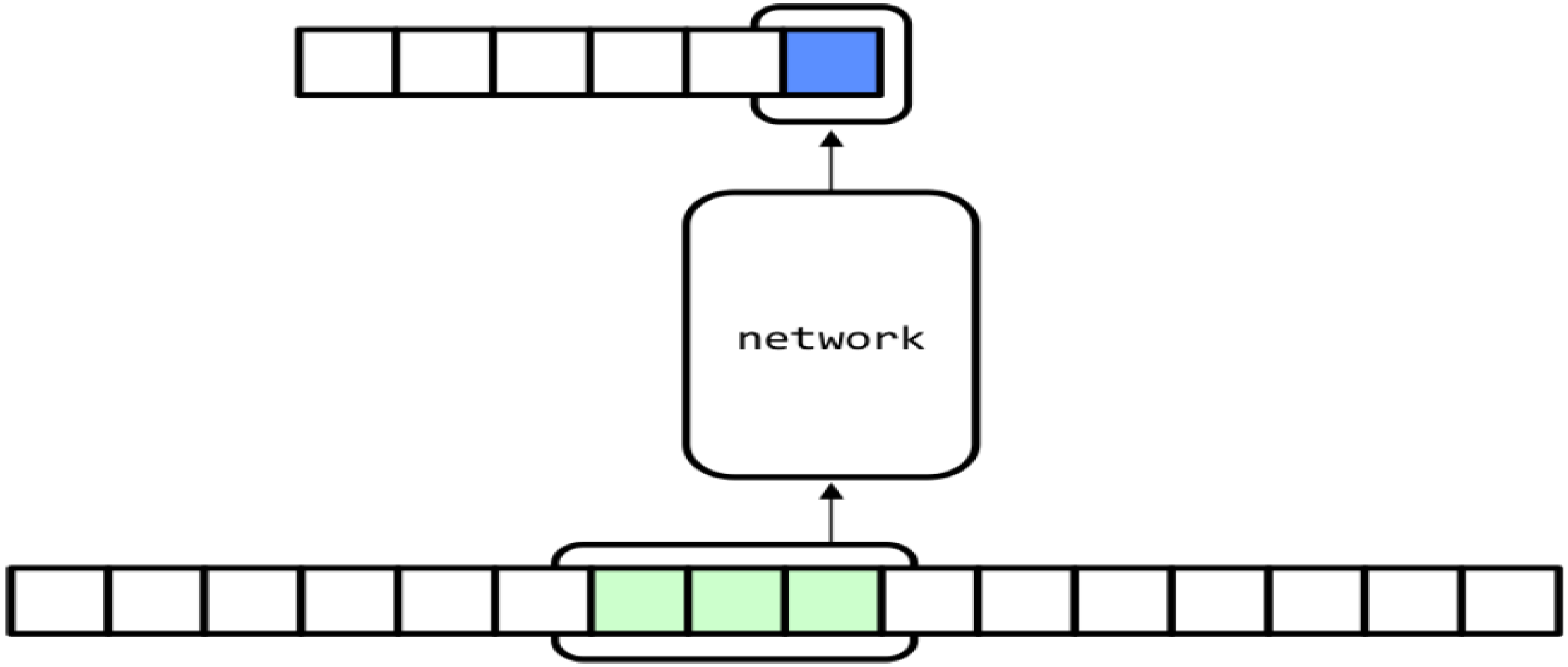


Then	the	woman	,	after	grabbing	her	umbrella	,	went	to	the	bank	to	deposit	her	cash	.
RB	DT	NN	,	IN	VBG	PRP$	NN	,	VBD	TO	DT	NN	TO	VB	PRP$	NN	.

Using feed-forward network to perform seq2seq ignoring long term dependencies

- It’s not obvious how we might take a body of text all at once to predict the full sequence of POS tags.
- Instead, we leverage a trick that is akin to the way we developed distributed vector representations of words.
- The key observation is this: it is not necessary to take into account long-term dependencies to predict the POS of any given word.
- Here, instead of using the whole sequence to predict all of the POS tags simultaneously, we can predict each POS tag one at a time by using a fixed-length subsequence.
- We utilize the subsequence starting from the word of interest and extending n words into the past.

Using feed-forward network to perform seq2seq ignoring long term dependencies



Context Window

- When we predict the POS tag for the i^{th} word in the input, we utilize the $i-n+1^{st}$, $i-n+2^{nd}$, . . . , i^{th} words as the input.
- We’ll refer to this subsequence as the context window.
- In order to process the entire text, we’ll start by positioning the network at the very beginning of the text.
- We’ll then proceed to move the network’s context window one word at a time, predicting the POS tag of the rightmost word, until we reach the end of the input.
- Leveraging the word embedding strategy, we’ll also use condensed representations of the words instead of one-hot vectors.
  - This will allow us to reduce the number of parameters in our model and make learning faster.

Implementing a Part-of-Speech Tagger

- On a high level, the network consists of an input layer that leverages a 3-gram context window.
- We’ll utilize word embeddings that are 300- dimensional, resulting in a context window of size 900.
- The feed-forward network will have two hidden layers of size 512 neurons and 256 neurons, respectively.
- Finally, the output layer will be a softmax calculating the probability distribution of the POS tag output over a space of 44 possible tags.
- As usual, we’ll use the Adam optimizer with default hyperparameter settings.
  - train for a total of 1,000 epochs.
  - leverage batch-normalization for regularization.

Word2vec using Gensim

- We'll leverage pretrained word embeddings generated from Google News.
- It includes vectors for 3 million words and phrases and was trained on roughly 100 billion words.
- We can use the gensim Python package to read the dataset.
  - ➢from gensim.models import Word2Vec
  - ➢model = Word2Vec.load_word2vec_format('/path/to/googlenews.bin', binary=True)
- To avoid loading the full dataset into memory every single time we run our program, especially while debugging code or experimenting with different hyperparameters, we cache the relevant subset of the vectors to disk using a lightweight database known as LevelDB.

CoNLL-2000 POS dataset

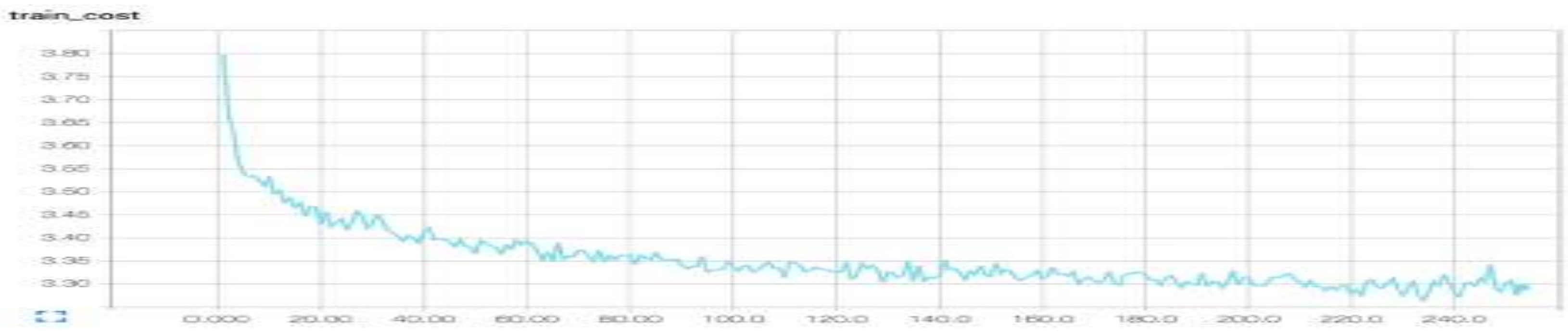
- The gensim model contains three million words, which is larger than our dataset.
- For the sake of efficiency, we’ll selectively cache word vectors for words in our dataset and discard everything else.
- To figure out which words we’d like to cache, we download the POS dataset from the CoNLL-2000 task.
  - It consists of sequence of rows of word and corresponding part of speech.
- To match the formatting of the dataset to the gensim model, we’ll have to do some preprocessing.
  - replace digits with '#' characters
  - combine separate words into entities where appropriate
  - utilize underscores where the raw data uses dashes

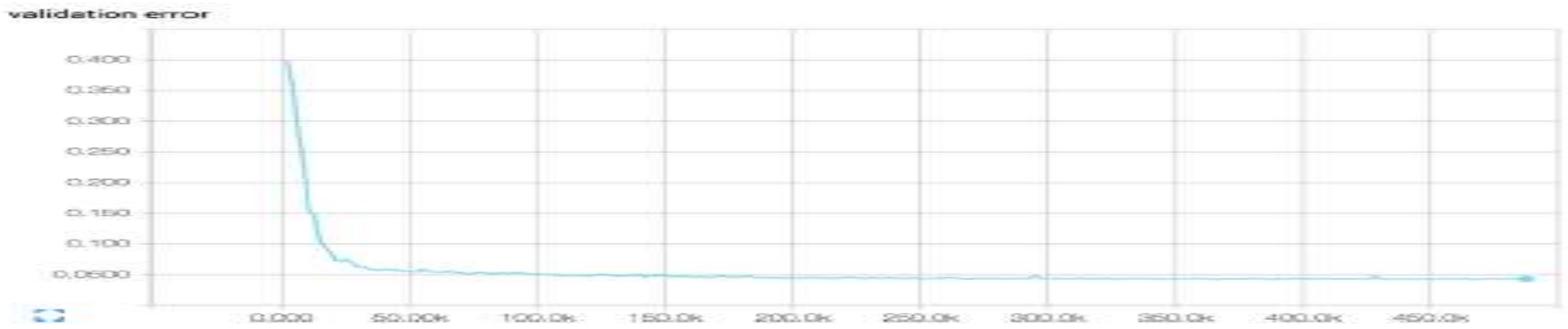
Loading words in LevelDB & Building dataset object

- Now that we’ve appropriately processed the datasets for use:
- We can load the words in LevelDB.
- If the word or phrase is present in the gensim model, we can cache that in the LevelDB instance.
- If not, we randomly select a vector to represent the token, and cache it so that we remember to use the same vector in case we encounter it again.
- Finally, we build dataset objects for both training and test datasets, which we can utilize to generate minibatches for training and testing purposes.
- Building the dataset object requires access to the LevelDB db, the dataset, a dictionary tags_to_index that maps POS tags to indices in the output vector, and a boolean flag get_all that determines whether getting the minibatch should retrieve the full set by default.

Designing the feed-forward network for POS tagging

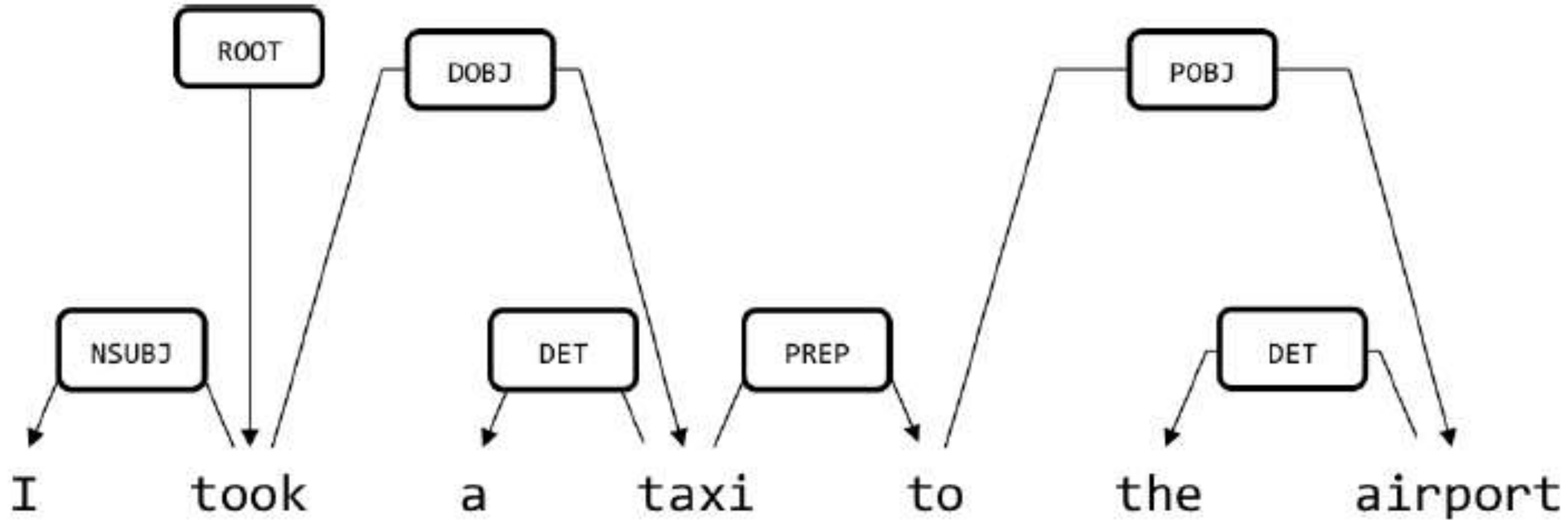
- Finally, we design our feed-forward network similarly to our approaches in previous chapters.
- In every epoch, we manually inspect the model by parsing the sentence.
- Within 100 epochs of training, the algorithm achieves over 96% accuracy and nearly perfectly parses the validation sentence.





Next Move for Parsing

- The POS tagging model was a great exercise, but it was mostly rinsing and repeating concepts we’ve learned in previous chapters.
- Now, we’ll start to think about much more complicated sequence-related learning tasks.
- To tackle these more difficult problems, we’ll need to broach brand-new concepts, develop new architectures, and start to explore the cutting edge of modern deep learning research.
- We’ll start by tackling the problem of dependency parsing next.



An example of a dependency parse, which generates a tree of relationships between words in a sentence

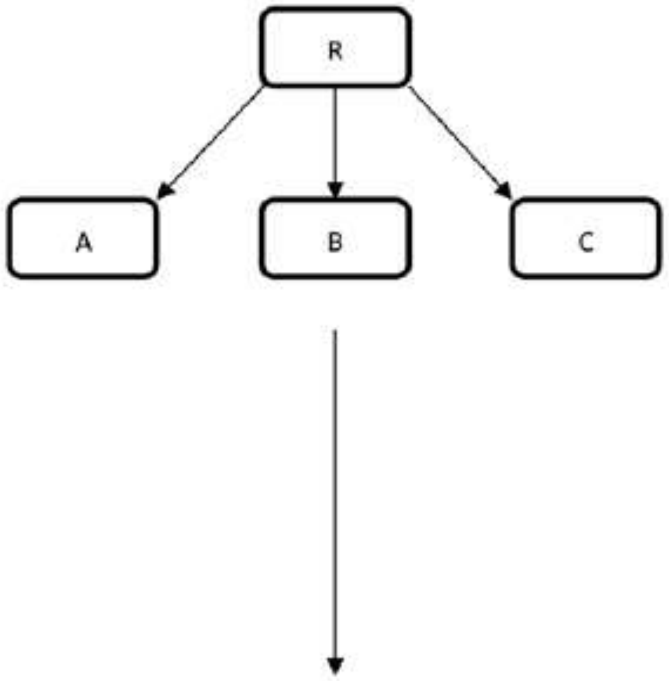
Tackling Complex Problem

- The framework we used to solve the POS tagging task was simple.
- Sometimes we need to be much more creative about how we tackle seq2seq problems, especially as the complexity of the problem increases.
- Now we’ll explore strategies that employ creative data structures to tackle difficult seq2seq problems like Dependency Parsing.
- The idea behind building a dependency parse tree is to map the relationships between words in a sentence.
- The words “I” and “taxi” are children of the word “took,” specifically as the subject and direct object of the verb, respectively.

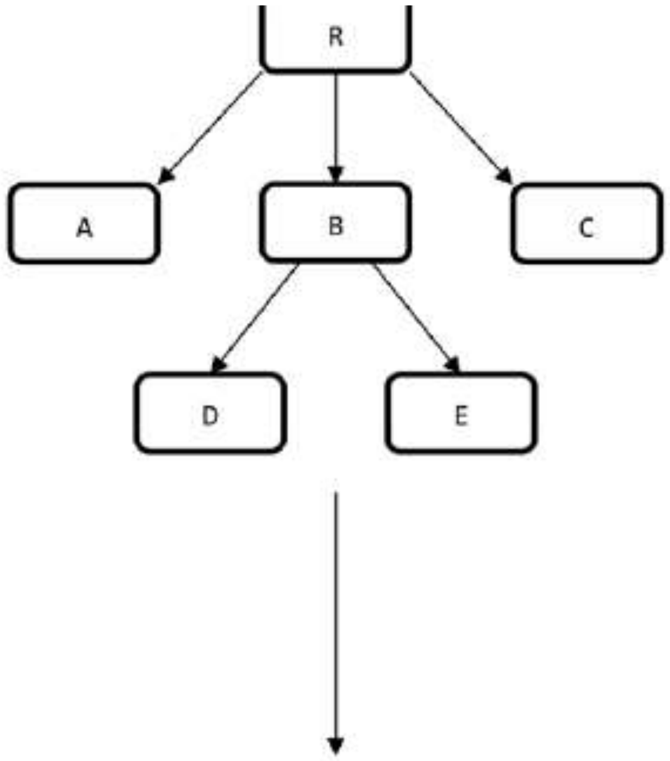
Linearizing the Tree

- One way to express a tree as a sequence is by linearizing it.
- If you have a graph with a root R, and children A (connected by edge r_a), B (connected by edge r_b), and C (connected by edge r_c), we can linearize the representation as (R,r_a,A,r_b,B,r_c,C).
- Suppose node B actually has two more children; D (connected by edge b_d) and E (connected by edge b_e). We can represent this new graph as (R, r_a, A, r_b, [B, b_d, D, b_e, E], r_c, C).
- Using this paradigm, we can take our example dependency parse and linearize it.

Linearize two example trees, the diagrams omit edge labels for the sake of visual clarity

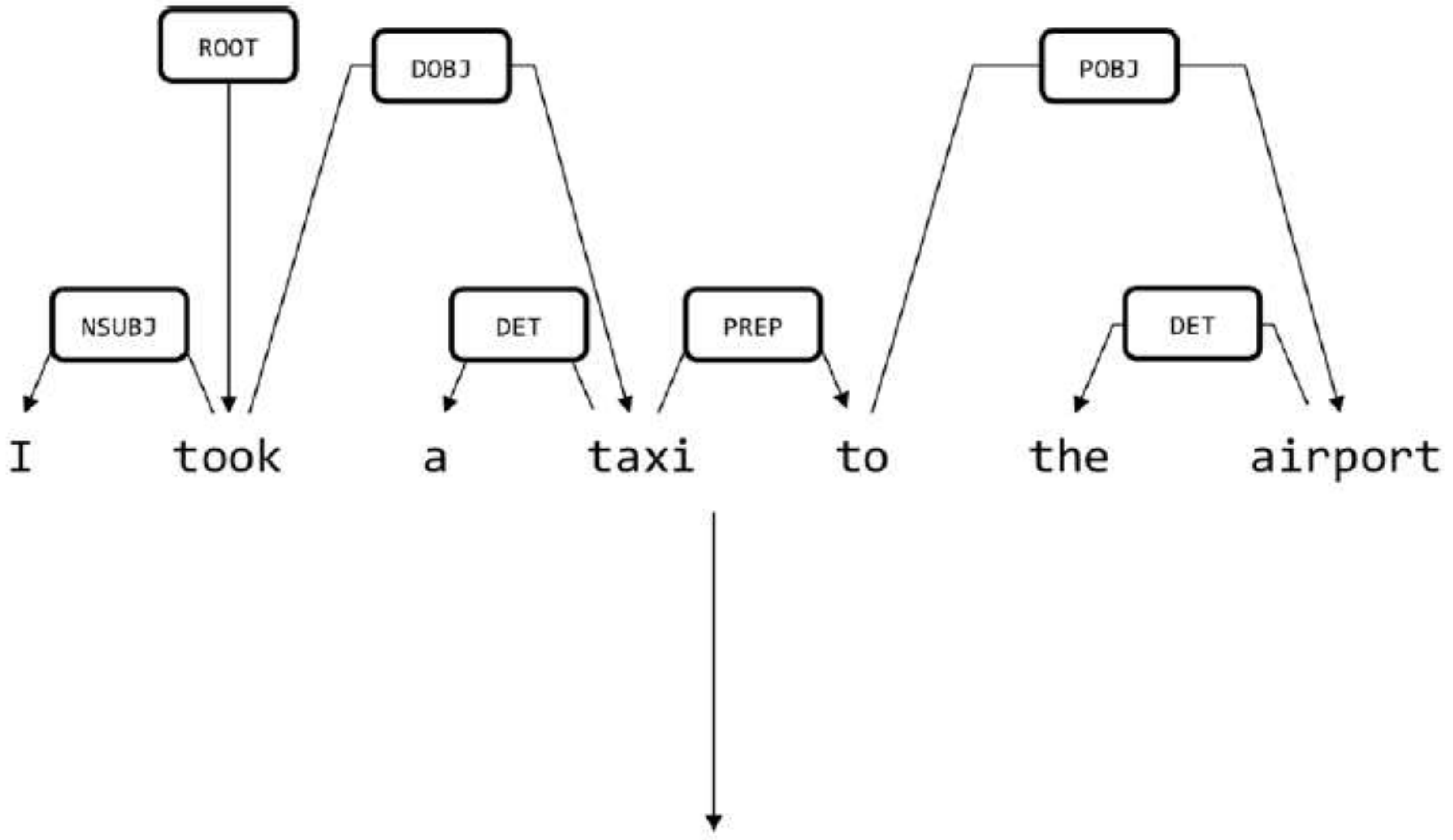


(R, r_a, A, r_b, B, r_c, C)



(R, r_a, A, r_b, (B, b_d, D, b_e, E), r_c, C)

Linearization of our example dependency parse tree



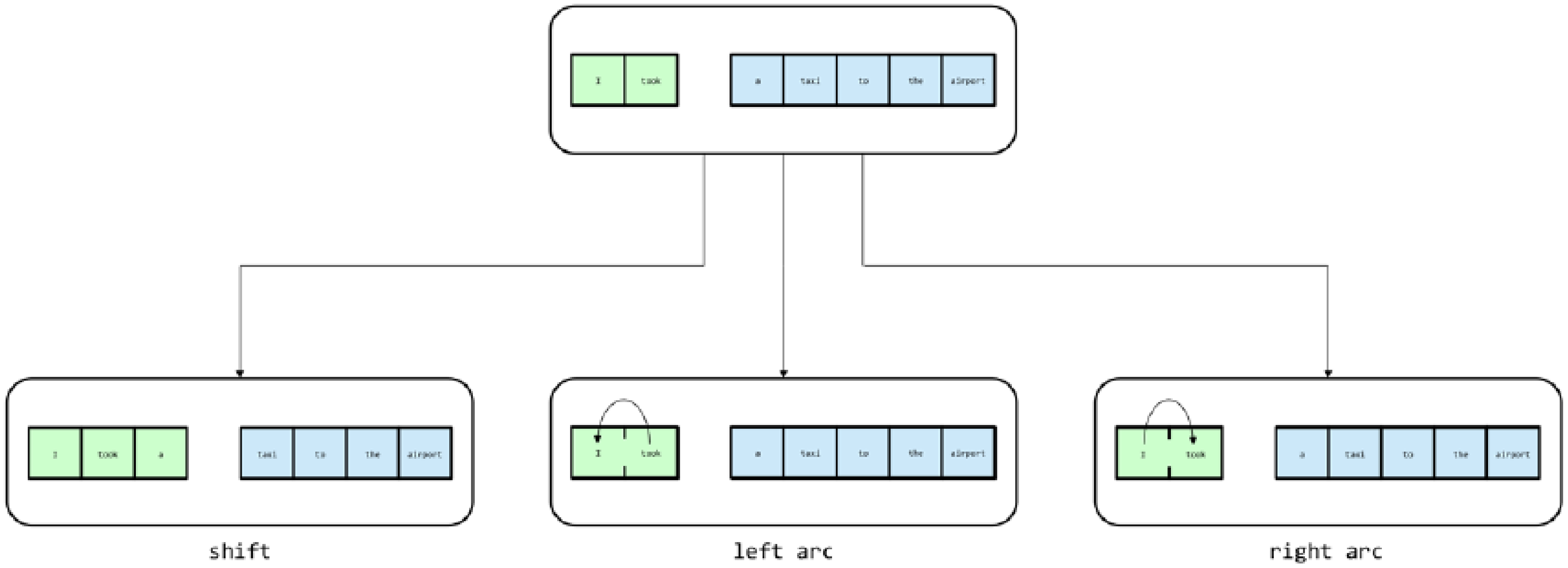
```
(took, NSUBJ, I, DOBJ, (taxi, DET, a, PREP, (to, POBJ, (airport, DET, the))))
```

One interpretation of this seq2seq problem

- Read the input sentence and produce a sequence of tokens that represents the linearization of the input's dependency parse.
- How to port our strategy where there was a clear one-to-one mapping between words and their POS tags?
- We easily make decisions on a POS tag by looking at nearby context.
- Here, there's no clear relationship between how words are ordered in the sentence and how tokens in the linearization are ordered.
- It also seems like dependency parsing takes us with identifying edges that may span a significantly large number of words.
- Here, it seems like this setup directly violates our assumption that we need not take into account any long-term dependencies.

Arc-standard System

- To make the problem more approachable, we instead reconsider the dependency parsing task as finding a sequence of valid “actions” that generates the correct dependency parse.
- This technique, known as the **arc-standard system**, was first described by Nivre in 2004 and later leveraged in a neural context by Chen and Manning in 2014.
- In the arc-standard system, we start by putting the first two words of the sentence in the stack and maintaining the remaining words in the buffer.



At any step, we have three options:

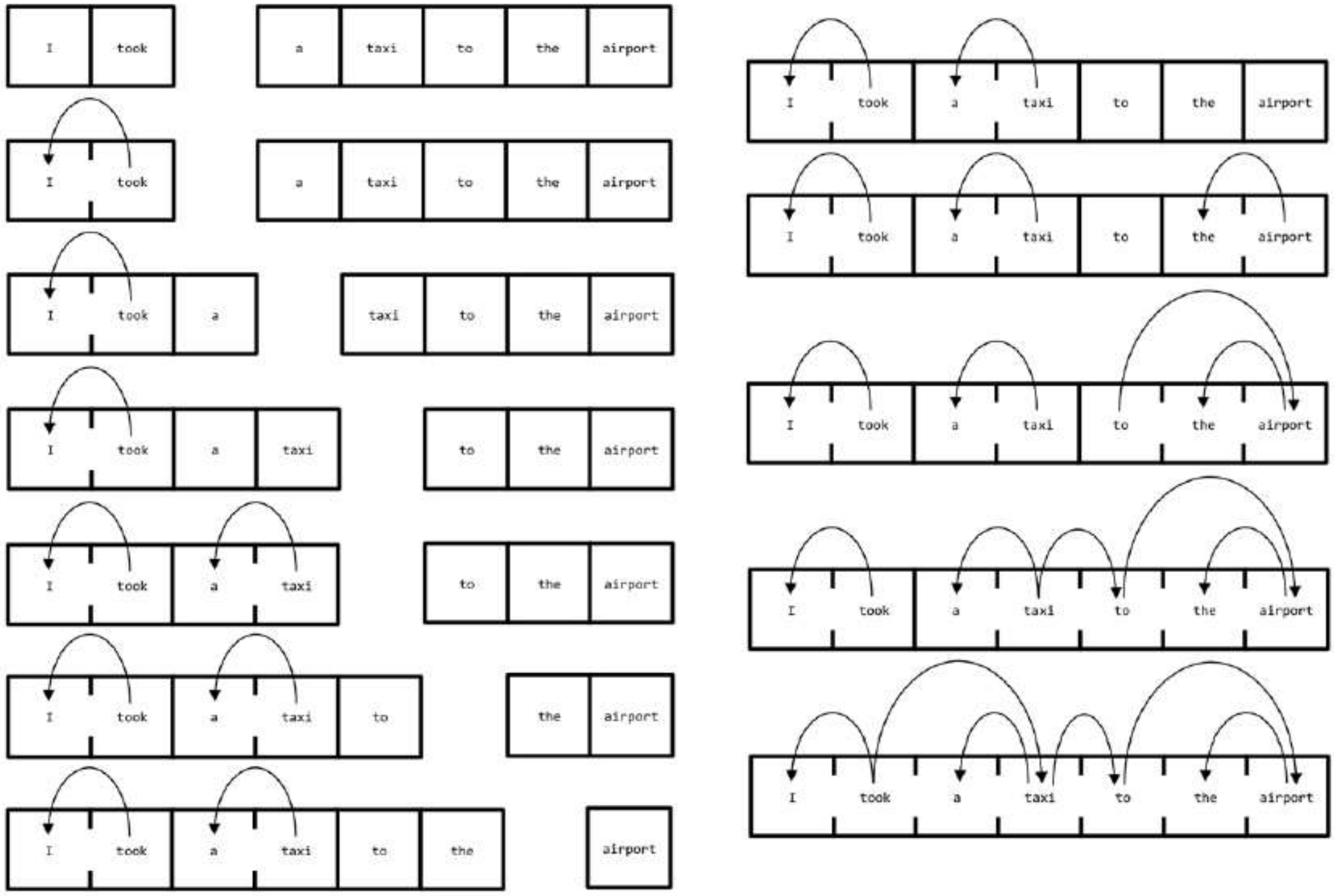
1. to shift a word from the buffer (blue) to the stack (green)
2. to draw an arc from the right element to the left element (left arc)
3. to draw an arc from the left element to the right element (right arc)

Steps

- At any step, we can take one of three possible classes of actions:
- SHIFT
    - Move a word from the buffer to the front of the stack.
- LEFT ARC
    - Combine the two elements at the front of the stack into a single unit where the root of the right element is the parent node and the root of left element is the child node.
- RIGHT ARC
    - Combine the two elements at the front of the stack into a single unit where the root of the left element is the parent node and the root of right element is the child node.

The Entire Process

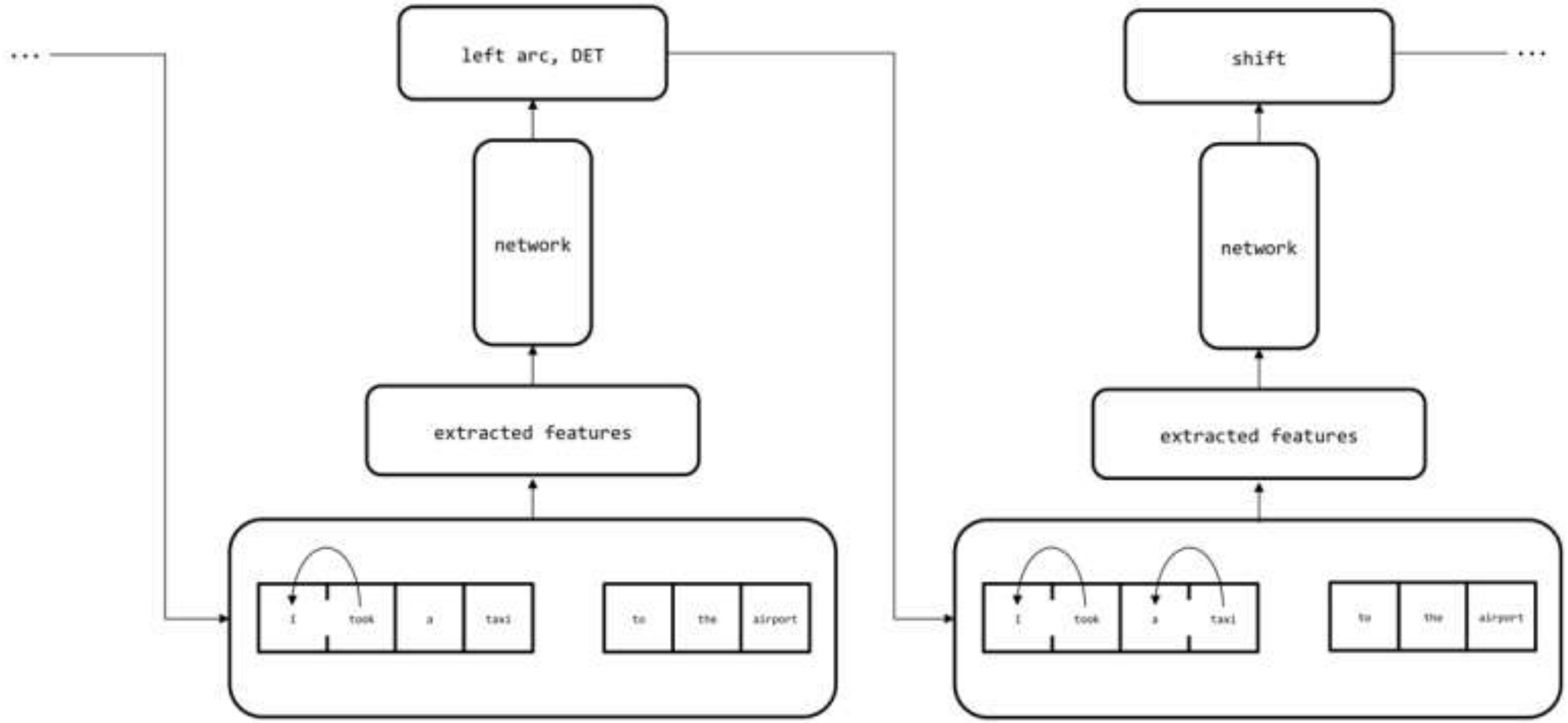
- While there is only one way to perform a SHIFT, the ARC actions can be of many flavors, each differentiated by the dependency label assigned to the arc that is generated.
- We’ll simplify our discussions and illustrations by considering each decision as a choice among three actions.
- We finally terminate this process when the buffer is empty and the stack has one element in it.
- We illustrate a sequence of actions that generates the dependency parse for our example input sentence.


I
took
a
taxi
to
the
airport

Reformulating the decision-making framework as a learning problem

- At every step:
  - We take the current configuration.
  - We vectorize the configuration by extracting a large number of features that describe the configuration.
    - words in specific locations of the stack/buffer
    - specific children of the words in these locations
    - part of speech tags
    - etc.
  - We can feed this vector into a feedforward network.
  - Then we compare its prediction of the next action to take to a gold standard decision made by a human linguist.

A neural framework for arc-standard dependency parsing



Idea in Google’s SyntaxNet

- To use this model in the wild
  - We can take the action/s that the network recommends
  - We apply it to the configuration, and
  - We use this new configuration as the starting point for the next step
    - feature extraction
    - action prediction, and
    - action application
- These ideas form the core for Google’s SyntaxNet, the state-of-the-art open source implementation for dependency parsing.
- Delving into the nitty-gritty aspects of implementation is beyond the scope of this class
  - Refer the implementation of Parsey McParseface, currently the most accurate publicly reported English language parser.

Issues of naive strategy for deploying SyntaxNet in practice

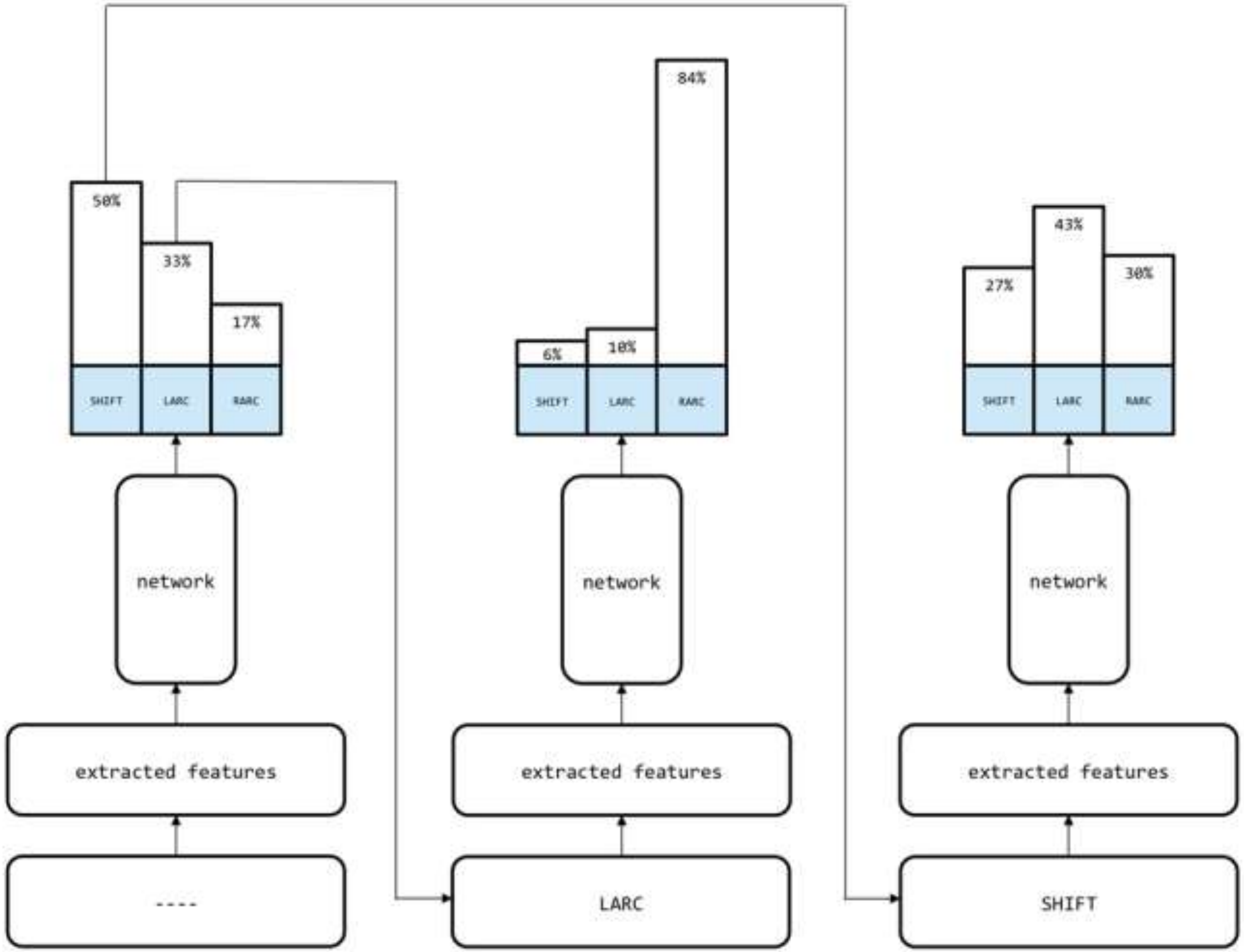
- The strategy was purely greedy; i.e., we selected prediction with the highest probability without being concerned that we might potentially paint ourselves into a corner by making an early mistake.
- In the POS example, making an incorrect prediction was inconsequential.
- This is because each prediction could be considered a purely independent subproblem.
- In SyntaxNet our prediction at step n affects the input we use at step n + 1.
- This implies that any mistake we make will influence all later decisions.
- Moreover, there's no good way of "going backward" and fixing mistakes when they become apparent.

A Garden path sentence

- Garden path sentences are an extreme case of where this is important.
- Consider the following sentence: “The complex houses married and single soldiers and their families.”
- “complex” is a noun (as in a military complex) and that “houses” is a verb.
- A greedy version of SyntaxNet would fail to correct the early parse mistake of considering “complex” as an adjective describing the “houses,” and therefore fail on the full version of the sentence.

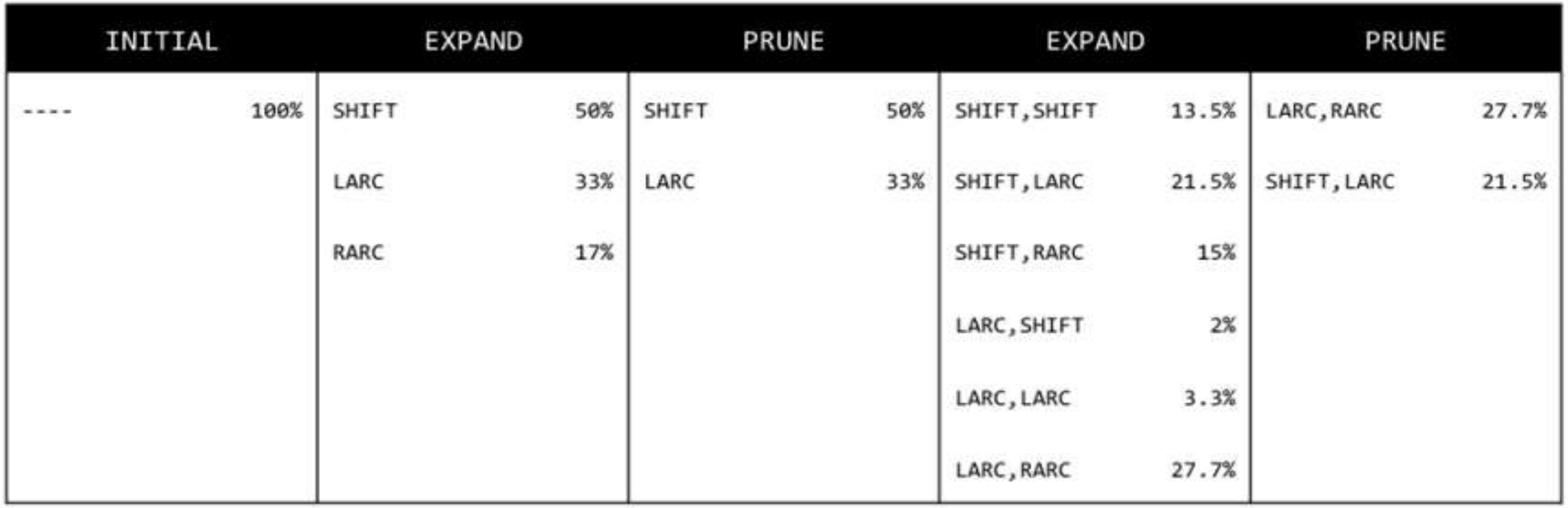
Beam Search and Global Normalization

- To remedy this shortcoming, we utilize a strategy known as beam search.
- We generally leverage beam searches in situations like SyntaxNet, where the output of our network at a particular step influences the inputs used in future steps.
- Basic idea behind beam search
  - instead of greedily selecting the most probable prediction at each step, we maintain a beam of the most likely hypotheses (up to a fixed beam size b) for the sequence of the first k actions and their associated probabilities.


50%
33%
17%
SHIFT
LARC
RARC
network
extracted features

6%
10%
84%
SHIFT
LARC
RARC
network
extracted features
LARC
27%
43%
30%
SHIFT
LARC
RARC
network
extracted features
SHIFT

Two major phases of Beam Search: expansion and pruning



INITIAL		EXPAND		PRUNE		EXPAND		PRUNE	
----	100%	SHIFT	50%	SHIFT	50%	SHIFT,SHIFT	13.5%	LARC,RARC	27.7%
		LARC	33%	LARC	33%	SHIFT,LARC	21.5%	SHIFT,LARC	21.5%
		RARC	17%			SHIFT,RARC	15%		
						LARC,SHIFT	2%		
						LARC,LARC	3.3%		
						LARC,RARC	27.7%		

Expansion step

- We take each hypothesis and consider it as a possible input to SyntaxNet.
- Assume SyntaxNet produces a probability distribution over a space of |A| total actions.
- We then compute the probability of each of the b|A| possible hypotheses for the sequence of the first $k + 1$ actions.

Pruning step

- We keep only the b hypothesis out of the b|A| total options with the largest probabilities.
- Beam searching enables SyntaxNet to correct incorrect predictions post facto by entertaining less probable hypotheses early that might turn out to be more fruitful later in the sentence.
- Greedy approach would suggest the correct sequence of moves as a SHIFT followed by a LEFT ARC. In reality, the best (highest probability) option would a LEFT ARC followed by a RIGHT ARC.
- Beam searching with beam size 2 surfaces this result.

local normalization

- The full open source version takes this a full step further and attempts to bring the concept of beam searching to the process of training the network.
- In a locally normalized network, our network is tasked with selecting the best action given a configuration.
- The network outputs a score that is normalized using softmax layer.
- This is meant to model a probability distribution over all possible actions, provided the actions performed thus far.
- Our loss function attempts to force the probability distribution to the ideal output (i.e., probability 1 for the correct action and 0 for all other actions).
- The cross-entropy loss does a spectacular job of ensuring this for us.

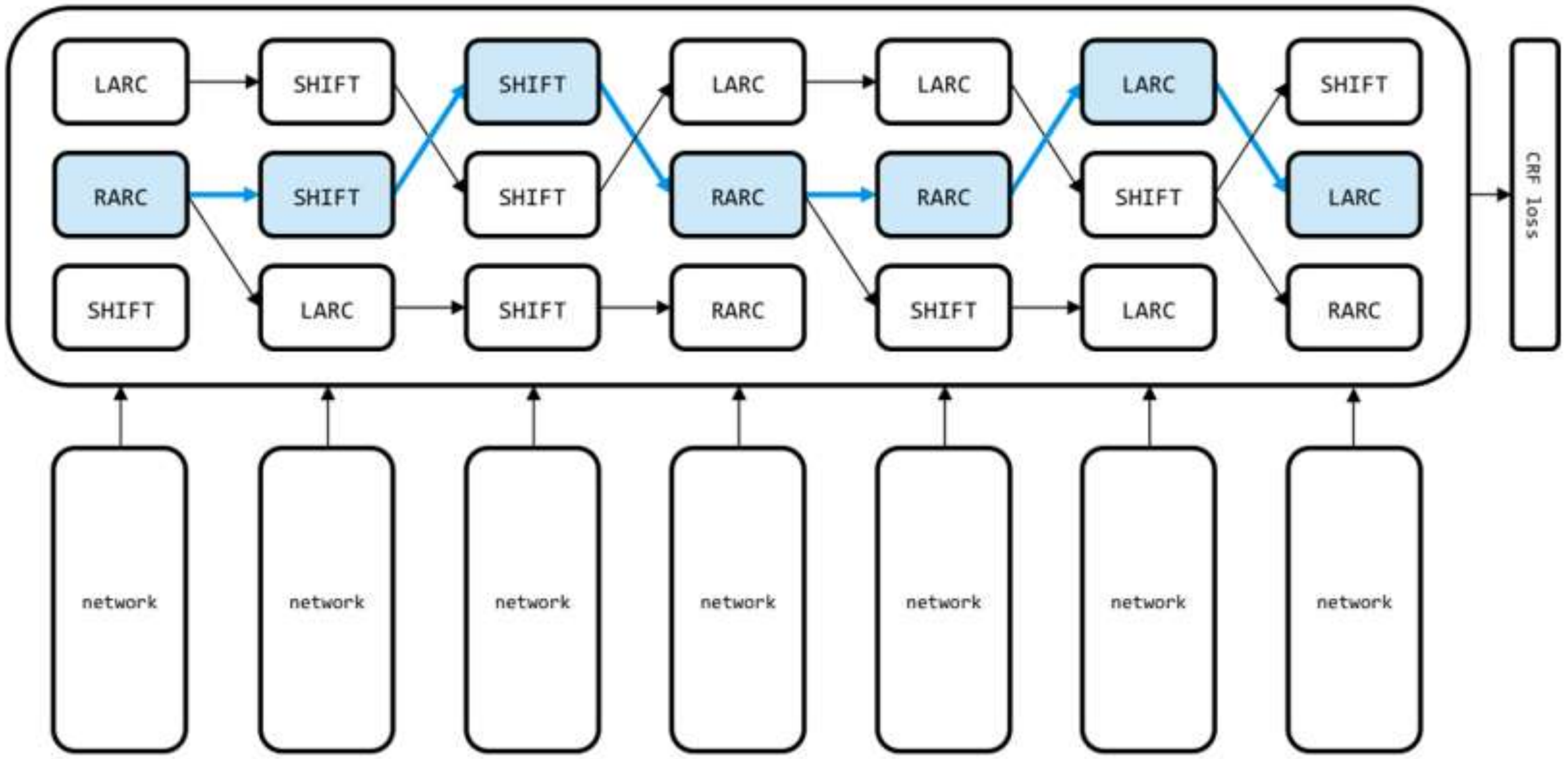
global normalization

- In a globally normalized network, our interpretation of the scores is slightly different.
- Instead of putting the scores through a softmax to generate a per-action probability distribution, we add up all the scores for a hypothesis action sequence.
- One way of ensuring that we select the correct hypothesis sequence is by computing sum over all possible hypotheses and then applying softmax layer to generate probability distribution.
- We could theoretically use the same cross-entropy loss function as we used in the locally normalized network.

Problem of the Global Optimization Strategy

- There is an intractably large number of possible hypothesis sequences.
- Considering average sentence length of 10 and conservative total number of 15 possible actions—1 shift and 7 labels for each of the left and right arcs—this corresponds to 1,000,000,000,000,000 possible hypotheses.
- To make this problem tractable, we apply beam search with fixed beam size until we
    1) reach the end of the sentence, or
    2) the correct sequence of actions is no longer contained on the beam.
- We then construct a loss function that tries to push the “gold standard” action sequence as high as possible on the beam by maximizing its score relative to the other hypotheses.

We can make global normalization in SyntaxNet tractable by coupling training and beam search



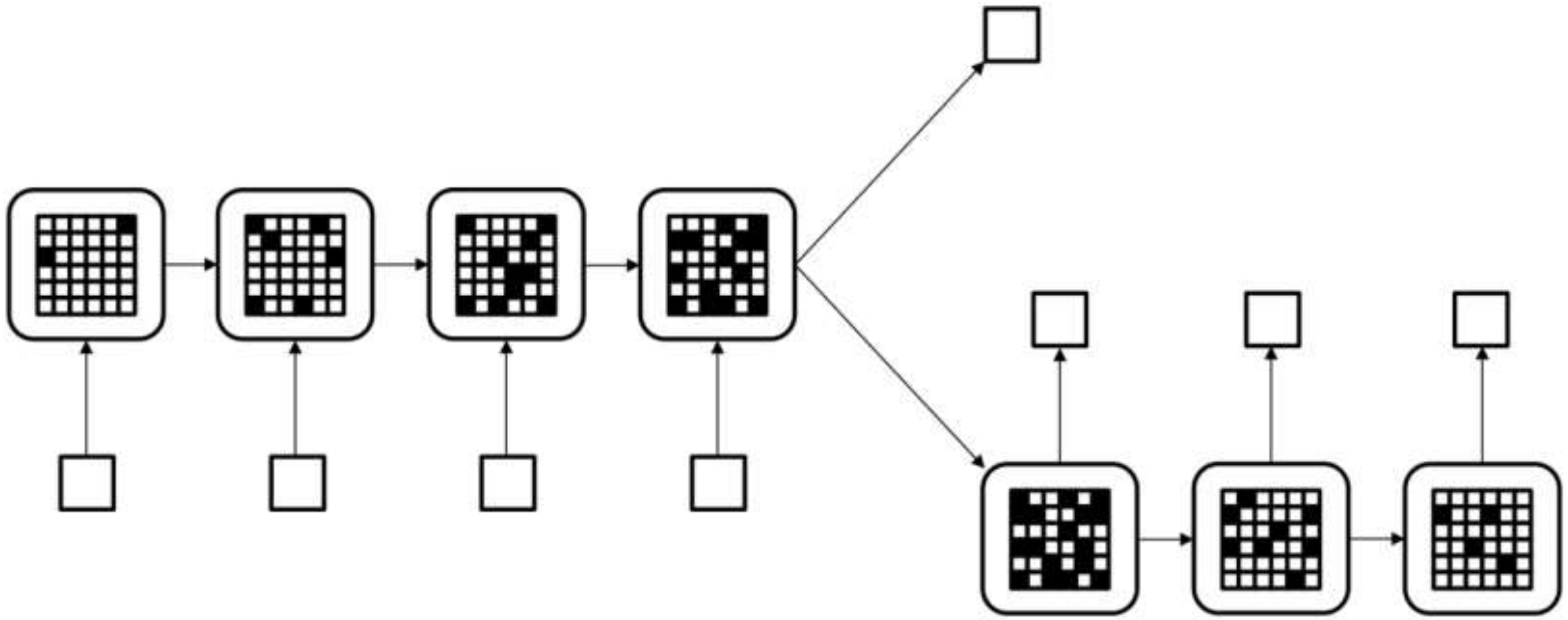
A Case for Stateful Deep Learning Models

- While we’ve explored several tricks to adapt feed-forward networks to sequence analysis, we’ve yet to truly find an elegant solution to sequence analysis.
- In the POS tagger example, we made the explicit assumption that we can ignore long-term dependencies.
- We were able to overcome some of the limitations of this assumption by introducing the concepts of beam searching and global normalization.
- Even still, the problem space was constrained to situations in which there was a one-to-one mapping between elements in the input sequence to elements in the output sequence.
    - For example, even in the dependency parsing model, we had to reformulate the problem to discover a one-to-one mapping between a sequence of input configurations while constructing the parse tree and arc-standard actions.

Cases having no one to one mapping

- Sometimes, the task is far more complicated than finding a one-to-one mapping between input and output sequences.
- We might want to develop a model that can consume an entire input sequence at once and then conclude if the sentiment of the entire input was positive or negative.
- We may want an algorithm that consumes a complex input (such as an image) and generate a sentence, describing the input.
- We may even want to translate sentences from one language to another (e.g., from English to French).
- In all of these instances, there’s no obvious mapping between input tokens and output tokens.

The ideal model for sequence analysis can store information in memory over long periods of time, leading to a coherent "thought" vector that it can use to generate an answer



The idea is simple

- We want our model to maintain some sort of memory over the span of reading the input sequence.
- As it reads the input, the model should able to modify this memory bank, taking into account the information that it observes.
- By the time it has reached the end of the input sequence, the internal memory contains a “thought” that represents the key pieces of information, that is, the meaning, of the original input.
- Then we should be able to use this thought vector to either produce a label for the original sequence or produce an appropriate output sequence (translation, description, abstractive summary, etc.).

The concept isn’t explored in previous chapters

- Feed-forward networks are inherently “stateless.”
- After it’s been trained, the feedforward network is a static structure.
- It isn’t able to maintain memories between inputs, or change how it processes an input based on inputs it has seen in the past.
- To execute this strategy, we’ll need to reconsider how we construct neural networks to create deep learning models that are “stateful.”
- To do this, we’ll have to return to how we think about networks on an individual neuron level.
- In the next class, we’ll explore how *recurrent connections* (as opposed to the feed-forward connections) enable models to maintain state.

Thank You