Last class...

We built a model to predict the part of speech (POS) for a given word in a sentence.

This is a big shift in the direction of this class - before we were previously only building models to classify and cluster entire pieces of text (eg. Does this synopsis look more like a Drama or a Comedy? Does this review look positive or negative?)

Now we are focusing on models that label at the word level - a “many to many” assignment where we have an input of a word sequence and output a tag sequence.

Let’s recap the model that we built for POS tagging.

The point of our model was to tag words by part of speech for each word -

A simple example is **[merger, proposed] -> [NOUN, VERB]**

Thus we will perform a multiclass prediction for each word - out the possible 10 classes of words, we make a guess for each one.

<u>Our inputs are twofold:</u>

1. A sequence of words **(t_1 through t_n)**
2. A sequence of the previous POS tags **(t_1 through t_{n-1})**

This allows us to make a prediction on all possible information - the words but also all the previous POS tags, which may affect what the current POS tag is

For each word in the sentence, we will predict the tag for the word based on:

1. all the preceding words in the sentence, including the current word
2. all the previous tags in the sentence

For a given position *idx*, the input is words[*idx*] and tags[*idx-1*].

An additional point:

We pad our inputs (with zeros in this case) on the left side of the input. This accomplishes a couple things:

1. It provides a starting point for training and predictions. Since our inputs require tags along with words, we need a dummy tag to allow for our first prediction in a sentence to made.
2. Padding is still a necessary step for neural networks to interpret sequences - as explained by Jason Brownlee:

   “Deep learning libraries assume a vectorized representation of your data. In the case of variable length sequence prediction problems, this requires that your data be transformed such that each sequence has the same length. This vectorization allows code to efficiently perform the matrix operations in batch for your chosen deep learning algorithms.”

An additional point:

- Another way to think about the need for padding is the idea that everything we do in a neural network is essentially matrix multiplication.
- We need fixed inputs to be able to build a weight matrix that will be appropriate to multiply by our inputs, i.e. our weight matrix can’t change in shape dynamically.

The model:

1.The model takes in both our tag inputs and word inputs in terms of indices (each word corresponds to a particular index)

2a. Each input is then embedded into a dense vector representation
2b. The inputs are then concatenated together

3. We pass this input through an RNN layer - the RNN layer will output a sequence of vectors instead of the most recent hidden state

- This is a necessary step to perform a **many-to-many prediction**, rather than the many to one calculations that we have previously been making

4. Time Distributed Dense - we pass the output of the RNN through a dense layer with a softmax function. The “time distributed” nature of this layer only means that it will be able to **accept a sequence of vectors** and **create a sequence of outputs.**

Next steps:

Our POS tagging model performs at a pretty decent accuracy level but our approach can be improved by changing the objective function.

Right now our objective function is a softmax function - This method makes local choices. In other words, even if we capture some information from the context in our sentence from the LSTM, the tagging decision is still local.

We don’t make use of the neighboring tagging decisions. For instance, in New York, the fact that we are tagging York as a location should help us to decide that New corresponds to the beginning of a location.

Next steps:

In the next slides, we'll introduce a new model task, called Named Entity Recognition or NER.

Using this task, we will figure out two more ways to incorporate context into our predictions.

Named Entity Recognition

(some material adapted from Guillaume Genthial)

```
John  lives in New   York  and works for the European Union
B-PER O     O  B-LOC I-LOC O   O     O   O   B-ORG    I-ORG
```

In the CoNLL2003 task, the entities are `LOC, PER, ORG and MISC` for *locations, persons, orgnizations* and *miscellaneous*. The no-entity tag is `O`. Because some entities (like `New York`) have multiple words, we use a *tagging scheme* to distinguish between the beginning (tag `B-...`), or the inside of an entity (tag `I-...`). Other tagging schemes exist (IOBES, etc). However, if we just pause for a sec and think about it in an abstract manner, we just need a system that assigns a class (a number corresponding to a tag) to each word in a sentence.

"But wait, why is it a problem? Just keep a list of locations, common names and organizations!"

- This problem is non-trivial because a lot of entities – such as names or organizations, have no real prior knowledge

- Thus, we need to extract contextual information from the sentence

For our implementation, we are assuming that the data is stored in a `.txt` file with one word and its entity per line, like the following example

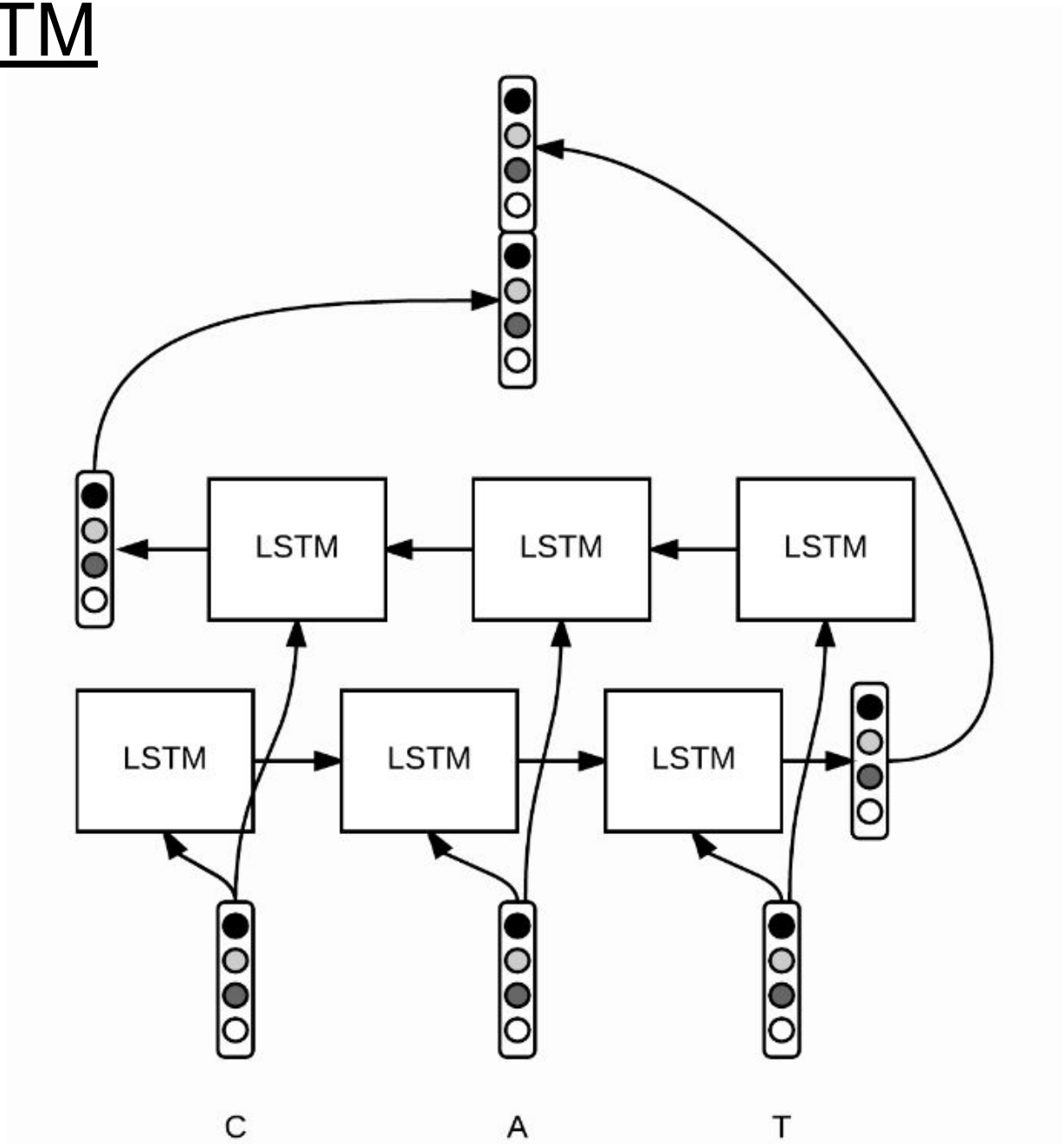
```
EU B-ORG
rejects O
German B-MISC
call O
to O
boycott O
British B-MISC
lamb O
. O

Peter B-PER
Blackburn I-PER
```

Model

Three parts:

- Word representation (use pretrained word embeddings)
- Contextual word representation – Bidirectional LSTM
- Decoding – transferring our coded word representation to a classification of whether or not its an entity

Bidirectional LSTM



Bidirectional LSTM

The Bidirectional LSTM should have some advantages because it is one way to incorporate more context into our predictions.

Instead of encoding our input in one sequence direction, we encode our input in two directions so that the encoding has a richer amount of information.

Again, we will use the hidden states at each time step and not just the final state

We had as input a sequence of m word vectors $w_1,\ldots,w_m \in R_n$ and now we have a sequence of vectors $h_1,\ldots,h_m \in R_k$. Whereas the w_t only captured information at the word level (syntax and semantics), the h_t also take context into account.

Decoding

We have two options to make our final prediction.

In both cases, we want to be able to compute the probability $P(y_1,\ldots,y_m)$ *of a tagging sequence* y_t *and find the sequence with the highest probability. Here,* y_t *is the id of the tag for the t-th word.*

Here we have two options:

Option 1:

softmax: normalize the scores into a vector $p \in \mathbb{R}^9$ such that $p[i] = \frac{e^{s[i]}}{\sum_{j=1}^{9} e^{s[j]}}$. Then, p_i can be interpreted as the probability that the word belongs to class i (positive, sum to 1). Eventually, the probability $\mathbb{P}(y)$ of a sequence of tag y is the product $\prod_{t=1}^{m} p_t[y_t]$.

Decoding

Option 2: Conditional Random Field (CRF)

- By modeling the conditional probability of a sequence of tags given the input, denoted **p(tag sequence|phrase)** or **p(y|x)**
- Imagine that someone handed us the perfect probability model p(y|x) that returns the “true” probability of a sequence of labels given an ingredient phrase. We want to use p(y|x) to discover (or *infer*) the most probable label sequence.

- Armed with this model, we could predict the best sequence of labels for an ingredient phrase by simply searching over all tag sequences and returning the one that has the highest probability.

Decoding

Option 2: Conditional Random Field (CRF)

- So given a model p(y | x) that encodes whether a particular tag sequence is a good fit for a ingredient phrase, we can return the best tag sequence. But how do we learn that model?

- A linear-chain CRF models this probability in the following way:

$$p(y \mid x) \propto \prod_{t=1}^{T} \psi(y_t, y_{t-1}, x)$$

where T is the number of words in the phrase x.

Decoding

Option 2: Conditional Random Field (CRF)

$$p(y \mid x) \propto \prod_{t=1}^{T} \psi(y_t, y_{t-1}, x)$$

The above equation introduces a “potential” function ψ that takes two consecutive labels, y_t and y_{t-1}, and the phrase, x.

We construct ψ so that it returns a large, non-negative number if the labels y_t and y_{t-1} are a good match for the *t* and *t−1*th words in the sentence respectively, and a small, non-negative number if not.

In our case, we’ll be feeding the hidden states of the LSTM into this function, which looks like this:

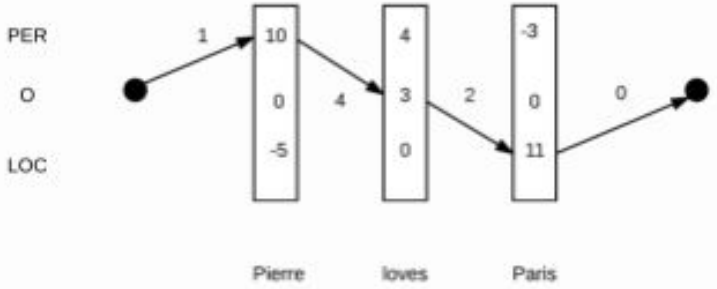
$$\text{score}_{lstm-crf}(x, s) = \sum_{i=0}^{n} W_{s_{i-1}, s_i} \cdot \text{LSTM}(x)_i + b_{s_{i-1}, s_i},$$

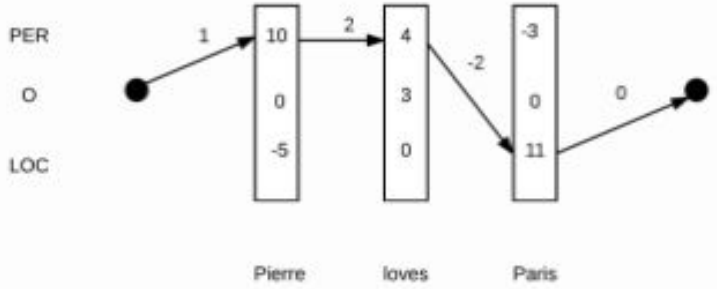
where W and b are the weight vector and the bias corresponding to the transition from s_{i-1} to s_i, (or y_{i-1} to y_i) respectively.

Decoding

$$C(y_1, \ldots, y_m) = b[y_1] + \sum_{t=1}^{m} s_t[y_t] + \sum_{t=1}^{m-1} T[y_t, y_{t+1}] + e[y_m]$$

$$= \text{begin} + \text{scores} + \text{transitions} + \text{end}$$





The path PER-O-LOC has a score of

$$1 + 10 + 4 + 3 + 2 + 11 + 0 = 31$$

The path PER-PER-LOC has a score of

$$1 + 10 + 2 + 4 - 2 + 11 + 0 = 26$$

Illustration of the scoring of a sentence with a linear-chain CRF. Between these two possible paths, the one with the best score is PER-O-LOC. Notice that if we make our decision locally, based on the score vector of each word, we would have chosen PER-PER-LOC

Decoding

For the purposes of this course, this is the furthest we need to go on Conditional Random Fields

To summarize:

- For prediction of a sequence that has no interdependency, softmax is an appropriate function
- CRFs go one step further by creating a weight matrix to determine the likelihood of transitions from tag to tag

More on this can be found at:

https://people.cs.umass.edu/~mccallum/papers/crf-tutorial.pdf