w266

gcloud compute ssh --ssh-flag="-L 8896:127.0.0.1:8896" --ssh-flag="-L 6006:127.0.0.1:6006" adamcy99@w266

Python PDB - import pdb; pdb.set_trace()

Lecture 1:
4-5 Assignments takes a LONG TIME (15-30 hours) Especially 3rd or 4th hw.
Quizzes!
Up to 5 late days (max 2 per assignment)

NLP is hard:
- So many features if we care about each word and their position
- A lot of sentences depends on the reader's interpretation
- Some words mean something 999/1000 times but means something else 1/1000 times.

AlphaGo played many games and learned based on the outcome of the games. Started with a large search space and filled them.

Entropy: Measure of chaos. Degree of how much information can be passed. High entropy = lots of different things we can pull out of a hat. Low entropy means it can be a few things. 0 entropy means it's 100% determined, only one choice.

Information Entropy: Abstract idea of "information" usually measured in bits. Information needed to specify a random variable.

$$\mathrm{H}(X) = \sum_{i=1}^{n} \mathrm{P}(x_i)\,\mathrm{I}(x_i) = -\sum_{i=1}^{n} \mathrm{P}(x_i) \log_b \mathrm{P}(x_i)$$

Expected length per symbol is the number of bits needed to represent a symbol. In the log, b is the number of possible outcomes. In binary entropy, b = 2.

BinaryEntropy(0.75) = -0.75log(0.75) - 0.25log(0.25) = 0.81 the base of the log is 2.

E[symbol_length] = ∑ p(symbol_i) * symbol_length_i

Exercise: P(A) = 0.5, P(B) = 0.25, P(C) = 0.25. What is the most efficient way to express this in binary? A = 0, B = 10, C = 11. To get the shortest expected length. (Note, don't want to set C = 1 because then it'll be hard to tell letters apart).

Cross-entropy: our cost function. When q = p, our H(p,q) function reaches a minimum.

Cross-Entropy

Cross-entropy: measure of information between two samples

P(x) = "true" distribution
Q(x) = predicted/estimated distribution

Given optimal encoding of for Q, how many bits to encode p ~ P?

$$H(p,q) = \mathrm{E}_p[-\log q] = H(p) + D_{\mathrm{KL}}(p\|q)$$

$$H(p,q) = -\sum_x p(x)\,\log q(x).$$

Pointwise Mutual Information: the measure of how much knowing one outcome tells you about another. For example: if I tell you it's raining, you'd probably guess it's cloudy. But if I told you one die rolled 6, you couldn't guess what the other die rolled. Applied to language, if you mention the word "rain", I learn that you're more likely to also say "wet" or "umbrella" in the same context.

$$\mathrm{PMI}(x,\ y) = \log_2 \frac{p(x,\ y)}{p(x)\,p(y)}$$

In short, it measures the chance two outcomes tend to co-occur (the numerator) relative to the chance they would co-occur if they were independent events (the denominator).

Neural Net: http://playground.tensorflow.org
Information passed from features to the next layer and multiplied by weights.

Weights is what we train.

Hidden Layer: X is input

Activation function.

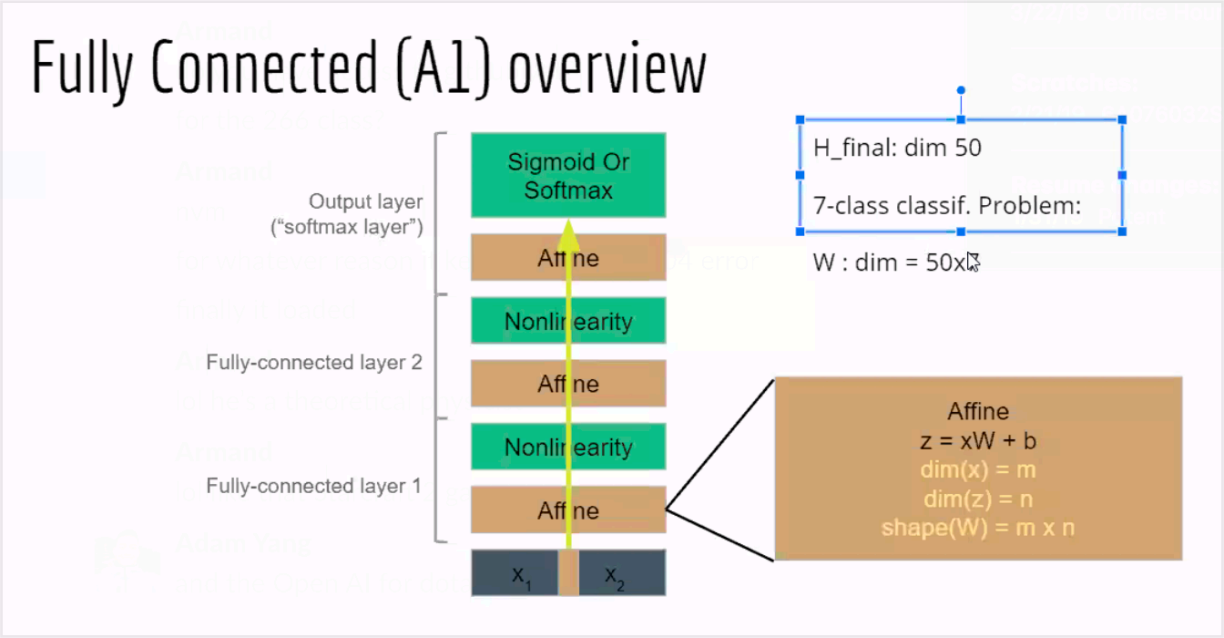
How would you optimize how many hidden layers and how many neurons per hidden layer?

I'm pretty sure that's the "art" part of neural networks. It's not as easy to use hyper parameter tuning, or something similar for something that's model architecture as opposed to an actual parameter. Generally speaking, you will be limited with the number of dimension and layers by the computing cost. Addtional layers CAN be useful to learning, but they aren't ALWAYS useful. You'll quickly reach diminishing returns

Affine Layer: xW + b

Batching send in many examples of x (each feature has many rows) throw it all in at once.

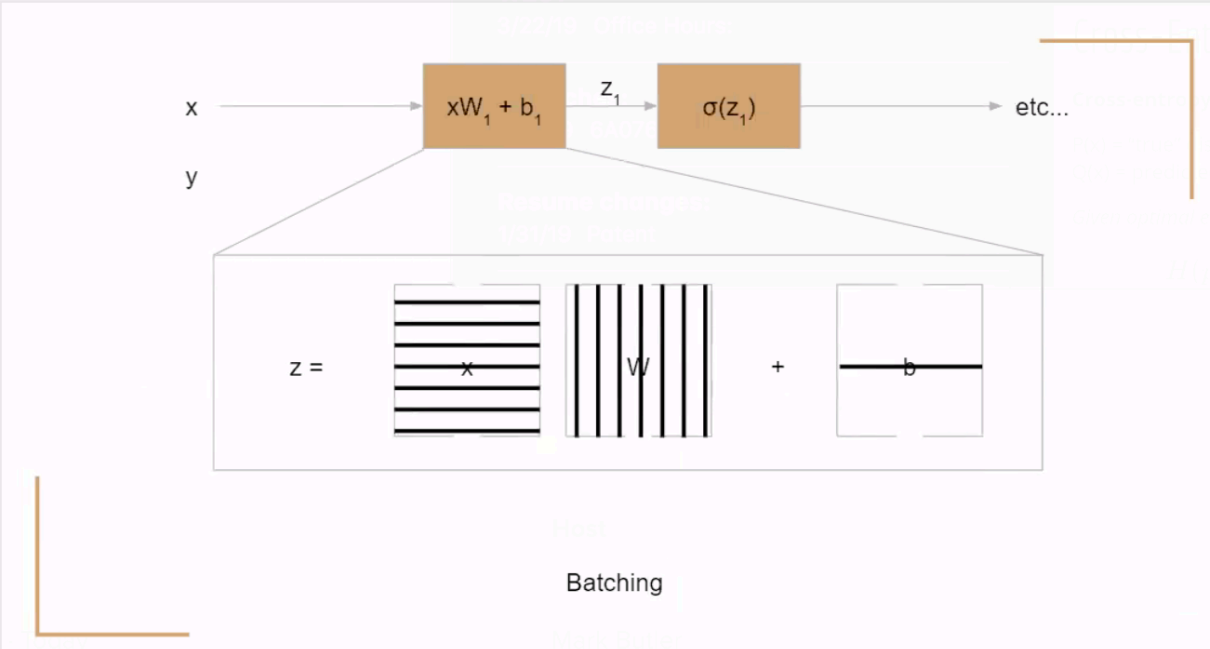
Use Sigmoid or Softmax at the end after the final Affine function to get probability.



Deeper layers are better than shallow long layers: limitations are training speed,

Nonlinearity functions = Activation functions, usually Sigmoid. Or commonly ReLus (when number is negative, set to 0)

Chain rule and back-propogation



Each column of W is a neuron.

If 4 neurons and 20 rows of data, and 15 parameters
x = 20x15, W = 15x4

Week 2

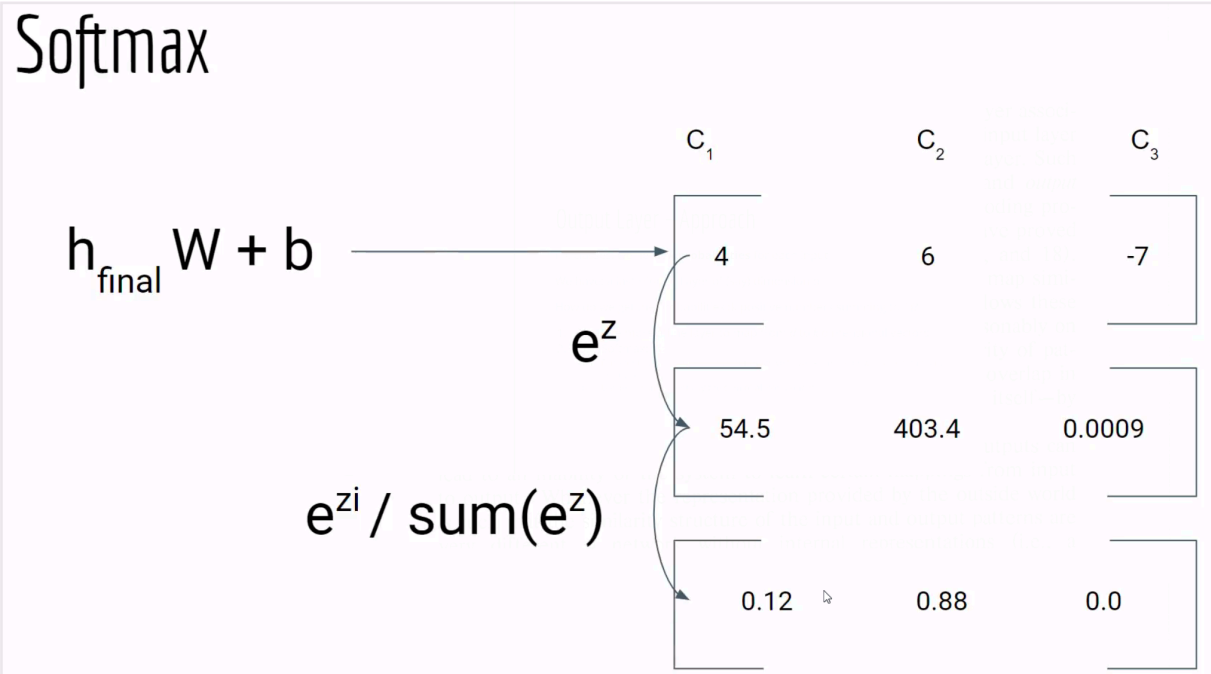
Output Layer - Approach

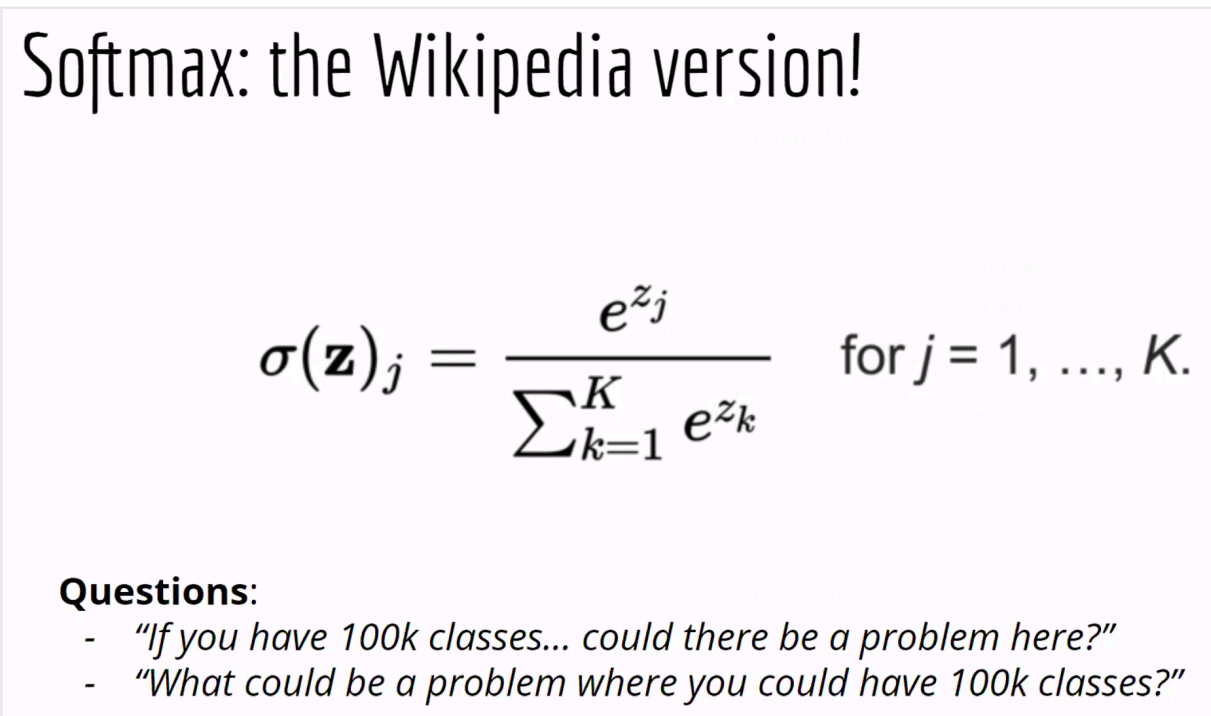
If *k* classes, we need: ***k* probabilities** for each input

We have: a last Hidden Layer of (say) dimension d

How do we get the probabilities (k positive numbers summing to 1)?

1. Use one more affine layer to map from *d* to *k* dimensional vectors
2. Use Softmax:
    a. Exponentiate all elements of vector (now we have k-dim vector with all components positive)
    b. Divide all components by the sum of all components





Sigmoid is Softmax with only 2 labels (yes or no). Sofmax is also called logistic regression.

Hyperameters: how many neurons per layer, number of layers, activation function, regularization

Fully connected neurons. All neurons are connected to all neurons in the next layers.

Training: Backpropagation

Backpropagation = Gradient Descent + Chain Rule

Want to minimize cross entropy (J). we care about p and q, p is the actual labels and q is the predicted labels. Solve for W.

W -> W-alpha*(dJ/DW)

Make predictions and get errors. Back propagate the errors to adjust the W.

Training: Backpropagation Math

$w := w - \alpha * \partial J/\partial w$

Apply to last layer:

$$z_f = z_{f-1} w_f + b_f$$

$$y\hat{} = \text{sigmoid}(z_f)$$

$$J = \text{xentropy}(y, y\hat{})$$

$$\partial J/\partial w_f = \partial J/\partial y\hat{} \times \partial y\hat{}/\partial z_f \times \partial z_f/\partial w_f$$

Zf = final affine layer, Z(f-1) is the previous affine layer

Training: Backpropagation Math

$\partial J/\partial w_f = \partial J/\partial y\hat{} \times \partial y\hat{}/\partial z_f \times \partial z_f/\partial w_f$

$J = \text{xentropy}(y, y\hat{}) = y \log(y\hat{}) + (1-y) \log(1-y\hat{})$

$\partial z_f/\partial w_f = z_{f-1}$
$\partial y\hat{}/\partial z_f = y\hat{} * (1- y\hat{})$
$\partial J/\partial y\hat{} = (y - y\hat{})/(y\hat{} * (1- y\hat{}))$

$\partial J/\partial w_f = (y - y\hat{}) * z_{f-1}$ (Simple! <u>And already calculated during prediction!</u>)

(**Note**: some care needs to be taken with dimensions and indices in general case, but this is good illustration.)

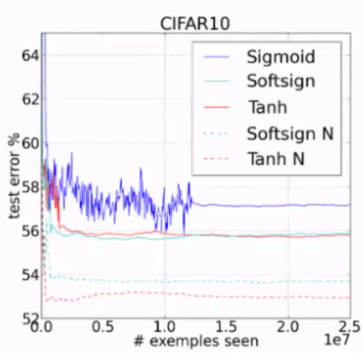
Note that y-y^ is the error. We back propagate the error backwards.

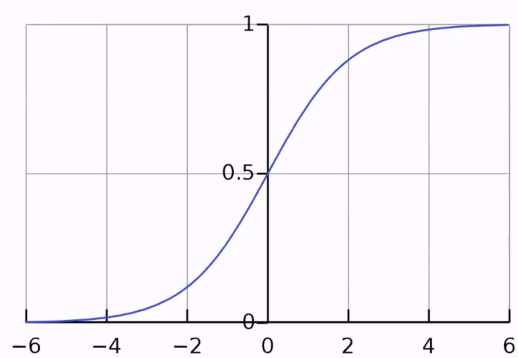
The sigmoid equation has a gradient problem. If a point sits too far out from 0, we will not learn because the gradient would approach 0. Use Xavier to initialize.

Initialization: Xavier

Generally, Xavier initialization for "w", 0s for "b".

Understanding the difficulty of training deep feedforward neural networks, Glorot and Bengio.





Similar to feature normalization but not exactly the same.

Optimizer (figure out how to optimize the hyperparamters and learning rate?)

TensorFlow

Tensorflow Ideas

In **Tensorflow** calculations are done first by setting up the plumbing ("graph") to define data flow. Tensors are data "nodes" in the graph.

Data flows sequentially through the graph *one step at a time*.
Each step progression is triggered by `session.run(...)`

The **Tensorflow** platform provides cutting-edge libraries for doing amazing things:

- simple things (such as adding two tensors together or multiplying vectors or matrices), or
- more complex things (such as optimizing a model to minimize cross-entropy loss.)

What's a Tensor?

Tensor: Just a fancy name for an array in tensorflow. Analogous to numpy arrays.

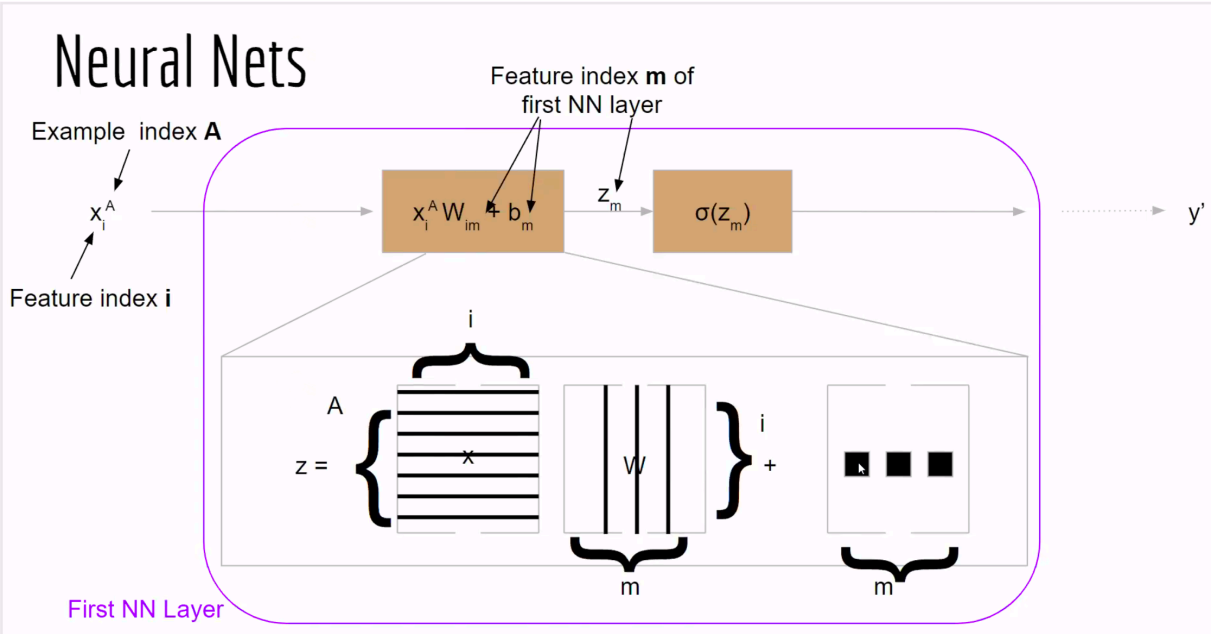
The most *important types* of tensors are:

- **Constants and sequences**, for example: tf.constant(), tf.zeros(), tf.linspace(). These create Tensors with a fixed value.
- **Variables**, are *persistent* tensors and can be modified during execution. They are used for model parameters, (for example weight matrices), and get updated by training.
- **Placeholders**, are used as data inputs. These are *empty* variables (hence the name) and must be fed with data during a session.run() operation.
- **Operations**, hold the result of an operation with 1 or more inputs. They can have inputs of placeholders, variables, constants, or other operations (depending on the operation.)

[6,] Vector (rank 1 tensor), [6,1] Matrix (rank 2 tensor), [,] scalar (rank 0 tensor).

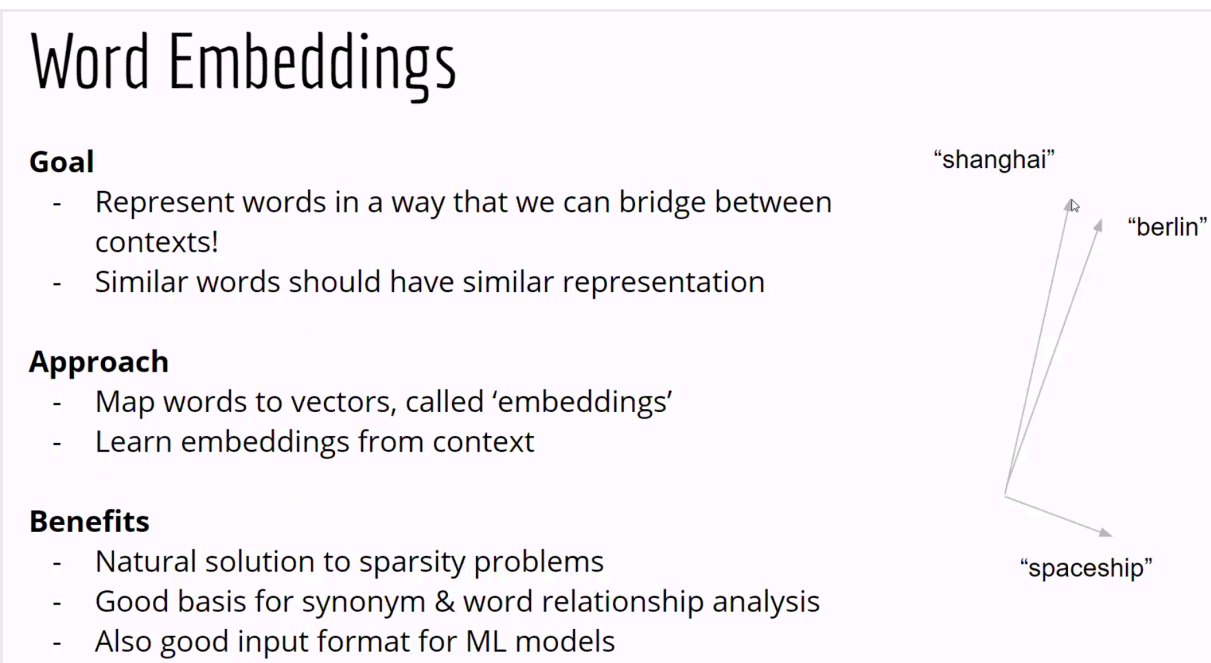
Week 3

KL Divergence = Cross entropy - Entropy (In HW 1)



Problems of numerical features
Problems with sparsity
Solution?

Convert words to vectors



Difference between plurals would be similar. So house-> houses would look like cat -> in terms of the vectors.

Cosine distance:

$$\cos\theta = \frac{\vec{w}_1 \cdot \vec{w}_2}{\|\vec{w}_1\|\|\vec{w}_2\|}$$

What is the difference in the angle between word vectors?

Implementation Details: Tuning Hyperparameters

Many things to try:

- Vocabulary size
- Word vector size
- Hidden layer size
- Learning rate
- Minibatch size and training schedule
- Sampled loss functions?
- Dropout?
- Optimizer (SGD, AdaGrad, Adam, RMSProp, etc.)?

True for non-neural algorithms too! But **NNs can be very sensitive.**

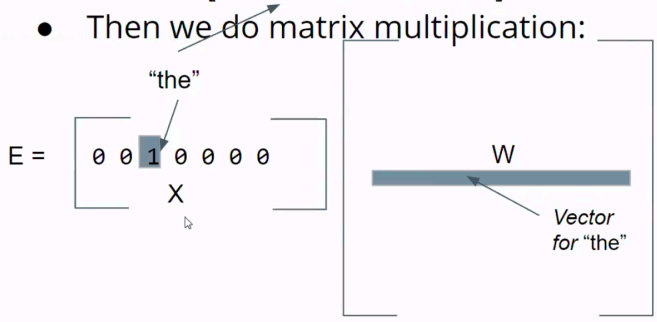
Might want to cut off long tail of vocab (uncommon words)

Week 4

Recap: Word Embeddings - convert vocabulary of 10000 words into vectors with x amount of bits. Define word vectors, will be very sparse.

Mathematically equivalent: one-hot & word vectors

- Usually presented as a "lookup" in some table, but...
- **(Only) think** of a word as a one-hot vector:
  - `X = [0 0 1 … 0 0 …. 0 0 0]`
- Then we do matrix multiplication:



Word Embeddings!

- Reasonable that similar words end up with similar vectors

Similar words have smaller cosine similarity. Vector angle is smaller. Each row in W

matrix is a word.

What's CNN? Convolution Neural Networks

Why Not Just Standard FC NN for Images and Text?

Way too many parameters!

Images:

Imagine 256 x 256 image in 3 colors, fully connecting to 256 x 256 x 12 dimensional output layer... how many parameters?

$$\underbrace{(256 \times 256 \times 3) \times (256 \times 256 \times 12)}_{\text{(weights)}} + \underbrace{(256 \times 256 \times 12)}_{\text{(biases)}} \sim \mathbf{154\ bn}$$

Text:

Imagine you concatenate 30 words with 100-d embedding, fully connecting to a layer of same size... how many parameters?

$$\underbrace{(30 \times 100) \times (30 \times 100)}_{\text{(weights)}} + \underbrace{(30 \times 100)}_{\text{(biases)}} \sim \mathbf{9m}$$

That's Why: Convolutional Neural Nets (CNNs)

Greatly reducing parameters!

Critical Observations:

- Many features are 'local', i.e., one needs to inspect only small neighborhood to detect
- Often, it does not matter where an object or a word is ("I just care that there is a bird, not where it is")

Core Idea:

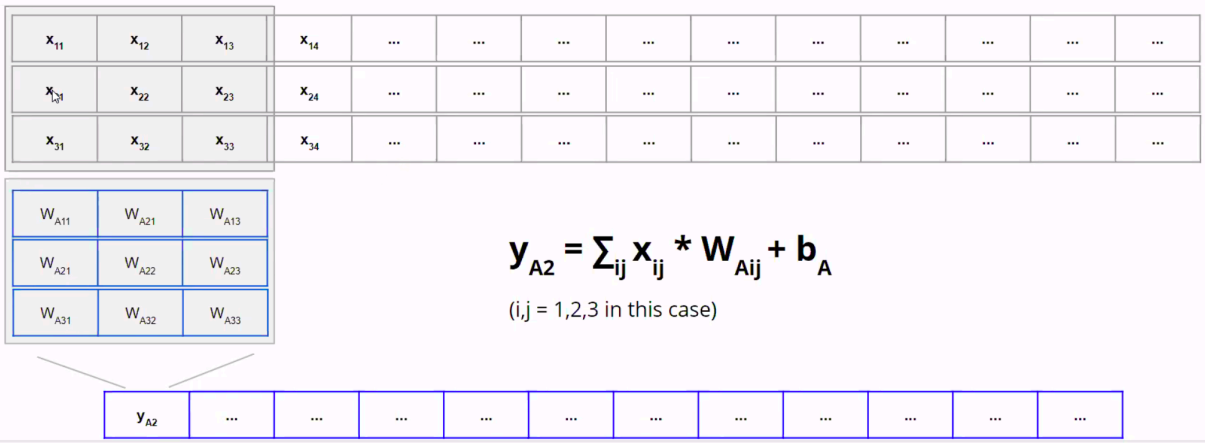
"Slide small window across image/text and that looks for features locally and that connects neuron of input with neurons of output layer."

- Neurons in output layer only 'see' small neighborhood
- The same parameters are used across image/text. MUCH cheaper!

Early layers learn concrete features (edges, etc.), Deeper layers learn more abstract features.

Layers: How Do We Continue?

Filters are now matrices...



$$y_{A2} = \sum_{ij} x_{ij} * W_{Aij} + b_A$$

(i,j = 1,2,3 in this case)

Regularization: Dropout

Still a ton of parameters. Regularization, anyone?

Dropout is a good option (regular regularization works too..)

- Invented by Geoff Hinton and team (U. Toronto), 2012 (https://arxiv.org/pdf/1207.0580.pdf)
- What it does:
  - Randomly drop (set to zero) a fraction of activations during training. Don't do that during evaluation(!)
  - Ensures that more robust features are learned
- Yet another hyper-parameter:
  - Dropout rate

Do dropout at every layer to regularize.

CNN Hyperparameters

Collecting an **(incomplete) set of key hyper-'parameters':**

- Number of layers
- Types of layers
- Shape of layers (3D in our example!)
- Padding approach
- Filter dimensions
- Step sizes
- Pooling approach
- Dropout rate
- Optimizer…
- …

…..lots to try!

Week 5
Markov Assumption
Let's say we want to form a sentence. We would have P(w1|START)P(w2|w1)P(w3|w2)… etc. but this has a scarcity problem because a string of words do not always occur often.
With the Markov assumption, we assume that only the previous word matters in terms of telling us what the next word should be P(wt|wt-1) (bigram). In a trigram model, we assume the previous 2 words are important in predicting the next word P(wt| wt-1, wt-2). In a unigram model, we assume that we can know the probability of a word occurring independent of context P(wt).

Build LM successively from prior sequences:

$$P(w_1, w_2, w_3, \ldots, w_n) = P(w_1, w_2, w_3, \ldots, w_{n-1}) \times P(w_n | w_1, w_2, w_3, \ldots, w_{n-1}) = \prod_{i=1}^{n} \boxed{P(w_i | w_1, w_2, \ldots, w_{i-1})}$$

Markov Assumptions for LMs

$P(w_1, \ldots, w_n)$

$= \prod_t P(w_t \mid w_{t-1}, w_{t-2}, \ldots, w_0)$	Why can't we use this directly?
$\cong P(w_t \mid w_{t-1})$	Assume that w_{t-1} contains all we need to know about the history (bigram model).
$\cong P(w_t \mid w_{t-1}, w_{t-2})$	Trigram model
$\cong P(w_t)$	Unigram model

Parameter Estimation

The parameters of an *n*-gram model:

- Relative frequency (maximum likelihood)
- Conditional probability estimates: θ

$$\hat{P}\left(w \middle| w_{-1}\right) = \frac{c(w_{-1}, w)}{\sum_{w'} c(w_{-1}, w')}$$

Top is the count of “the door”, bottom is the count of “the *”

Unigram Model: Probability of a sentence P(w1...wn) = P(w1)P(w2)...

Measuring Model Quality:
Entropy: Improve my 0.1 bits
Perplexity = entropy but scaled up. 2^entropy

Perplexity

- Problem with entropy: Improvement of 0.1 bits doesn't sound impressive.
- "Solution": perplexity.

$$\text{perp}(X,\theta)=2^{H(X|\theta)}$$

- Interpretation: average branching factor of the model.

Easy to get bogus perplexities if your probabilities are not normalized correctly (perplexities need to add up to 1)

Extrinsic Evaluation

Extrinsic Evaluation

Word error rate (WER) is task specific and directly relevant.

$$\frac{\text{insertion} + \text{deletions} + \text{substitutions}}{\text{true sentence size}}$$

Correct answer: Andy saw a part of the movie
Recognizer output: And he saw apart of the movie

WER: 4/7 = 57%

Insertion errors + deletion errors + substation errors.

N-Gram Smoothing
Sparcity is a big problem, especially if you increase the size of the n-gram.
Zipf law describes relation between word types and tokens. Word frequency is inversely proportional to word rank

Maximum likelihoods will get us in trouble. We will have a lot of zero probabilities of word pairs we haven't seen. There are terms that can be poorly estimated. We need smoothing (e.g. Naive Bayes) to make the probabilities more realistic.

Smoothing flattens spiky distributions so its more smooth.

Generalizing From a Corpus

Suppose we have a 1M word corpus—thousands of types, some frequent.

- Problems with estimating *n*-gram probabilities:
  - Known words in unseen contexts
  - Entirely unknown words
- Goals:
  - Use higher-order *n*-grams when possible.
  - Use lower-order *n*-grams when necessary.

Backoff Smoothing (Katz)
If count(history) < k, reduce the size of h. Trigram -> bigrams -> unigrams

Interpolation:

$$\lambda P(w|w_{-1},w_{-2})+\lambda' P(w|w_{-1})+\lambda'' P(w)$$

$$\lambda+\lambda'+\lambda'' = 1$$

P(door | close, the)
= λP(door | close, the)
+ $\lambda' P$(door | the)
+ $\lambda'' P$(door)

There are reasonable ways to estimate the λs.

Add 1 Smoothing (Laplace Smoothing)

Laplace Smoothing

Add a small number to each *n*-gram count.

$$P_{\text{add-}\delta}(x)=\frac{c(x)+\delta}{\sum_{x'}(c(x')+\delta)}$$

However, sometimes Laplace smoothing overestimates fraction of new bigrams.

Held-Out Reweighting (Good-Turing)

Held-Out data (development data) are held out from training. Used to help reweighs the probabilities.

Kneser-Ney Smoothing

Absolute Discounting

Count in 22M Words	Future c* (Next 22M)
1	0.448
2	1.25
3	2.24
4	3.23

Suppose there's some true distribution of *n*-grams.

- A corpus is just a small sample.
- *N*-grams can't have fractional counts in the corpus.
- Observed *n*-grams appear more than they will later.

Absolute discounting: Just subtract 0.75.

$$P_{\text{ad}}(w|w') = \frac{c(w',w) - d}{c(w')} + \alpha(w')\widehat{P}(w)$$

Normalizing term on there right of equation is because we lost some probability mass, so we have to add that probability mass back in evenly.

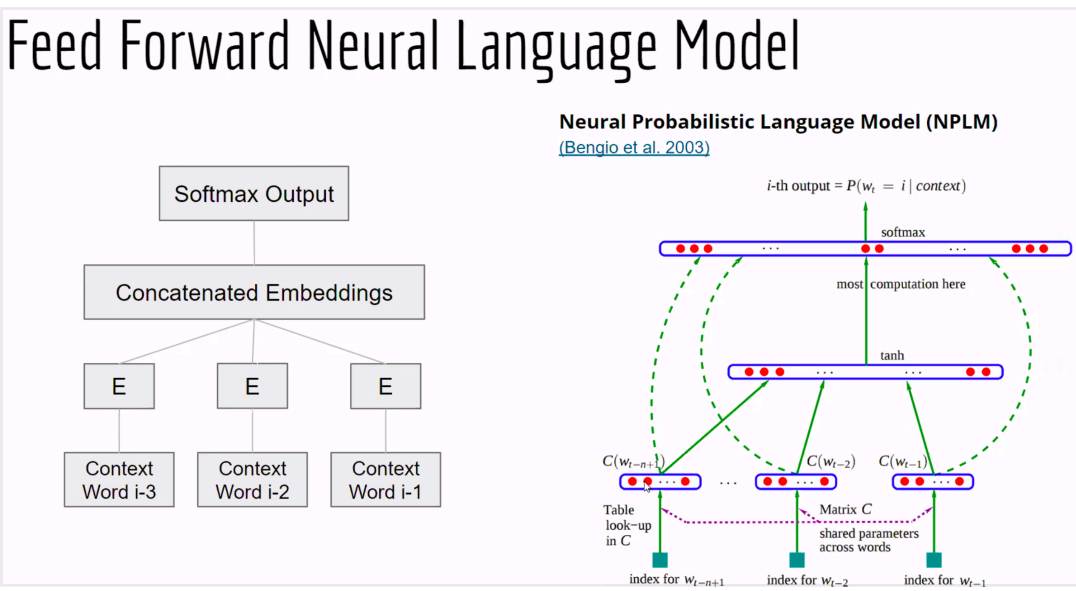
Type Fertility

- Shannon game: There was an unexpected ____? [*delay* or *Francisco*]?
  - "Francisco" is more common than "delay."
  - ...but "Francisco" always follows "San."
  - ...so it's less fertile.
- Solution: type-continuation probabilities
  - In the backoff model, we don't want the unigram $P(w)$.
  - Instead, we want the probability that w is allowed in a novel context.
  - For each word, count the number of bigram types it completes.

How many unique words does each word follow?

Word Embeddings. Usually we get similar word relationships man->woman would have similar vector as boy->girl.
However, 1 problem is inherent bias in text where computer scientist -> homemaker vector is also similar. Or when 1 word has many meanings but has only 1 word embedding, that vector would be very confused.

Window Models:



Concatenate the 5 size 100 vectors. To do tanh. y(x) = tanh(x W +b) + x. The dotted lines send the 5 size 100 vectors to the end to add. Then you get y = x plus some delta. If tanh = 0, y = x.

$$y = b + Wx + U\tanh(d + Hx)$$

No more sparcity issues because as long as you have an embedding for it. Word embedding allow us to learn from words that are similar.

Limitations of Window Models?

- **Blind to Input Text Outside of Window**
  Text outside of the fixed window is ignored when making predictions.
- **Ignores Surrounding Predictions**
  Predictions for the prior and next sequence predictions don't inform current position.*
- **No Modeling of Whole Sequence Level Information**
  Makes no attempt to integrate information across a sequence an use if for making predictions.

Why still use them? ***Fast to train and use.***

Spends most time calculating in the last layer (softmax) because the size of that layer is huge calculate probability of all the words.

Oh... One More Thing: Optimizations

A language model is a classifier: P(next word | previous words).

Challenge:

Softmax can get really expensive for normal vocabulary size!

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

Need to sum all exp(z_i)! How many K?

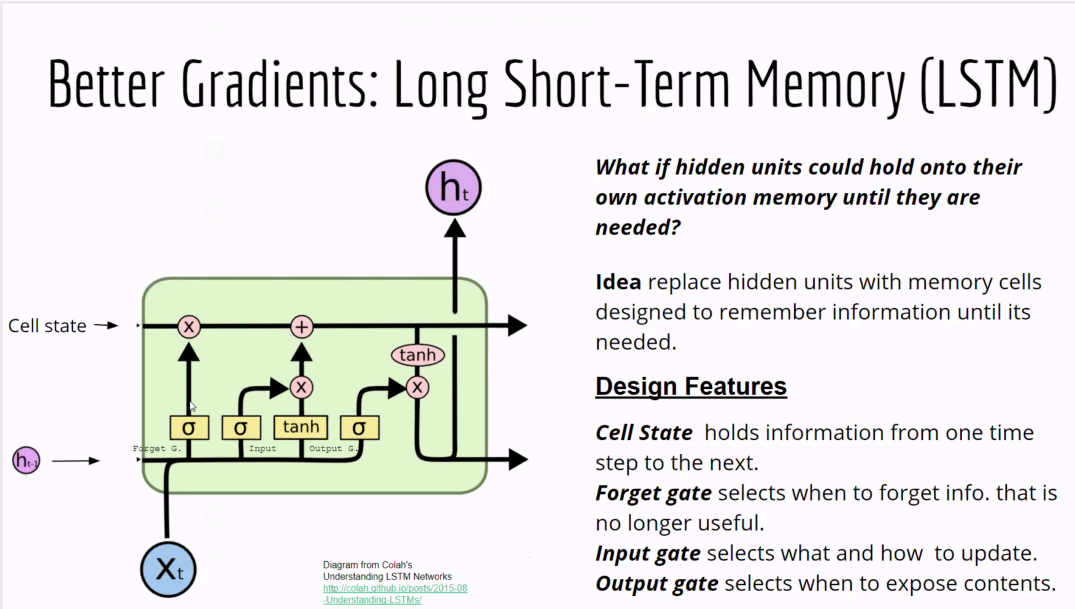
Lazy Softmax - Look at week 5 ppt.

Hierarchical Softmax - Look at week 5 ppt.

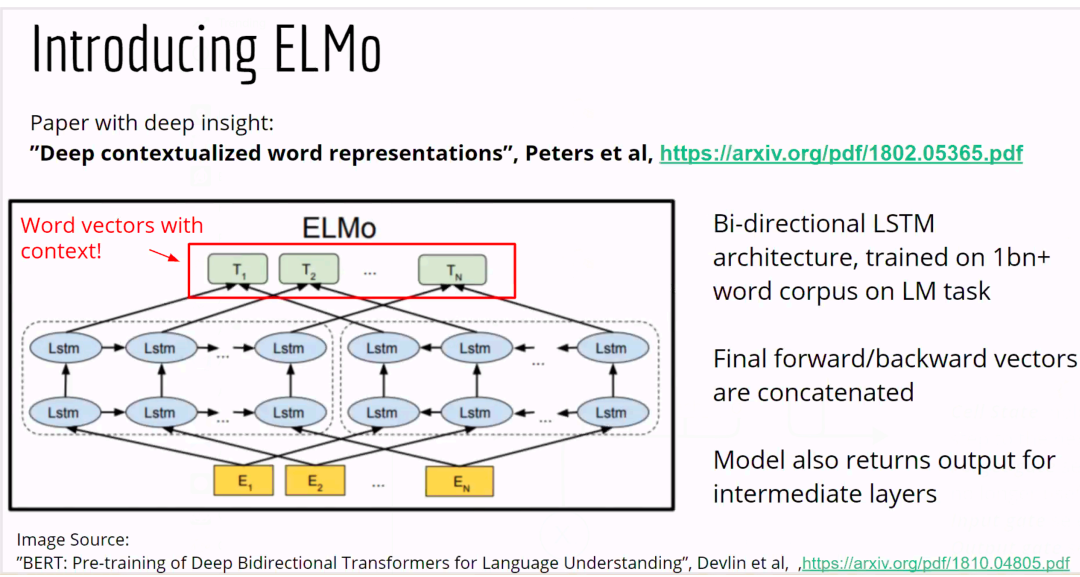
The depth of the tree is Log2(number of words)

Week 6

The standard A box has exploding and vanishing gradient issues.



Is better for the problem and mostly avoids the exploding/vanishing gradient problem. LSTM is different than RNN because you add things. Easier to maintain state over long sequences.



You predict what a word is based on stuff before it in one model, and also stuff after it in the second model

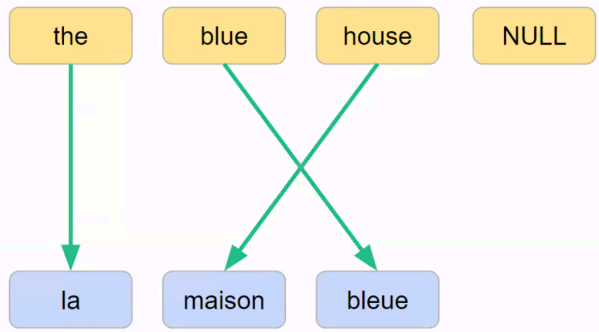
Week 7

IBM Model 1

Noisy-channel translation **f** → **e**

Model: generative process **e** → **f** with P(f | e)

1. Sample a length m ~ Q(m)
   - Distribution Q(m) doesn't matter!
2. Sample an alignment **{a$_i$}** for i = 1,...,m
   - **a$_i$** ~ Uniform(0, ℓ + 1)
   - **a$_i$** independent
3. Sample a word **f$_i$** from **t(f | e$_{ai}$)**
   - **t(f | e)** tabulated



$$P(f, a \mid e) = \frac{\epsilon}{(\ell + 1)^m} \prod_{i=1}^{m} t(f_i \mid e_{a_i})$$

L is the count of the source language.

Talked about BLEU formulation - why we have each parts of the BLEU evaluation algorithm

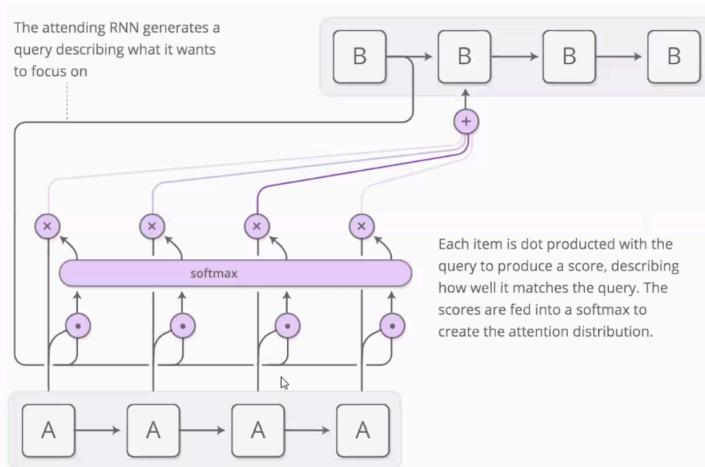
Attention Models

Problem: fixed-length vector representation

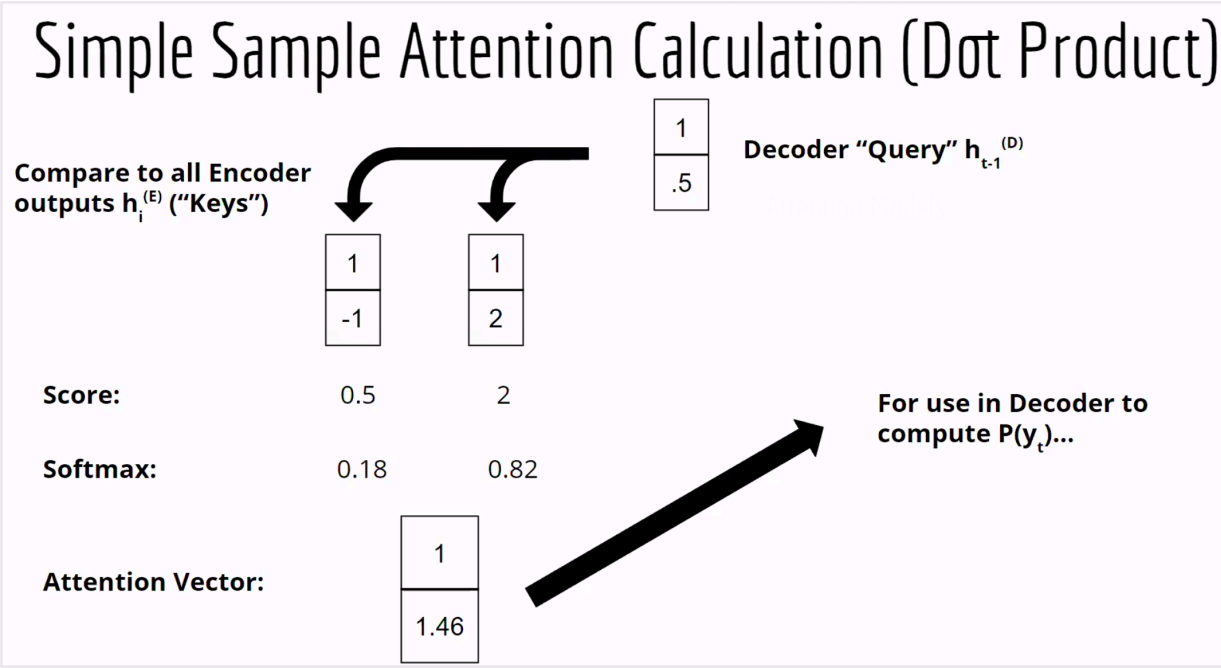
- Bad for long sentences!

Idea: allow model to look back at input sequence during decoding

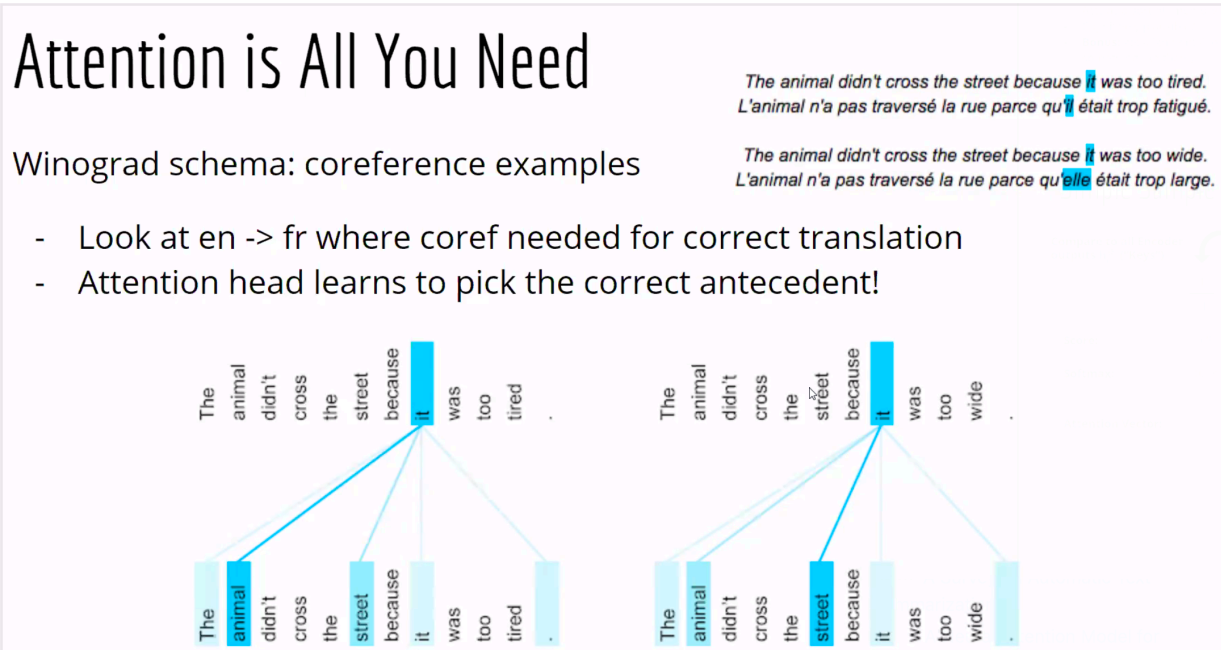
- Acts as a sort of "memory"
- Cognitively plausible?
- **Bonus:** get alignments! (or something like them)



A is the encoder network, B is the decoder network. First tilmestep we get something in the decoder, but something into the second timestep (augment it with reference to the previous stage). Dot product of output and input.



Week 8
The attention model issue is that you can't parallelize it and so it takes a long time. That's why we do self attention model.



Able to tell what "it" was referring to, animal or street.

Transfer Learning:
Can I use a big pre-trained corpus to help train my small corpus?

Week 9
Abstractive vs Extractive Summarization
- Abstractive means you use your own words

- Extractive means you use the given words to create summaries

Question-Answering = Summarization w.r.t a query

Summarization: Evaluation

Many criteria:

- Information satisfaction (answer query)
- Coverage (summarize corpus)
- Fluency (sounds natural)
- Concision (not redundant, no fluff)

Hard to measure **and** hard to tune for!

- Data: reference summaries (similar to MT)
- No "right answer" - hard to optimize (are we overfitting reference?)

Summarization: Evaluation

ROUGE-N

$$= \frac{\sum_{S \in \{ReferenceSummaries\}} \sum_{gram_n \in S} Count_{match}(gram_n)}{\sum_{S \in \{ReferenceSummaries\}} \sum_{gram_n \in S} Count(gram_n)}$$

In **BLEU**, we cared about **precision**:

- How many n-grams in the candidate are in a reference translation?

In **ROUGE**, we care about **recall**:

- How many n-grams in the reference summaries are in the candidate?

"Did we include all the salient points in our limited length summary?"

Summarization: Evaluation

ROUGE-N

$$= \frac{\sum_{S \in \{ReferemceSummaries\}} \sum_{gram_n \in S} Count_{match}(gram_n)}{\sum_{S \in \{ReferenceSummaries\}} \sum_{gram_n \in S} Count(gram_n)}$$

Several forms of ROUGE.

ROUGE-N: N-gram overlap as described on previous slide

ROUGE-L: Longest common subsequence

ROUGE-S: Skipgram (any pair of words in a sentence)

Longest common subsequence problem:

AQBQC
ABC
ATTBC

Longest common subsequence is 3 for each pair of the three above: A,B,C. Stuff in the middle doesn't matter. (Fun Interview question)

Attention Models are abstractive

Attention Models

Problem: fixed-length vector representation

- Bad for long sentences!

Idea: allow model to look back at input sequence during decoding

- Acts as a sort of "memory"
- Cognitively plausible?
- **Bonus:** get alignments! (or something like them)

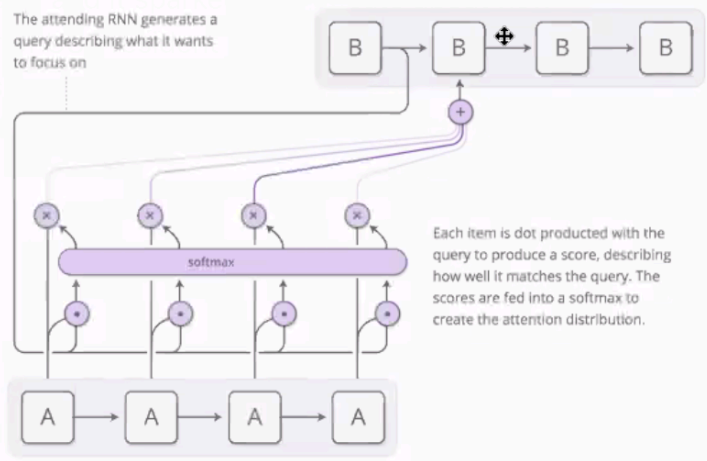


Image by Chris Olah and Shan Carter

Can we get out of this choice of abstractive vs. extractive?

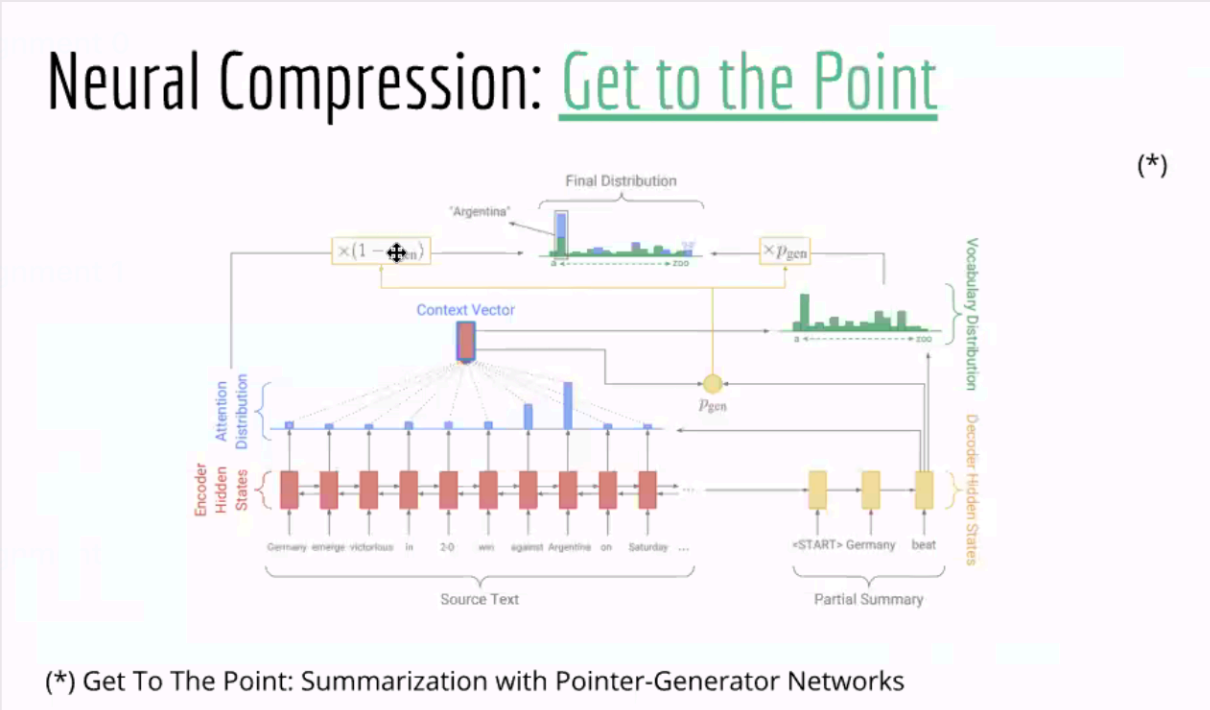
Neural Compression: Get to the Point

- Abstractive? Extractive? Why pick?
- Idea: sometimes copy text from source material & sometimes find a new word

Concretely:

- Build an attention model with a softmax over a RNN as usual
- Add to this structure p(generate), the probability we generate from the softmax (vs. just copying a word)
- Final word distribution is:

p(generate) x softmax output + (1 - p(generate)) x attention



Attention Model and mix the Softmax with the attention output to get the Final distribution. Result sometimes uses extraction, and sometimes abstraction method.

Week 10

Summarization - BIO = Beginning, Inside, Out (Other)

How to predict?

Sequence tagging task: for each word, assign {PER, LOC, ORG, MISC}

- Need BIO encoding to signal endpoints

O	O	B-LOC	I-LOC	B-LOC
train	fares	New	York	Boston

O	O	B-LOC	I-LOC
train	fares	Boston	Massachusetts

B,I,O + PER, LOC, ORG, MISC

Examples

Sid Reddy lives in New York and Rob stays in Seattle.

Sid->B-PER Reddy->I-PER lives->O in->O New->B-LOC York->I-LOC and->O Rob->B-PER stays->O in->O Seattle->B-LOC.

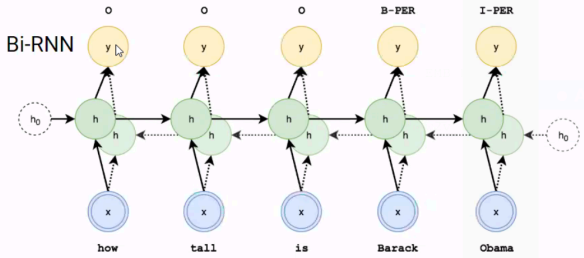
Her Majesty's Theatre is in London England and Madison Square Garden is in New York New York.

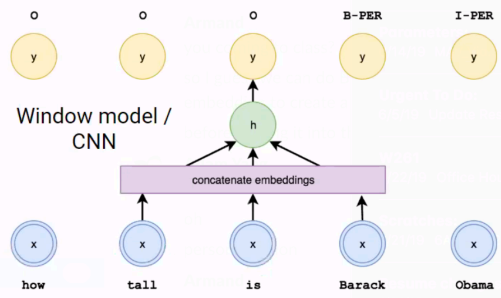
<Breakout exercise>

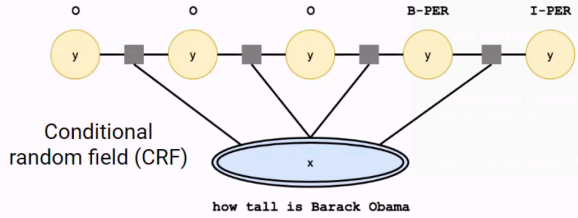
Her (B-LOC) Majesty's (I-LOC) Theatre (I-LOC) is (O) in (O) London (B-LOC) England (I-LOC) and (O) Madison (B-LOC) Square (I-LOC) Garden (I-LOC) is (O) in (O) New (B-LOC) York (I-LOC) New (B-LOC) York (I-LOC).

How to predict?

Standard sequence-labeling models







Brainstorm: features?

Bi-Directional for words and meaning (B-PER, etc) can add a third like part of speech features (verb, noun, etc)

Hearst Patterns

Handful of hand-written patterns,

- "...X, such as the Y"
- "Y, Y, and/or other X"
- "X, including Y, Y … and Y"
- "X especially Y and/or Y"

Brainstorm: what are we missing?

Knowledge Bases

WordNet: low-level semantic relations

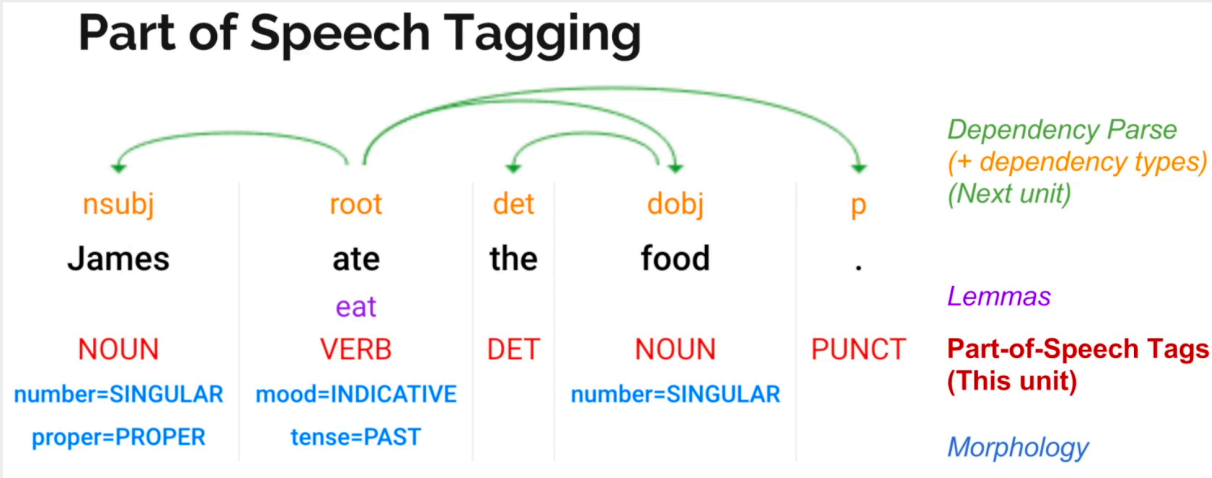
- Hypernymy: `(Giraffe, hyponym-of, animal)`
- Meronymy: `(leg, part-of, chair)`

Freebase / Google Knowledge Graph: world knowledge

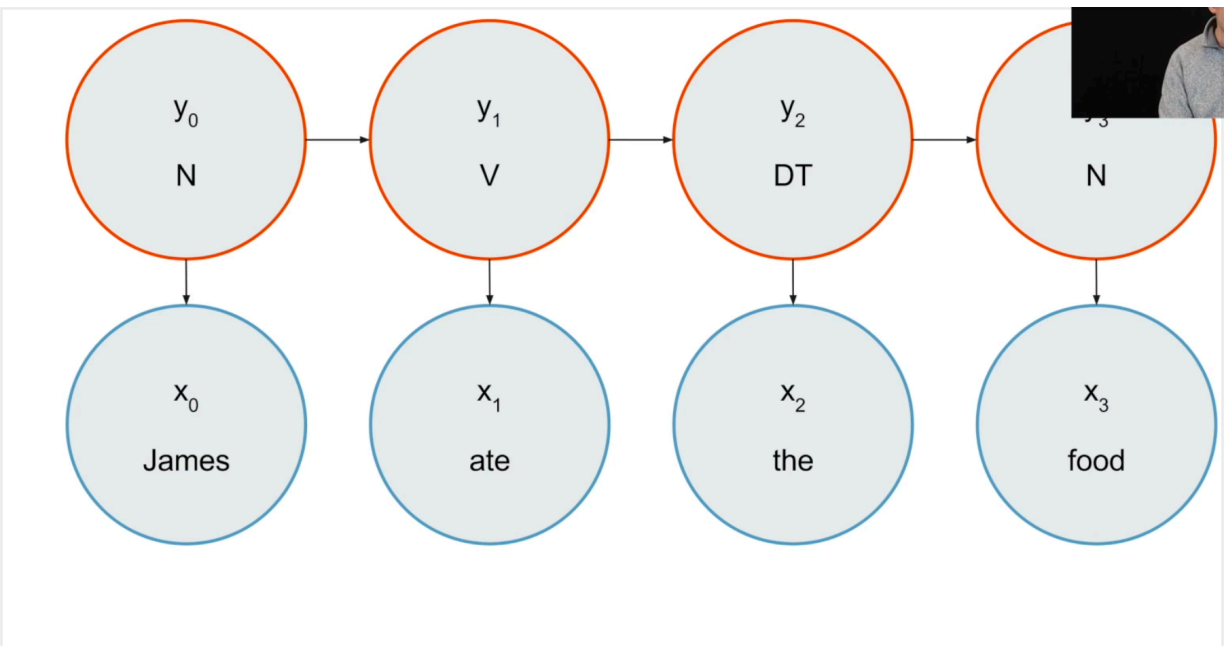
- `(Fight Club, /film/film/starring, Brad Pitt)`
- `(Barack Obama, /people/person/date_of_birth, 1961-08-04)`

Week 11

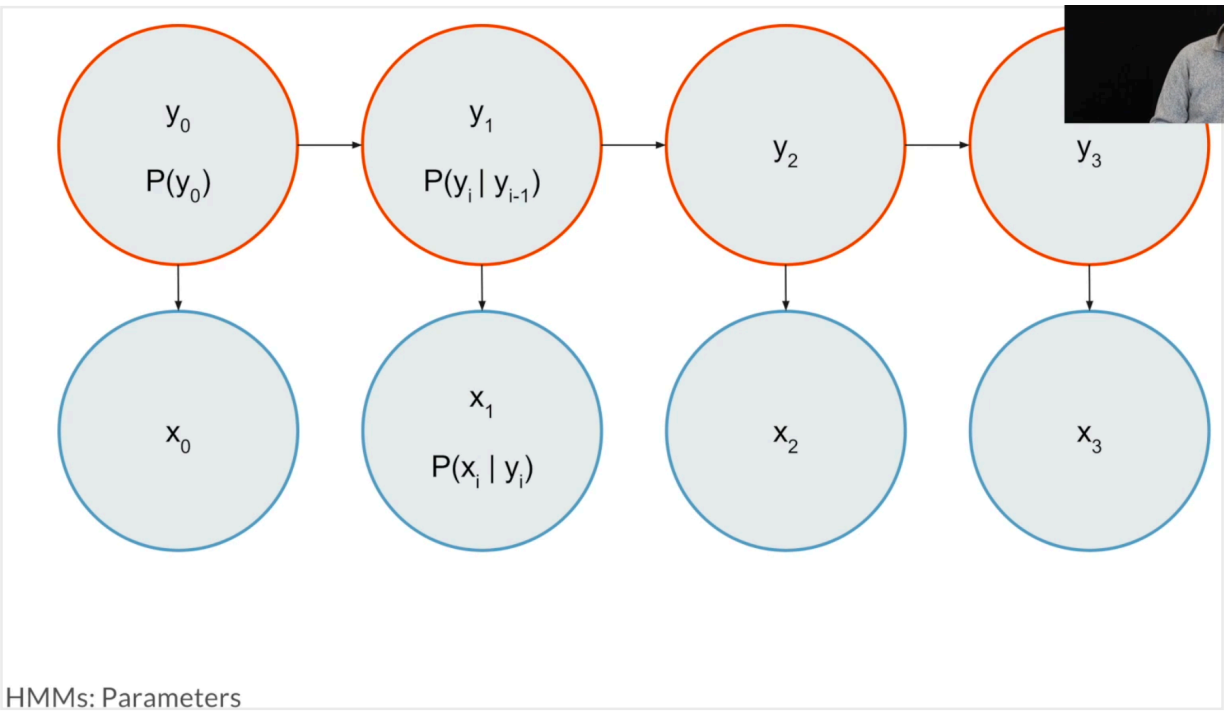
Part of Speech tagging using Hidden Markov Models - is a word a Noun, verb, punctuation, etc...



Hidden Markov Models (HMM) models a sequence of states. Each step has an internal state and also can output something into the world. Hidden and Visible parts of the model. Part of speech is the hidden part, the words itself are the visible part.



The End Goal: Determine sequence of states, y, from the observed sequence of emissions, x.

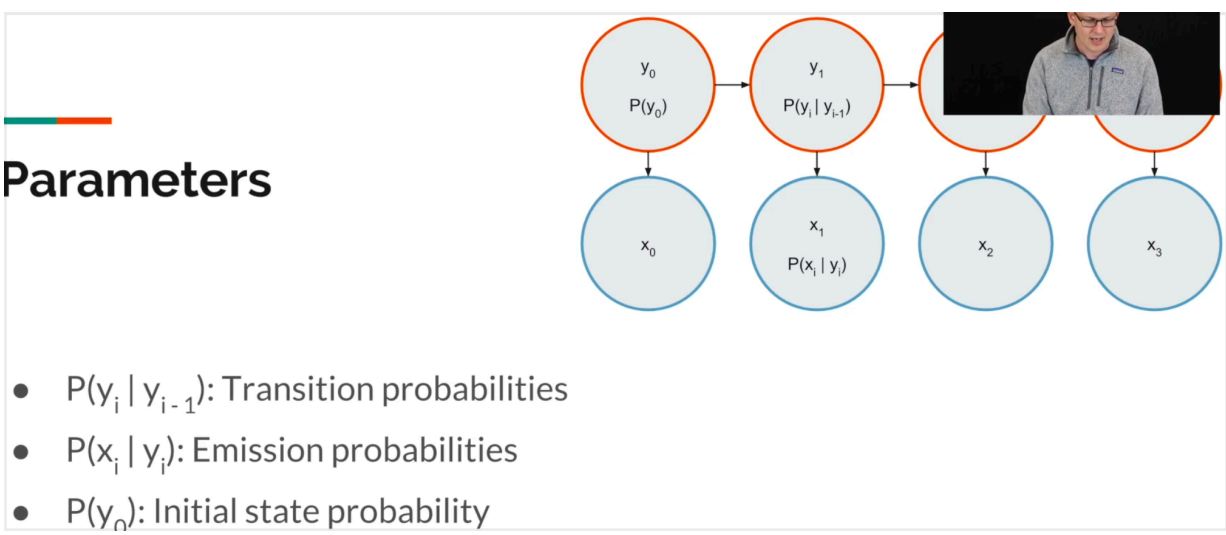


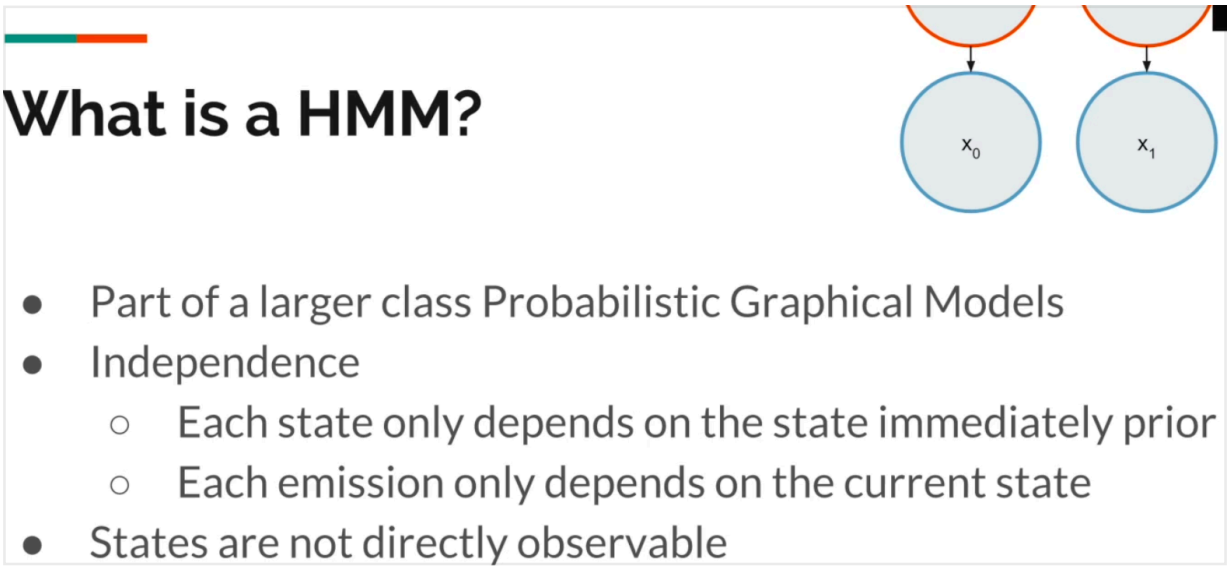
The parameters of Hidden Markov Models:

Transition probabilities: If the machine is in one state (state A) what's the probability that it transitions to state A in the next time stamp, probability that it transitions to state B, etc. (Many states in our case, Noun, Verb, adjective, etc)

Emission probability: If I'm in the state 'Noun', what's the probability that I output, James, or Door, or Run, etc. Run given Noun would be really low.

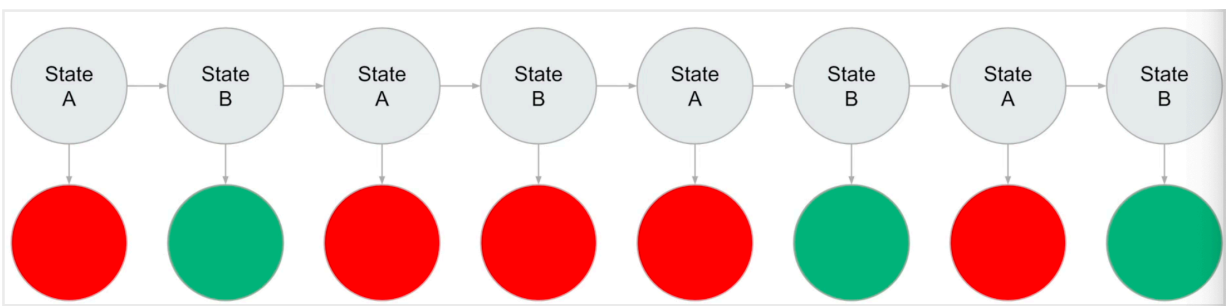
Initial state Probability





Why Markov? Because this assumes independence (current state only depends on the one state previous to it).

Training Hidden Markov Model:



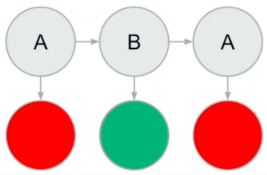
Transition probability takes the ratio of times State A goes to State B over total number of State A's to find the transition probability of State A going to State B. In the example above, the transition probability of State A -> State B is 1.

$$P(y_0, y_1, \cdots y_n | x_0, x_1, \cdots, x_n) = \frac{P(y_0, y_1, \cdots y_n, x_0, x_1, \cdots, x_n)}{P(x_0, x_1, \cdots, x_n)}$$

$$\propto P(y_0, y_1, \cdots y_n, x_0, x_1, \cdots, x_n)$$

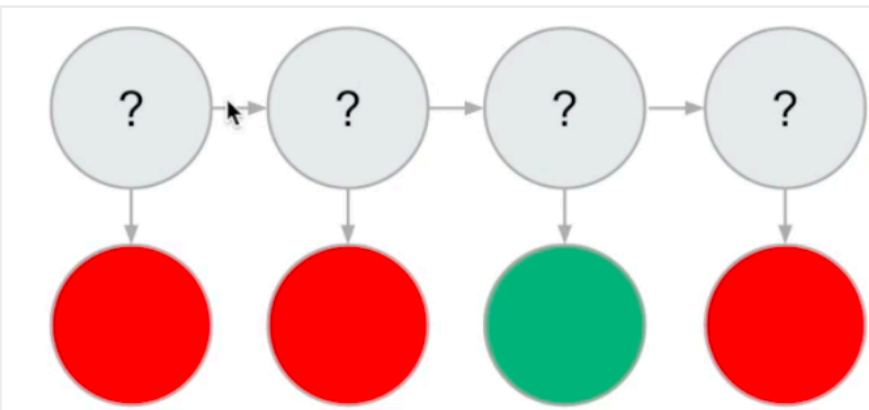
Scoring a Sequence

Initial State	Transition	Emission
$P(y_0 = A) = 1.0$	$P(y_i = A \mid y_{i-1} = B) = 1.0$	$P(x_i = \text{Red} \mid y_i = A) = 1.0$
$P(y_0 = B) = 0.0$	$P(y_i = B \mid y_{i-1} = A) = 1.0$	$P(x_i = \text{Red} \mid y_i = B) = 0.25$
		$P(x_i = \text{Green} \mid y_i = B) = 0.75$



$$P(y_0, y_1, y_2, x_0, x_1, x_2) = [P(y_0) \times P(x_0 \mid y_0)] \times [P(y_1 \mid y_0) \times P(x_1 \mid y_1)] \times [P(y_2 \mid y_1) \times P(x_2 \mid y_2)]$$
$$= [1.0 \times 1.0] \times [1.0 \times 0.75] \times [1.0 \times 1.0]$$
$$= 0.75$$

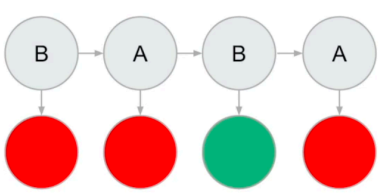
Training:



Given the visible states, we calculate all the P(y0,y1,y2,y3,x0,x1,x2,x3) values where y0,y1,y2,y3 are variable and pick the one that has the highest probability.

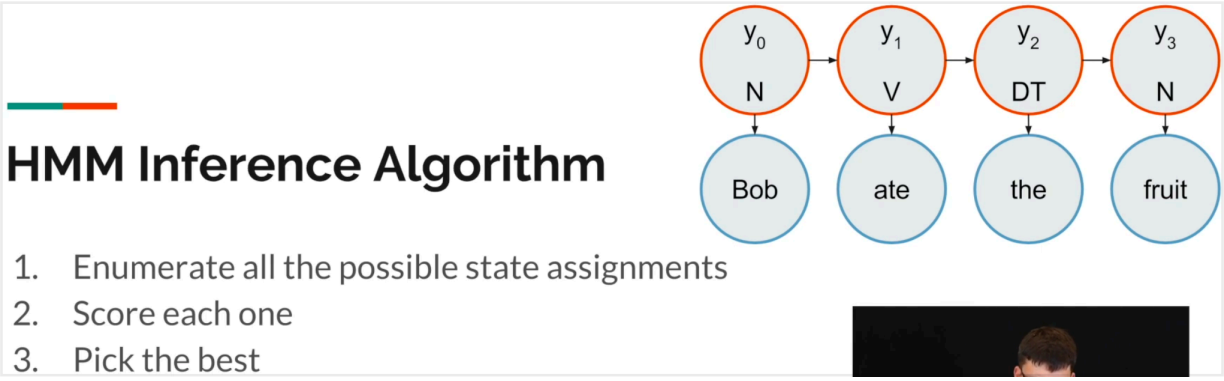
Try them all, pick the best

Initial State	Transition	Emission
$P(y_0 = A) = 0.5$	$P(y_i = A \mid y_{i-1} = B) = 1.0$	$P(x_i = \text{Red} \mid y_i = A) = 1.0$
$P(y_0 = B) = 0.5$	$P(y_i = B \mid y_{i-1} = A) = 1.0$	$P(x_i = \text{Red} \mid y_i = B) = 0.25$
		$P(x_i = \text{Green} \mid y_i = B) = 0.75$



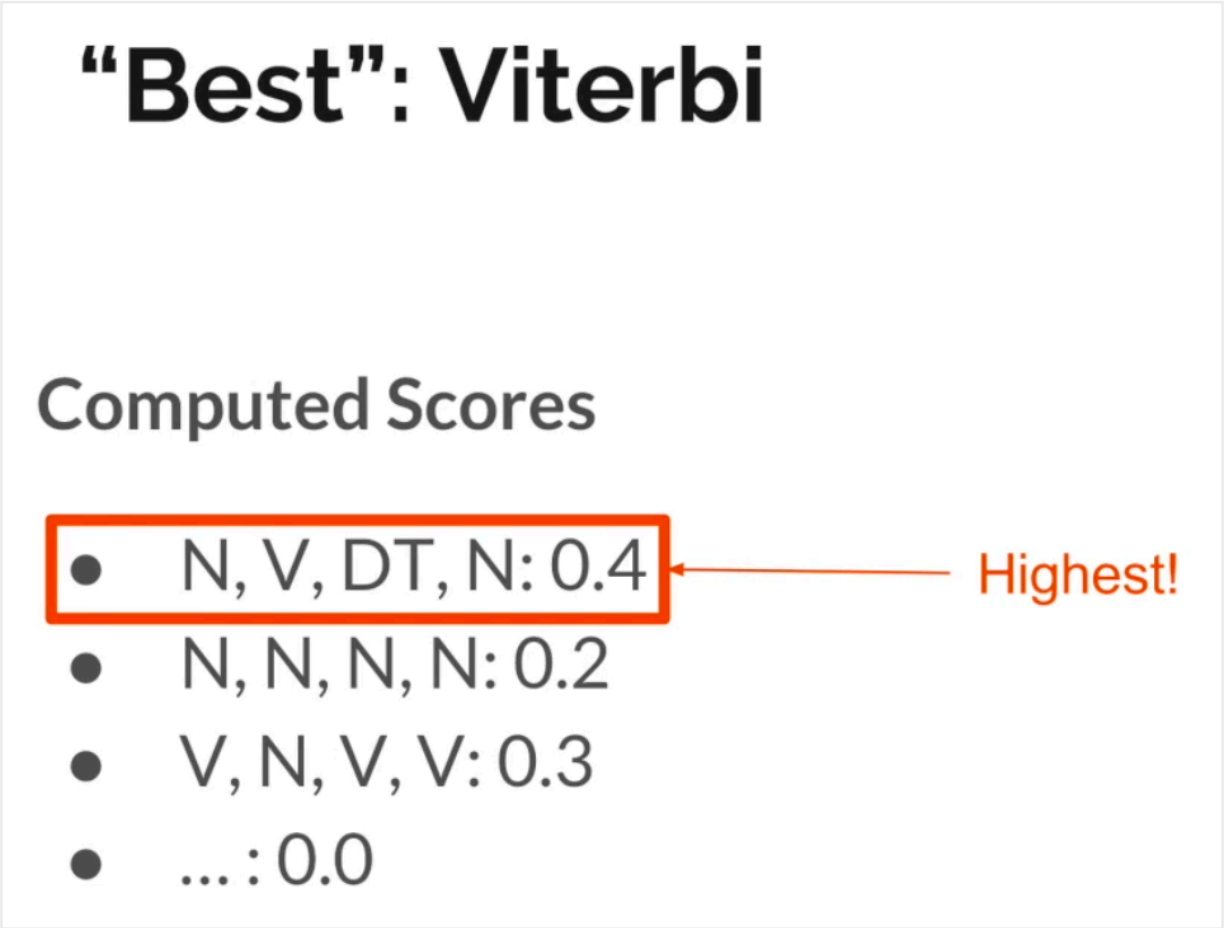
$$P(y_0, y_1, y_2, y_3, x_0, x_1, x_2, x_3) = [0.5 \times 0.25] \times [1.0 \times 1.0] \times [1.0 \times 0.75] \times [1.0 \times 1.0]$$
$$= 0.875$$

B->A->B->A ends up with the highest probability in this case.

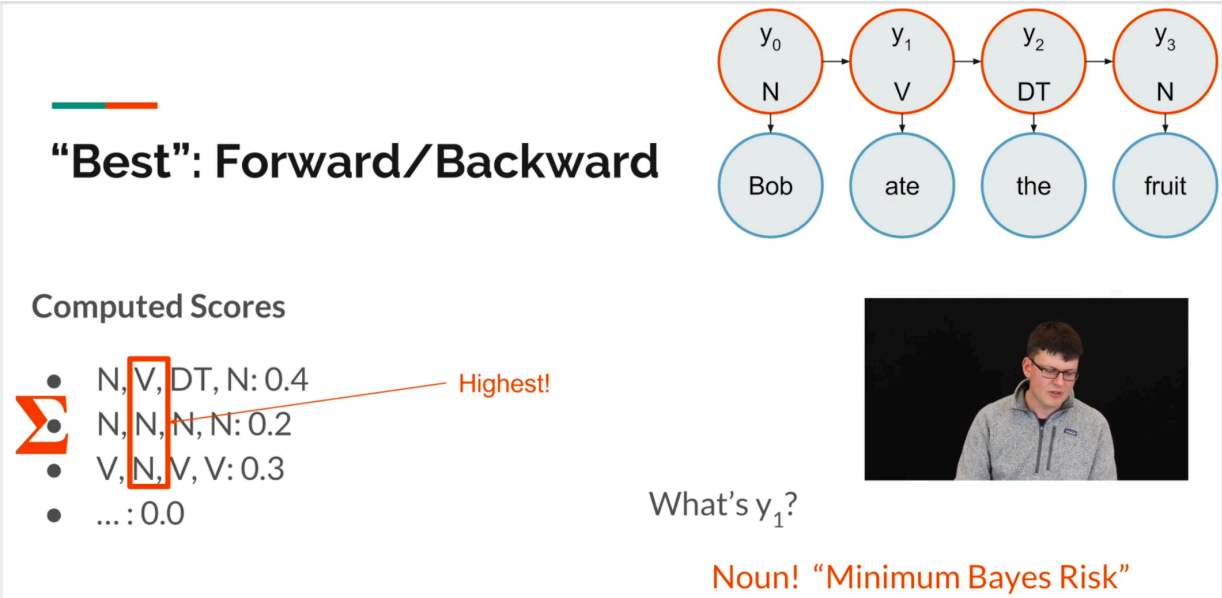


This gets expensive though because the number of possible hidden states goes up, we will require a lot of calculations.

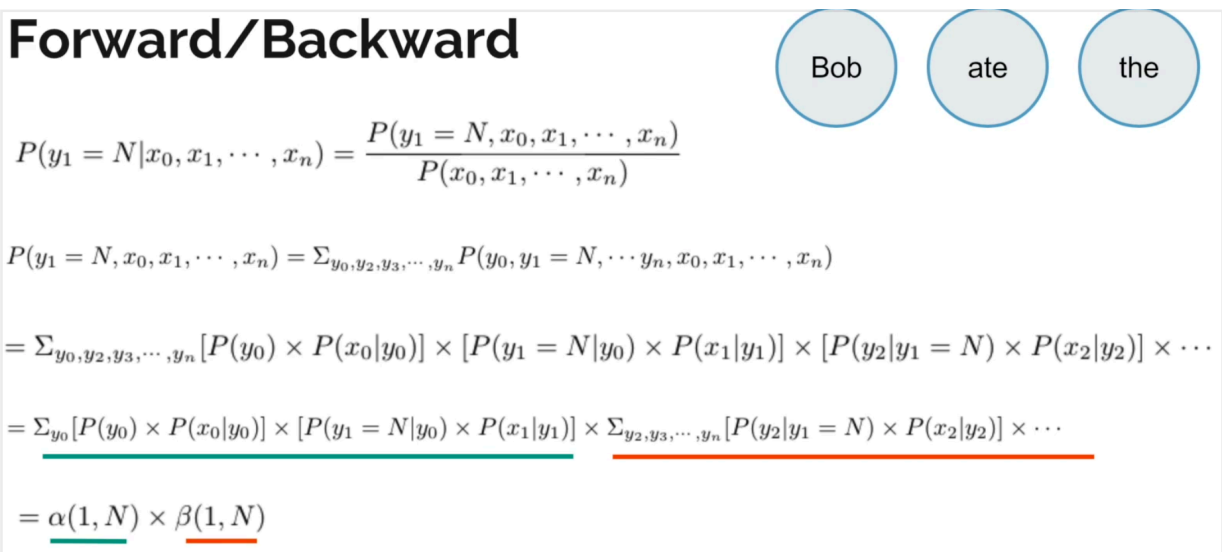
Also, the idea of “Best” is a bit flawed:



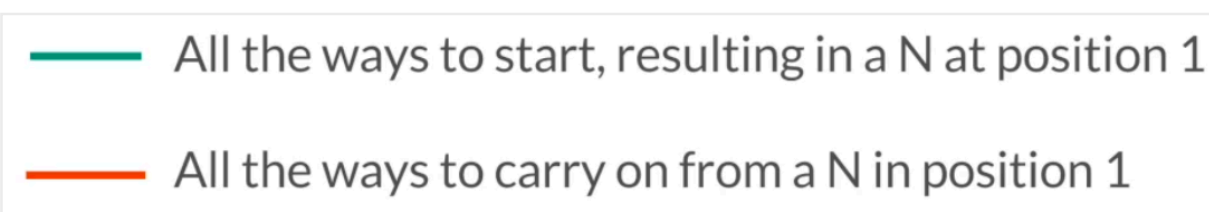
What if we ask, “what is y1?”. If we only look at y1, Noun appears twice and the aggregate score is 0.5... is that the best choice? Or is it still 0.4? In practice we would go with N because the probability is now 0.5. (Forward-backward solutions aka minimum Bayes risk solution)



How to calculate Forward/Backward scores?



Observations:



What about the score that position 2 is a Noun?

Position 1

$$= \Sigma_{y_0}[P(y_0) \times P(x_0|y_0)] \times [P(y_1 = N|y_0) \times P(x_1|y_1)] \times \Sigma_{y_2,y_3,\cdots,y_n}[P(y_2|y_1 = N) \times P(x_2|y_2)] \times \cdots$$

$$= \alpha(1, N) \times \beta(1, N)$$

Position 2

$$= \Sigma_{y_0,y_1}[P(y_0) \times P(x_0|y_0)] \times [P(y_1|y_0) \times P(x_1|y_1)] \times [P(y_2 = N|y_1) \times P(x_2|y_2 = N)]$$

$$\times \Sigma_{y_3,y_4,\cdots,y_n}[P(y_3|y_2 = N) \times P(x_3|y_3)] \times \cdots$$

$$= \alpha(2, N) \times \beta(2, N)$$

Incremental Construction

Forward

$$\alpha(2, N) = \Sigma_{y_1} P(y_2 = N|y_0) \times P(x_2|y_2 = N) \times \alpha(1, y_1)$$

Backward

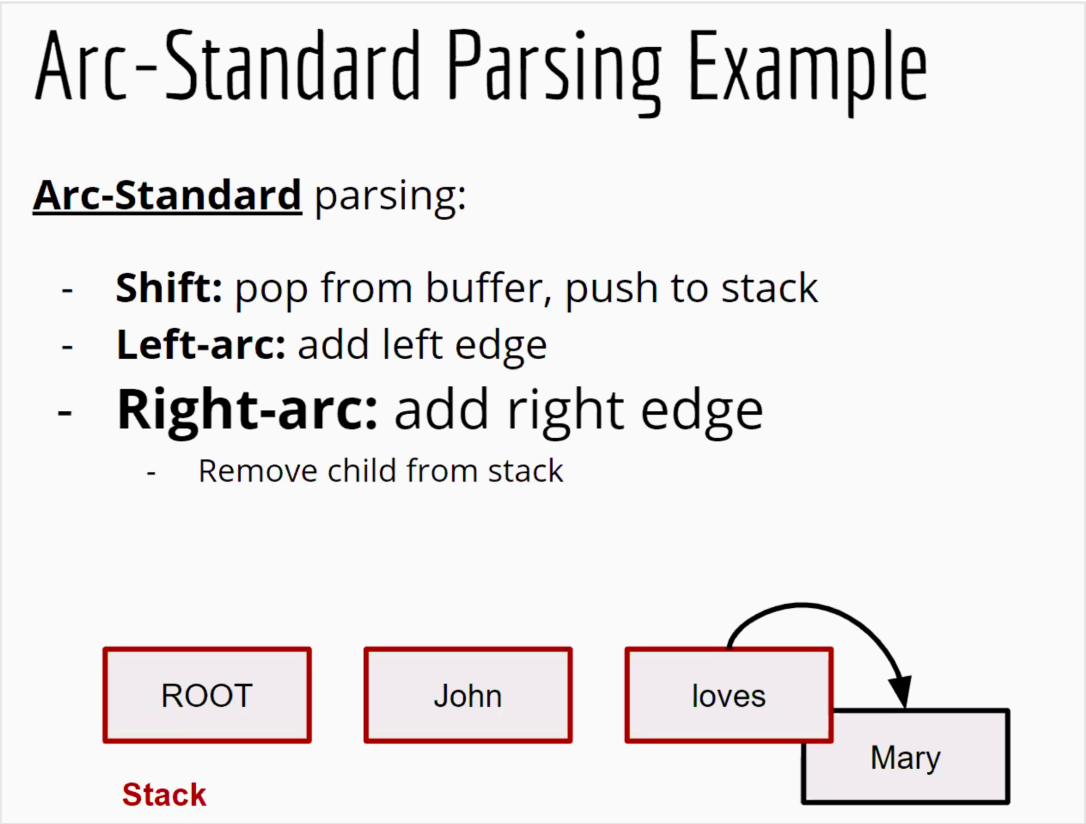
$$\beta(1, N) = \Sigma_{y_2} P(y_2|y_1 = N) \times P(x_2|y_2) \times \beta(2, y_2)$$

You can reuse results and get alpha(2,N) trivially from alpha(1) and Beta(1) from Beta(2) trivially. You run a forward pass to calculate the alphas and a backward pass to calculate the betas simultaneously then you combine the alphas and the betas to get the probabilities.

Viterbi is the same as just doing a forward pass, but instead of summing them up, you take the maximum.

Example: https://www.youtube.com/watch?v=PAJpFypixh8&list=PLh1VT8Lm3apRPGB1uKZsMyMt1DmUf_ItE&index=7

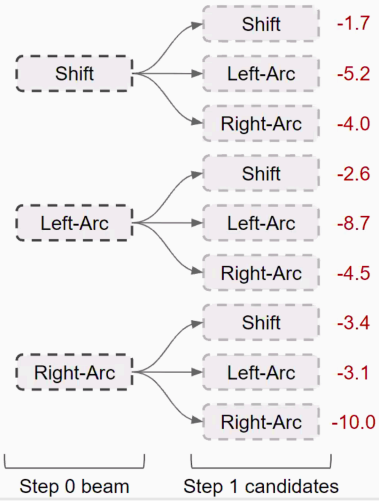
Week 12



O(n) because you touch each word only twice.

If you mess up you can't go back. It's a greedy algorithm. (Use beam search to counter the mistakes)

Inference: Beam Search



Beam search: keep "beam" of k top-scoring candidates (up to complete trees)

- Similar to Viterbi, but can't make Markov assumption
- Solution: keep top k *histories*

1. Generate (k x num_actions) candidates
2. Sort and prune to top k

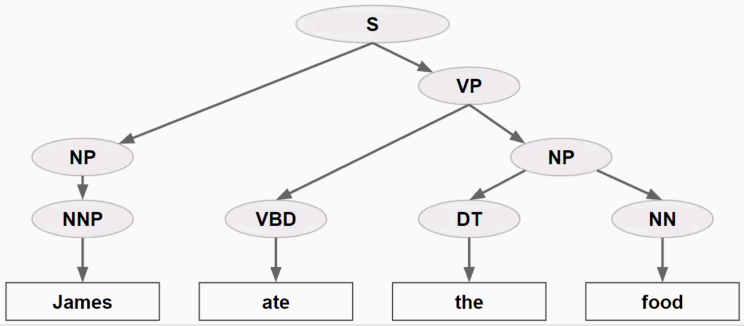
Beam Search Key point: keep top n options, as it changes over time.

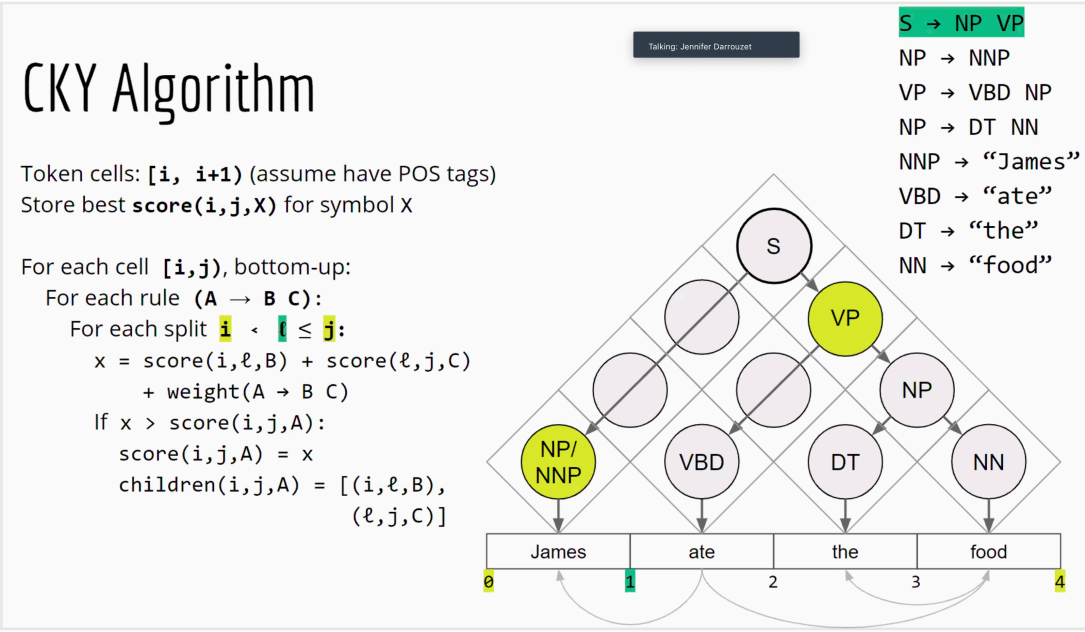
Context-Free Grammars

Grammar rules: `A → B C, A → "a"`

- Apply by recursive expansion of symbols (nonterminals)

```
S → NP VP
NP → NNP
VP → VBD NP
NP → DT NN
NNP → "James"
VBD → "ate"
DT → "the"
NN → "food"
```





O(n^3) algorithm
Order matters! VP -> VBD NP, not VP -> NP VBD