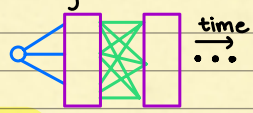
Most word types only have one POS tag; but a large fraction of word tokens are ambiguous. BIO encoding: Beginning, Inside, Outside. HMM is a gen. model

HMM: $P(x_1, x_2, \dots, x_n, y_1, y_2, \dots, y_n) = P(y_1)\prod P(y_{i+1}|y_i)\prod P(x_i|y_i) = \pi_{y_1}\prod A_{y_i y_{i+1}}\prod B_{y_i x_i}$ where π ($K\times 1$ initial prob); A ($K\times K$, transition); B ($K\times M$ emmission)

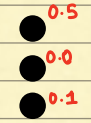
Trellis diagram:



Initial Probability to enter state

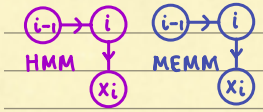
Each set of states will emit our desired state with a probability

Transitions between each state



Thus, we want to get highest weighted path through a trellis diagram will get us the best POS. The Viterbi algorithm does this by using DP with a run-time of $O(n \cdot k^2)$ since each layer has k^2 time ane we do this

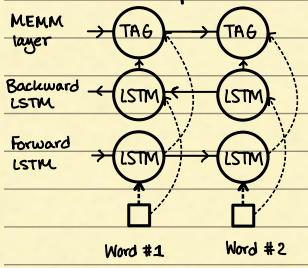
Viterbi: requires a backpointer & $T[j][i] = P(w_i|t^j)\,\text{Max}_k(T[k][i-1]P(t^j|t^k)) \equiv$ Generating curr observation $\cdot$ (Best score given all possible k-values). Learning in HMM can be conducted in both an supervised & unsupervised attention. $\pi_s = \frac{\text{count}(\text{start}\to s)}{n}$; $A_{s',s} = \frac{\text{count}(s\to s')}{\sum \text{count}(s\to s'')}$; $B_{s,x} = \frac{\text{count}(s\to x)}{\text{count}(s)}$; The supervised training process is count and divide.



MEMM is a discriminative model since it gets $P(y|x)$ directly and classifies a decision boundary. $\Rightarrow P(y_i|y_{i-1}, y_{i-2}, \dots, x_i, x_{i-1}, \dots) \equiv$ Markov Assumption $\equiv P(y_i|y_{i-1}, x_i) \therefore P(y|x) = \prod P(y_i|y_{i-1}, x_i)$

To model $P(y|x)$ we use a log-linear model and use features: $P(y_i|y_{i-1}, x) \propto \exp(w^T \phi(x, i, y_i, y_{i-1}))$ we can train this using SGD. We use the viterbi algorithm still for inference time, we gain rich feature representation and this may generalize better, note we cannot do this for generative models.

HMM Pros: unsupervised learning; cons: doesn't allow use of features to represent inputs. MEMM pros: allows features to be used, cons: no unsupervised learning $\Rightarrow$ label bias problem.



Named Entity Recognition (NER): PER (Person), LOC (LOCATION), ORG (Orginization); GPE (geo-political, dates, times, prices) and we can apply the BIO The limitation of an RNN for this is that the output of a tag will not influence the next tag. Thus we combine MEMM and transformers. We model it using Softmax ($W^P(h_t : P_{t-1})$) where we combine the outputs of the forward and backward LSTMs into p_t. This provides two benefits: we directly model output dependencies by MEMMs and we have powerful automatic feature learning using LSTMs. We can also jointly train model parameters. $h_t : P_{t-1}$ means concatenation with previous tag where h_t is the combination of the forward and backward pass (this combination can occur in multiple ways. We apply softmax to both regular MEMM & MEMM + LSTMs: $\sigma(z)_i = e^{z_i} / \sum e^{z_j}$

Precision: of the answers I gave what percent were correct? Recall: what percent of the correct answers did I give? $F_1 = 2/(1/\text{Precision} + 1/\text{Recall})$. The goal of parsing is too construct a syntax tree, and we evaluate the closeness of trees based on their f-1 score. Given a grammar we can get parse tree of a sentence using the CYK algorithm. It has a run-time of $O(n^3)$ and space complexity of $O(n^2)$. However it requires the grammar to be in Chomsky Normal Form where $X\to YZ \,||\, X\to x$. We can also add a weight to each rule and add these weights. We use negative log-likelyhood so the smaller the weight the better. We use the Penn-tree bank to train and evaluate. Some trees are non-binary so we take them and convert it to a binary tree and then run the CYK algo. We train this algorithm using count and divide: $P(X\to YZ|X)$ and $P(X\to x|X)$. Disadvantages: doesn't model languages with discontinuous information. In CFG form, lexically grounded selection preference is lost. May contain too much information for downstream applications.

Dependency view: no explicit phrase structure: advantages: closer alignment between analysis of different languages. Natural modeling of relation ships between discontinuous words. Lexically grounded relationships. Disadvantages: some loss of expressivity. Requires notion of head. Used for information extraction. Content versus functional head, we typically choose the content head. Projectivity if every subtree occupies a contiguos span of the sentence. Non-projective if there exists connection that causes a non-contiguos span. Parsing approaches: convert the constituency tree and project it into a dependency tree; however this requires that the grammar rules require the head words to be specified and thus leads to lexicalization being very sparse. This works well for english but not for other languages. Their exist other graph based approaches. Transition based (shif-reduce) is a linear time algorithm, it provides an approximate solution and doesn't handle non-projective cases. Shift reducer parser has three actions: shift (add element to stack), Reduce Left: the top of the stack is head the next is dependent; Reduce Right: top of stack is dependent next is the head. Determine step seq. given a tree (deterministic algorithm) $\Rightarrow$ if you can reduce left do it, if you can reduce right and all dependencies on the top have been added do it, otherwise shift. Training: learn a probabilistic model P(action | state) the state can include anything such as words on a stack, words in the buffer.... we can do feature learning using a neural networle. The neuralized version is called Neural Shift-Reduce Parser. Shift reducy is a greedy based approach, we can use beam search to not be as greedy. This is the Arc-standard approach there is an Arc-Eager approach & the neural approach is using a stack pointer networle.

Levels of language - phonetics: what words are we dealing with; Semantics: what is the meaning?; Syntax: what words modify each other? Pragmatics: how you should react. Homonym: words that share a form but have distinct meanings. Polysemy: word has related meanings. Hyponym: denotes a sub class (hypernym); Meronym: part-whole relation (holonym). Bag of Words is a classification algorithm. We assume the order of words doesn't matter $\Rightarrow k(V^d-1) \Rightarrow$ Naive Bayes: words are independent conditioned on their class $\Rightarrow k(V-1)$

$\arg\max_y P(Y=y|X=x) = \arg\max_y \frac{P(X=x|Y=y)P(y=y)}{P(X=x)} \rightarrow$ Naive-Bayes $\rightarrow \arg\max_y P(Y=y)\prod_j P(X_j|y)$ we get $P(X_j|y)$ by count and divide

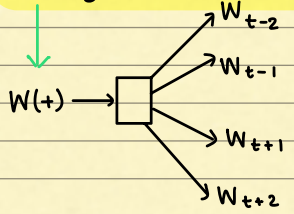
↳ Naive Bayes is a Generative Model.

A lemma can have multiple senses, a sense is a discrete representation of exactly one meaning. path-len = 1 + num edges.

$\text{simpath}(c_1,c_2) = \frac{1}{\text{pathlen}(c_1,c_2)}$ $\text{word-sim}(w_1,w_2) = \max \text{simpath}(c_1,c_2)$ where $c_1 \in \text{senses}(w_1)$ & $c_2 \in \text{senses}(w_2)$. Based on the tree structure

$P(x,y)$ = sum of $M[x][y]$ / all entries; $P(x)$ = sum of row / all entries; $P(y)$ = sum of col / all entries. $PMI = \log_2 P(x,y)/P(x)P(y)$; $PPMI = \max(PMI, 0)$

Skip-gram (Word2Vec) given a word predict it's neighbors. This attempts to learn the word embedings. Every word has 2 vectors v_w when it is the center word and u_w when it the context (input/output embeddings).



$J(\theta) = \frac{-1}{T}\sum_{i=1}^{T}\sum_{-m..m}\log(P(w_{t+j}|w_t)) \rightarrow P(w_{t+j}|w_t) = \frac{\exp(u_{w_{t+j}}\cdot v_{w_t})}{\sum_{w'\in V}\exp(u_{w'}\cdot v_{w_t})} \Leftrightarrow W^T W$ | $y = \text{softmax}(X \cdot W_{IN} W_{OUT}^T)$ at pred time (1×V, 1×V, V×N, N×V)

Cosine Similarity: $\cos(v,w) = \frac{v\cdot w}{|v|\cdot|w|} = \frac{\sum v_i w_i}{\sqrt{\sum v_i^2}\cdot\sqrt{\sum w_i^2}}$; LSA we apply SVD to a matrix to find latent components and then project into a k-dim subspace. Note that cosine simalirty may not work well if basis are not orthogonal.

The n-gram LLM models $p(x_{t+n}|x_{\{t:t+n-1\}})$. To capture this behaviour we pad with n-1 <bos> and one <eos>. For training we do count and divide for MLE estimation. However, if our data is very sparse without smoothing we may get a lot of zero probabilities. The n-gram assumption states the probability only depends on only a fixed history. Word type $\Rightarrow$ distinct item; token $\Rightarrow$ occurrance of that type. Extrinsic evaluation: measure performance on a downstream application. Intrinsic Evaluation: design a measure that is inherent to the current task. Perplexity: how surprised is our model $\Rightarrow$ lower perplexity $\Rightarrow$ better model performance. As we increase n $\rightarrow$ the possible set decreases in size

$P_{MLE}(w_i|w_{i-2}w_{i-1}) = \frac{\text{count}(w_{i-2}w_{i-1}w_i)}{\text{count}(w_{i-2}w_{i-1})}$ cross-entropy: $\sum \frac{\log_2(P(w_i|\dots)}{N}$; Perplexity: $2^{\text{cross-entropy}}$

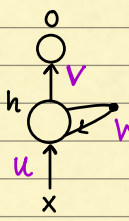
We can use smoothing to deal with unseen circumstances. Smoothing increases bias but decreases variance. Smoothing should not break the law of total probability. Add one smoothing: $+1/+|V|$ this means we are more likely to see novel events. λ-smoothing: $+\lambda/+\lambda|V|$ we discover the optimal λ using using k-fold cross validation (we can do leave-one-out estimation by removing one set of sentence counts)

Backoff Smoothing: $P(z|xy) = u_3P(z|xy) + u_2P(z|y) + u_1P(z)$ where $u_1+u_2+u_3=1$. Backoff smoothing reduces higher order n-grams to lower order n-grams when the higher order n-gram is not present.

Log-Linear model: assigns joint probabilities based on a score: $Pr(x,y) = \exp(\theta\cdot f(x,y))/\sum\theta\cdot f(x',y')$ this allows us to learn weights of features.

Feedforward Model: One-hot encoding $\rightarrow$ Word2Vec $\rightarrow$ Word Embedding $\rightarrow$ n-gram like concatenation $\rightarrow$ neural network weights + bias $\rightarrow$ softmax output

Here we can learn both the model parameters and word embeddings using backprop. We solve both sparsity & word similarity but we still make an n-gram independence assumption and can only deal with finite length $\Rightarrow$ (Still assumes some words are independent of each other since we concatenate words together. RNNs attempt to use sequential information and assume dependence with all words. It performs the same task at every step, inputs to the next step will depend on the output of the previous step. U,V,W are reused for the RNN and across the time steps, $x^{(t)}$ is the word embedding of the t^{th} word which can be received from the embedding matrix E. At each time step the output will be the probability of seeing that word given the entire context. In theory this model will encapsulate an infinite length sequence. However, in practice RNNs are unable to capture long-term dependencies due to the vanishing or exploding Gradient Problem



$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)}$

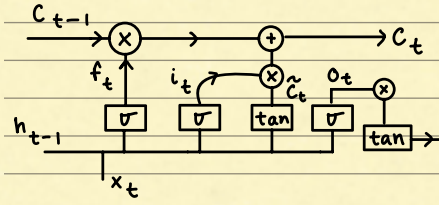
$h^{(t)} = \tanh(a^{(t)})$

$o^{(t)} = c + Vh^{(t)}$

$\hat{y}^{(t)} = \text{softmax}(o^{(t)})$

$P_{RNN}(w_i|w_{i-1}, s_{i-2}) \Rightarrow$ Model the probability of the next word at each step.

Long-Short-Term-Memory (LSTM): are a type of RNN designed to capture long term dependencies, by turning cascading RNN mults to addition



$$i_t = \sigma(x_t U^i + h_{t-1} W^i)$$
$$f_t = \sigma(x_t U^f + h_{t-1} W^f)$$
$$o_t = \sigma(x_t U^o + h_{t-1} W^o)$$
$$\tilde{C}_t = \tanh(x_t U^g + h_{t-1} W^g)$$
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

$h_t = \tanh(C_t) * o_t$

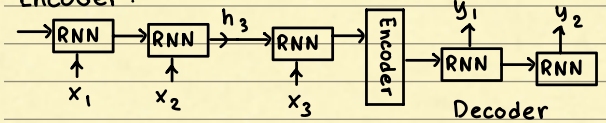
where * denotes element wise mult and sigmoid is applied per ele

Three gates: input, output, forget; these control information flow and are to be learned. Parameters θ in RNNs:
Word Embedding matrix: R
RNN weights: W, U, b, V, c
LSTM weights: $W^i, W^f, W^o, W^g, U^i, U^f, U^o, U^g$ (optional biases)

One issue with RNNs / LSTMs is that we cannot parrallelize computation; the vanishing gradient problem is still an issue for LSTMs

Sequence-to-Sequence Model: A sequence $x_1, x_2, \dots, x_n$ goes in and a different sequence $y_1, y_2, \dots, y_m$ comes out. RNNs were initially used as a base

Encoder:



The last hidden state of the encoder will be used as input to our decoder to produce an output. However this assumes the last hidden state encapsulates *all* the information but this does not happen in practice. Instead we know do a weighted sum of *all* hidden states of the encoder. We use softmax to calculate the score of the inner product between encoder and decoder hidden states, the more similar they are, the higher the attention score. Self attention allows allows weighting other positions in the input sequence for clues that help

Self Attention: the first word will query each other word and based on their keys decide it's attention. If an input sequence has length L then each word does L queries so across an entire sentence L^2 queries will be made. We then divide by $\sqrt{d_k}$ to normalize w.r.t to the query / key vector dimension & then apply softmax. Then we multiply each value vector by the attention score to get a weighted sum. This then produces one hidden state output for the current word just like an RNN. The query, key, value matricies will *all* have the same dimension. The dimension of key, value, query will be $d \times h$, where d is the embedding dimension and h is the hidden dimension.

$$\underbrace{X}_{L\times d} \times \underbrace{W^Q}_{d\times h} = \underbrace{Q}_{L\times h}$$; Same for key & value $\Rightarrow$ $\text{softmax}\left(\frac{Q K^T}{\sqrt{d_k}}\right) V = Z$ (V: $L\times L$, Z: $L\times h$) | each row is the representation of each word

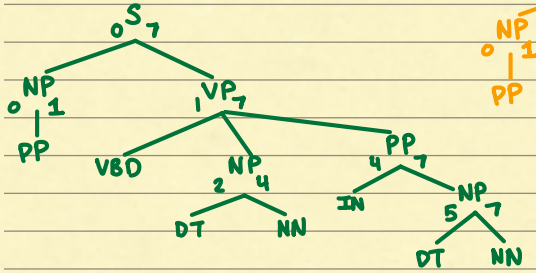
Note that here $d_k = h$; And we apply the softmax function **row-wise** where each column are the attention scores

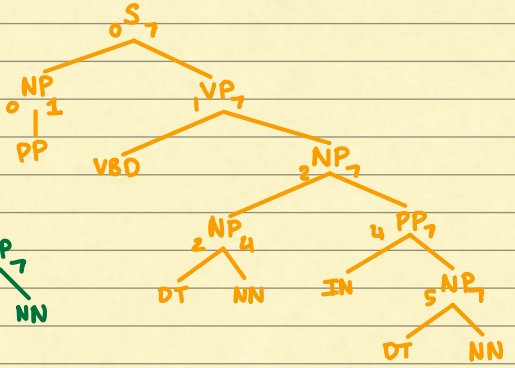
Multi-head Attention: we maintain multiple Q, K, V matricies; so instead of just z_0 we have $z_1, \dots, z_8$ & not just Q^0, K^0, V^0 we will have $Q_0, \dots, Q_n, K_0, \dots, K_n, V_0, \dots, V_n$. How do we combine multiple values we have another learnable weight matrix to weight all the value matricies. This matrix W_0 will have a dimension $k \cdot h \times d$ where k is the number of heads, h is the hidden dimension and d is the embedding size. #Params = 3 × (num heads) × (Layer) × (embedding dim) × (hidden dim), doesn't include the W_0 matrix. W_0 or the feed-forward portion can serve as a way to match the shape of the output back to the input. In this vanilla form the transformer does not account for the position between words. Thus we add a positional encoding to each embedding.

$$\vec{P}_t^{(i)} = f(t)^{(i)} = \begin{cases} \sin(w_k \cdot t), & \text{if } i = 2k \\ \cos(w_k \cdot t), & \text{if } i = 2k+1 \end{cases}$$ where t is the position, i is the dimension, d is the vector size.

Residual Connections & Layer Norm: we add input back in at each sub-layer, since there is an issue of information loss → this happens when self attention decides to not attend to itself: $\text{LayerNorm}(X + \text{Sublayer}(x))$ → happens after each self attention layer. We also say $\text{LayerNorm}(v) = \gamma \frac{v-u}{\sigma} + \beta$ to normalize and prevent exploding values. The query is used to compute a computability score of a target element with *all* elements of an input sequence. The decoder also consists of an encoder-decoder attention layer so that we put attention on our input and not only the input being generated by the decoder. An encoder consist of a multi-layer attention that gets parsed to a feed forward network. A decoder block consists of self-attention, encoder-decoder, and feed-forward. A transformer will have 6 encoder and decoder layers. The encoder layer has bi-directional weighting while the decoding layer will need a mask since attention should only be uni-directional.

Parser Evaluation:





(S 0 7)	(S 0 7)
(NP 0 1)	(NP 0 1)
(VP 1 7)	(VP 1 7)
(NP 2 4)	(NP 2 7)
(PP 4 7)	(NP 2 4)
(NP 5 7)	(PP 4 7)
	(NP 5 7)

$R = 6/6$; $P = 6/7$; $F_1 = \frac{2}{1+7/6} = 0.923$

Precision: $\frac{TP}{TP+FP}$ Specificity: $\frac{TN}{TN+FP}$

Recall: $\frac{TP}{TP+FN}$ Sensitivity: $\frac{TP}{TP+FN}$

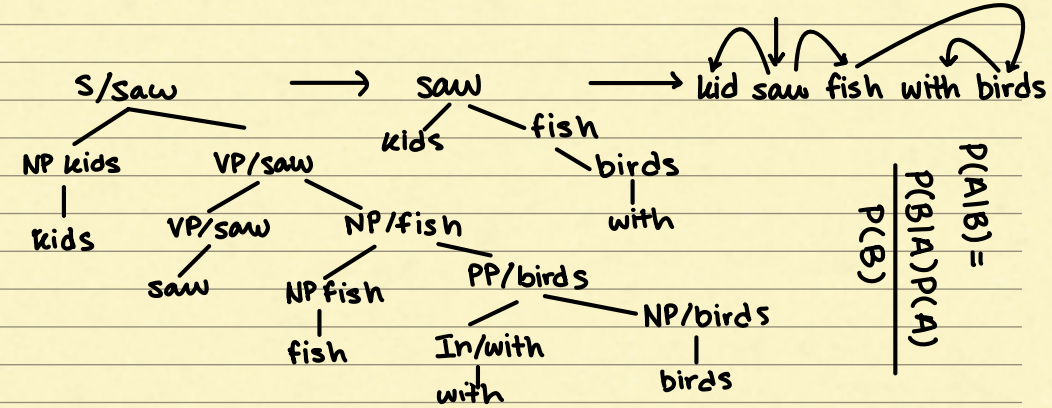
Accuracy: $\frac{TN+TP}{ALL}$

TP	FP
FN	TN

Probabalistic CYK Parser:

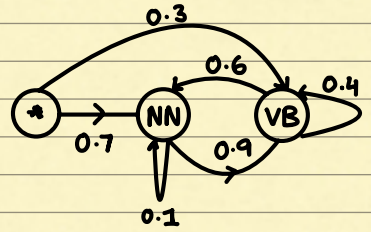
	time 1	flies 2	like 3	an 4	arrow 5
0	NP 3 VSt 3	NP 10 S 8 S 13			NP 24 S 22 S 27 NP 24 S 27 S 22
1		NP 4 VP 4			NP 18 S 21 VP 18
2			P 2 V 5		PP 12 VP 16
3				Det 1	NP 10
4					N 8

1 S → NP VP
6 S → Vst NP
2 S → S PP
1 VP → V NP
2 VP → VP PP
1 NP → Det N
2 NP → NP PP
3 NP → NP NP
0 PP → P NP



Beam Search is a general way to not be a greedy, instead of the best step, we take k best next steps. We only keep the top k of those k^2 steps and thus the runtime for a sequence length is $O(n \cdot k^2)$

	NN	VB
food	4	6
eat	2	8
we	10	2



	food	eat	we
NN	4/16	2/16	10/16
VB	6/16	8/16	2/16

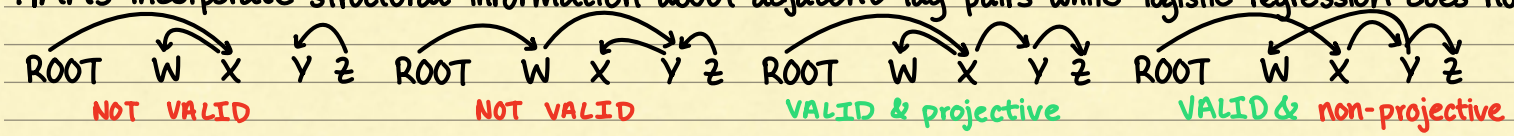
Most Probable tag sequence for we eat:
NN NN: $0.7 \cdot 0.625 \cdot 0.1 \cdot 0.125$
NN VB: $0.7 \cdot 0.625 \cdot 0.9 \cdot 0.5$ → most probable
VB NN: $0.3 \cdot 0.125 \cdot 0.6 \cdot 0.125$
VB VB: $0.3 \cdot 0.125 \cdot 0.4 \cdot 0.5$

Which of the following decoding algorithms for HMM do not gurantee to find the optimal solution? : Greedy Search, A* search
Which statements about HMM is true? HMM can be used for NER, we can perform both unsupervised & supervised learning for model parameters.
Which statements are true about parsing techniques: ambiguity is a major issue for many parsing techniques. Parsing techniques utilize grammars to produce graph structures from sentences

MEMM model: $P(y|x) = \frac{\prod_{t=1}^{n} \exp(f(y_t, y_{t-1}, x_t))}{\prod_{t=1}^{n} \sum_{y_t'=1}^{K} \exp(f(y_t', y_{t-1}, x_t))}$

where we have input sequence: $\{x_1, \dots, x_n\}$ and output sequence $\{y_1, \dots, y_n\}$ where $y_i \in \{1, \dots, K\}$.

HMMS incorporate structural information about adjacent tag pairs while logistic regression does not.



On an arc-standard system with shift reduce dependency parser will require n-shift operations. ELMo uses a bi-directional LSTM model while BERT uses a transformer module as the base model. When a language model is decoding with a beam size k, the time complexity is $O(nk^2)$ except the first step we consider k^2 candidates

Using a neural sequence to sequence model does not gurantee that the predicted sequence of actions always leads to a valid parse tree.

A good positional encoding will output a unique encoding for each time-step; any two time steps should be consistent regardless of input sequence lengths. The algorithm should be able to generalize to longer sentences.

Transition:	V	N
start	0.3	0.7
V	0.4	0.6
N	0.6	0.4

Emission:	V	N
fire	0.2	0.8
fights	0.5	0.5

$$l_i(x_i, y_i, y_{i-1}) = W^T \phi(x_i, y_i, y_{i-1}) \qquad \phi = \begin{bmatrix} e(x_i|y_i) \\ t(y_i|y_{i-1}) \end{bmatrix}$$

$$P(y_i|y_{i-1}, x_i) = \frac{l_i(x_i, y_i, y_{i-1})}{\sum_y l_i(x_i, y, y_{i-1})}$$

$$score_i(s) = \max P(s|y^*, x_i) score_{i-1} y^*$$

HMM: $score_1(V) = P(V|start, fire) = P(fire|V)P(V|start)$
$score_1(N) = P(N|start, fire) = P(fire|N)P(N|start)$

$score_2(V) = P(fights|V) \cdot \max(score_1(V) \cdot P(V|V), score_1(N) \cdot P(V|N))$
$score_2(N) = P(fights|N) \cdot \max(score_1(V) \cdot P(N|V), score_1(N) \cdot P(N|N))$

MEMM: $l_1(fire, V, start) = W[0.2, 0.3] = 0.5$
$l_1(fire, N, start) = W[0.8, 0.7] = 1.5$

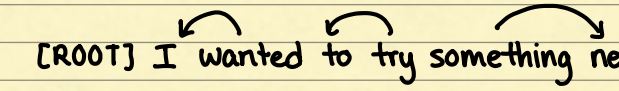
thus: $score_1(V) = P(V|fire, start) = 0.5/(1.5+0.5) = 1/4$
$score_1(N) = P(N|fire, start) = 1.5/2 = 3/4$

$l_2(fight, V, V)$ & $l_2(fight, V, N)$ & $l_2(fight, N, V)$ & $l_2(fight, N, N)$ after calculating these we can compute the following:

$$score_2(V) = \max(P(V|fight, N) score_1(N), P(V|fight, V) score_1(V))$$

$$score_2(N) = \max(P(N|fight, N) score_1(N), P(N|fight, V) score_1(V))$$

Perform: S, S, RL, S, S, RL, S, S, RR
on [ROOT] I wanted to try something new
STACK: [ROOT], wanted, try, something
BUFFER: Empty



HMM, MEMM, RNNs & BERT can be used for NER.
Different pretrained language models can use different tokenizers
We can use cross-validation to pick the best values of lambda

Multi-head attention is designed such that we can give the model chances to learn different aspects of the input sequences
The input & output dimension of the subsequent feed forward network is preferred to be divisible by the number of heads
The dense word vectors in Word2Vec are more expressive than the one hot word vectors. Dense word vectors have better chances to learn rare words rather than those using one-hot word vectors. Dense word vectors are easier to include as features. For dependency trees, every word has at least one parent, every word has exactly one parent, and ROOT has exactly one child. HMM is a generative model & MEMM can use rich features for training.

Attention Example: <Massive> = [1,-1,0], thunderstorm = [1,-1,-1], in = [0,2,0], California = [-1,-3,1]

$W_{q_1}, W_{k_1}, W_{v_1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$ & $W_{Q_2} = W_{k_2} = W_{v_2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -2 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$ thus: $Q = w \cdot W_{Q_1} = \begin{bmatrix} 1 & -1 & 0 \\ 1 & -1 & -1 \\ 0 & 2 & 0 \\ -1 & -3 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & -1 & 0 & 0 \\ 1 & -1 & -1 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$

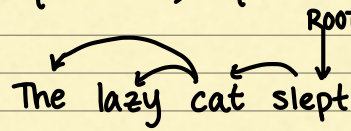
Then $Q \cdot k^T = \begin{bmatrix} 1 & -1 & 0 & 0 \\ 1 & -1 & -1 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & -1 & 0 & 0 \\ 1 & -1 & -1 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}^T = \begin{bmatrix} 2 & 2 & -2 & 0 \\ 2 & 3 & -2 & -1 \\ -2 & -2 & 4 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix}$ divid by $\sqrt{d_k} = \sqrt{4} \Rightarrow \begin{bmatrix} 1 & 1 & -1 & 0 \\ 1 & 1.5 & -1 & 0.5 \\ -1 & -1 & 2 & 0 \\ 0 & -0.5 & 0 & 0.5 \end{bmatrix}$ attention without softmax; thunderstorm is 2nd row.

So in the case of thunderstorm for the words in-terms of how much is placed on each is: thunder > massive > cali > in

There may be a mismatch during pretaining and finetuning for language models leveraging masked learning objectives, while it is not an issue if we use classical left to right language modeling objectives.

HMM assumes that words are independent of each other given the tag sequence.
For an unigram with A unique words then it will have A parameters to store

A is the number of heads, B maximum number of tokens, C input and output dimension of self-attention, D: key vector dimension, E value vector dimension
W_i^K: C×D; W_i^Q: C×D ; W_i^V: C×E ; W^O: AE×C



; SSSLLSLR

For an LSTM to sum inputs over time we set the input gate and forget gate as 1.

Pre-trained LLMs: Byte pair encoding: split words into sequence of known subwords. Uni-directional: decoder only, can't condition on future words (GPT). GPT-3 can use in-context info to learn. Independently bi-directional: train seperate left to right and right to left. (ELMO, LSTM based). GPT is only uni-directional and Elmo's bidirection is shallow. Bidirectional: encoder only pre-training condition on biderectional words (BERT). Masking: mask out k% of words then predict them. Using around 15% masks too much info and thus gets expensive to train.