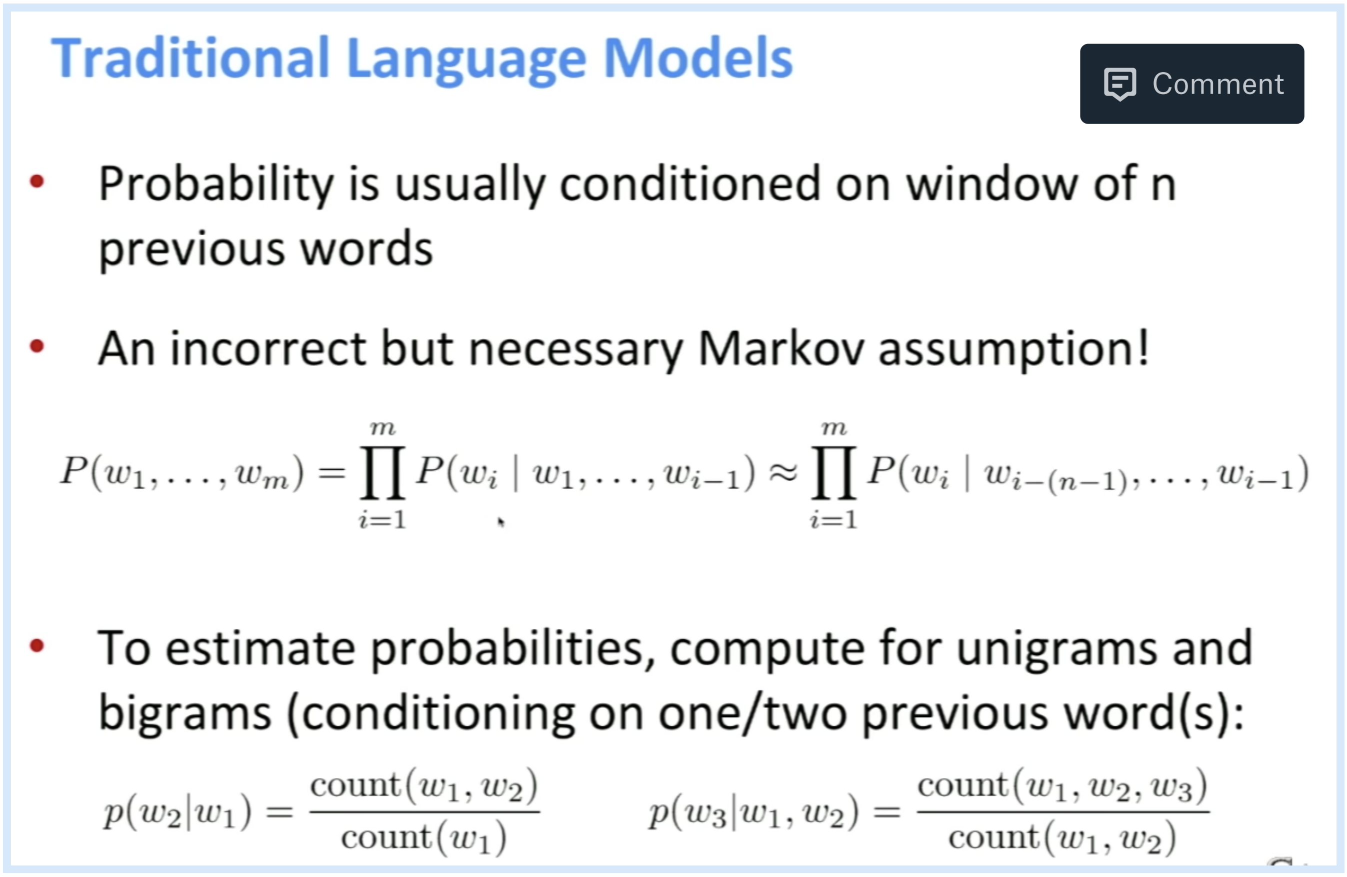
language models

- language models are concerned with the probability of a sequence of words in a language. this can be used for many things:
    - choosing how to phrase a response in a chat bot so that it is grammatically correct.
    - choosing how to generate text in a different language in the machine translation task
    - in speech recognition, to figure out oh the person has said “he is a” until now so the next word is probably X.
    - in next word suggestion in Google keyboard on mobile
    - image captioning → basically anywhere where you need to generate text!
- language models are useful for both word choice as well as word ordering
- a language model computes $P(w_1, w_2, ...w_T)$



Traditional Language Models

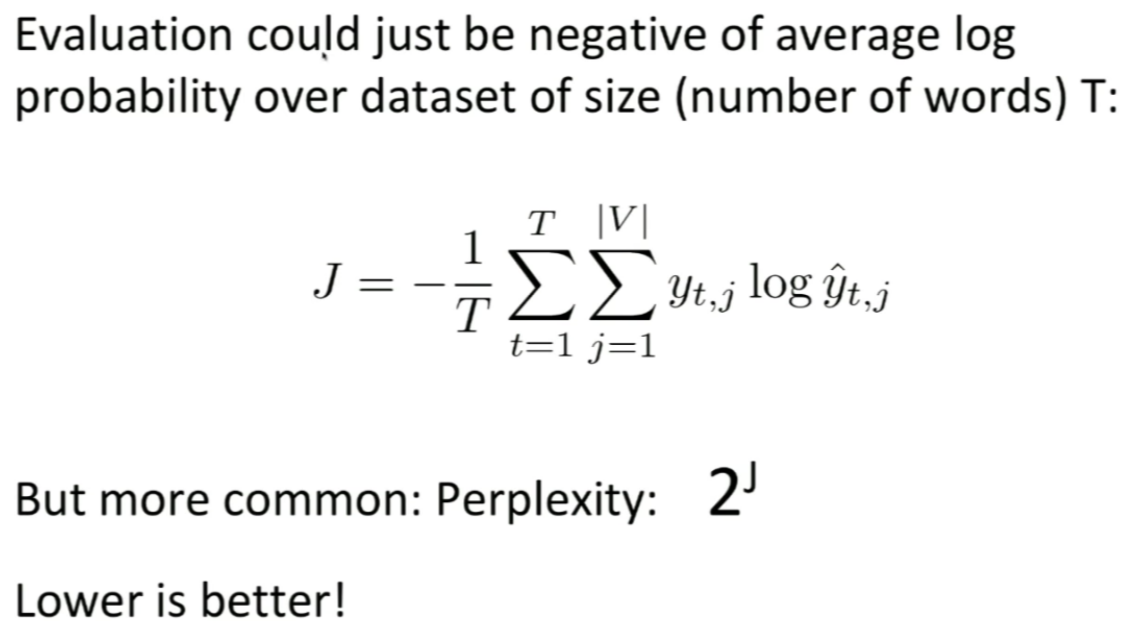
- Probability is usually conditioned on window of n previous words
- An incorrect but necessary Markov assumption!

$$P(w_1, \ldots, w_m) = \prod_{i=1}^{m} P(w_i \mid w_1, \ldots, w_{i-1}) \approx \prod_{i=1}^{m} P(w_i \mid w_{i-(n-1)}, \ldots, w_{i-1})$$

- To estimate probabilities, compute for unigrams and bigrams (conditioning on one/two previous word(s):

$$p(w_2|w_1) = \frac{\text{count}(w_1, w_2)}{\text{count}(w_1)} \qquad p(w_3|w_1, w_2) = \frac{\text{count}(w_1, w_2, w_3)}{\text{count}(w_1, w_2)}$$

- the above is traditional language models that are simple count based models → no ML here.
- As you increase n, the past window in which you look, the number of possible n-grams seen from the corpus explodes and its very hard memory wise. also the more your n the lesser data you have with that for a given sequence.
- the above simple count based-ness and not really looking past a reasonable n due to lack of data is bad! so enter, neural nets, actually recurrent neural nets because these are all inherently sequence based.
- note that as we allowed bigger n, the number of sequences we had could be exponentially more. doesn’t scale well for memory. but with RNNs, the longer the sequence, the memory requirement only increases linearly.
- the loss used is log likelihood loss



Evaluation could just be negative of average log probability over dataset of size (number of words) T:

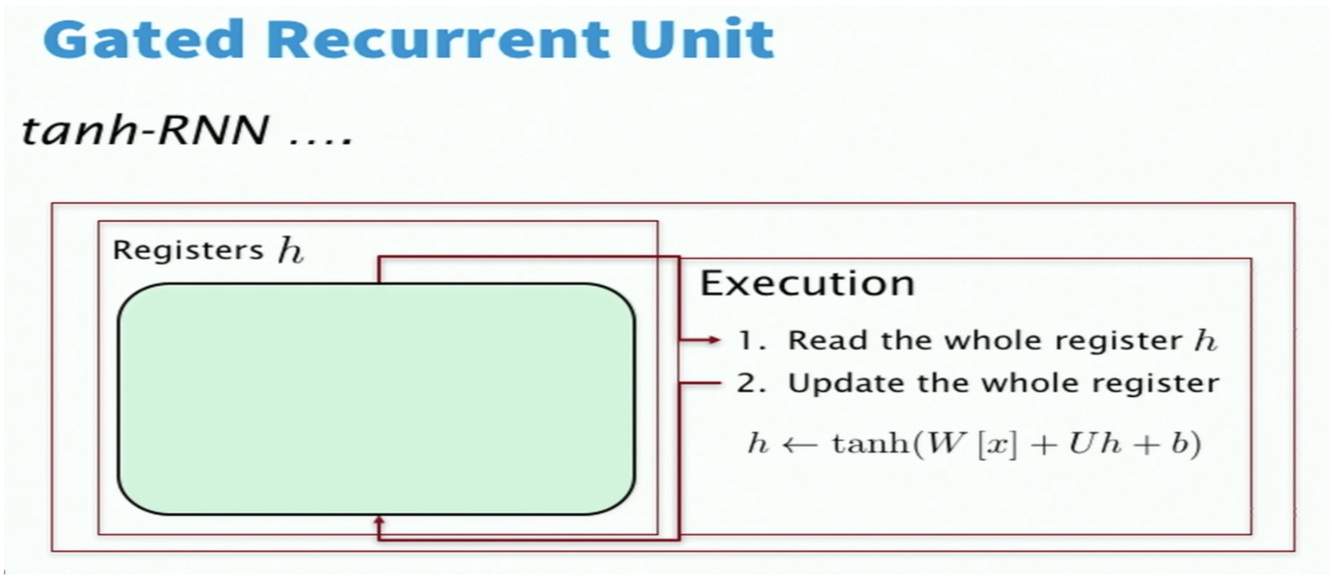
$$J = -\frac{1}{T}\sum_{t=1}^{T}\sum_{j=1}^{|V|} y_{t,j}\log \hat{y}_{t,j}$$

But more common: Perplexity: 2^J

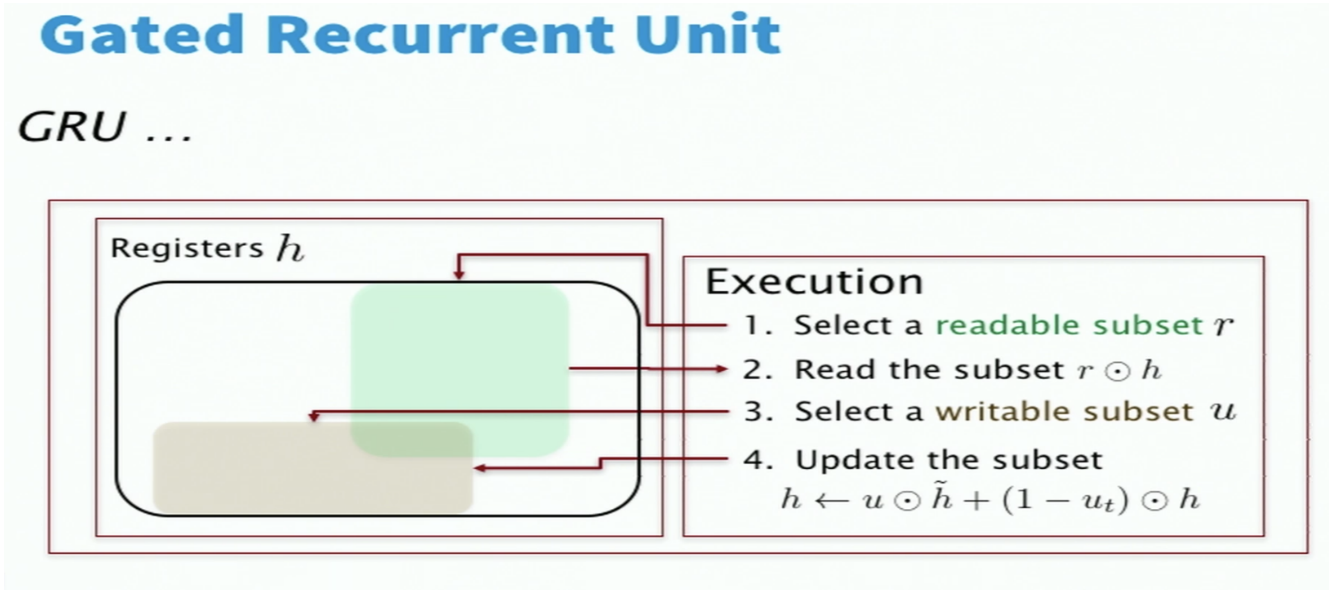
Lower is better!

- the metric used to evaluate performance is called perplexity → the less perplex (less confused) the better.
- RNNs use a unit called LSTM (or a simpler version called GRU). These help to explicitly model long term dependencies, and gives a prior which helps to learn what parts of past to keep and what to throw away. in terms of register like computation, here's a naive $tanh$ based RNN vs a RNN with LSTM or GRU

Practical tips for RNNs

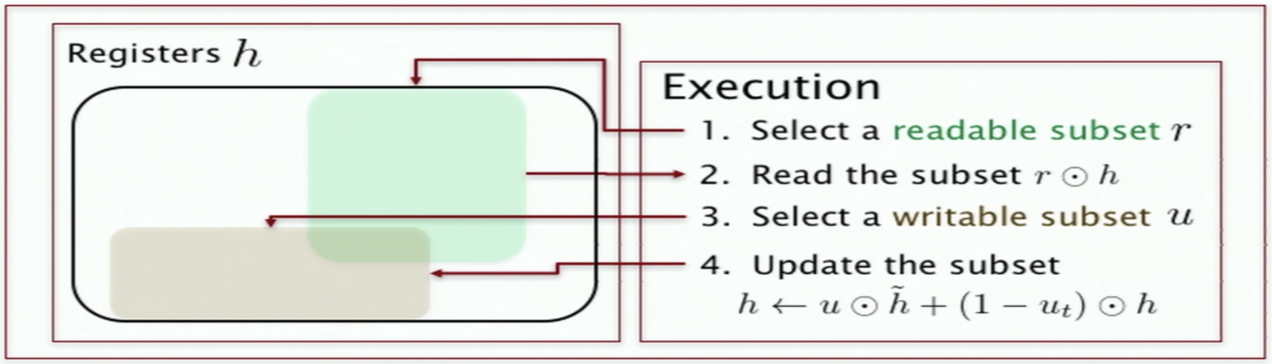

Gated Recurrent Unit
tanh-RNN ....
Registers h
Execution
1. Read the whole register h
2. Update the whole register
$h \leftarrow \tanh(W[x] + Uh + b)$

Vs

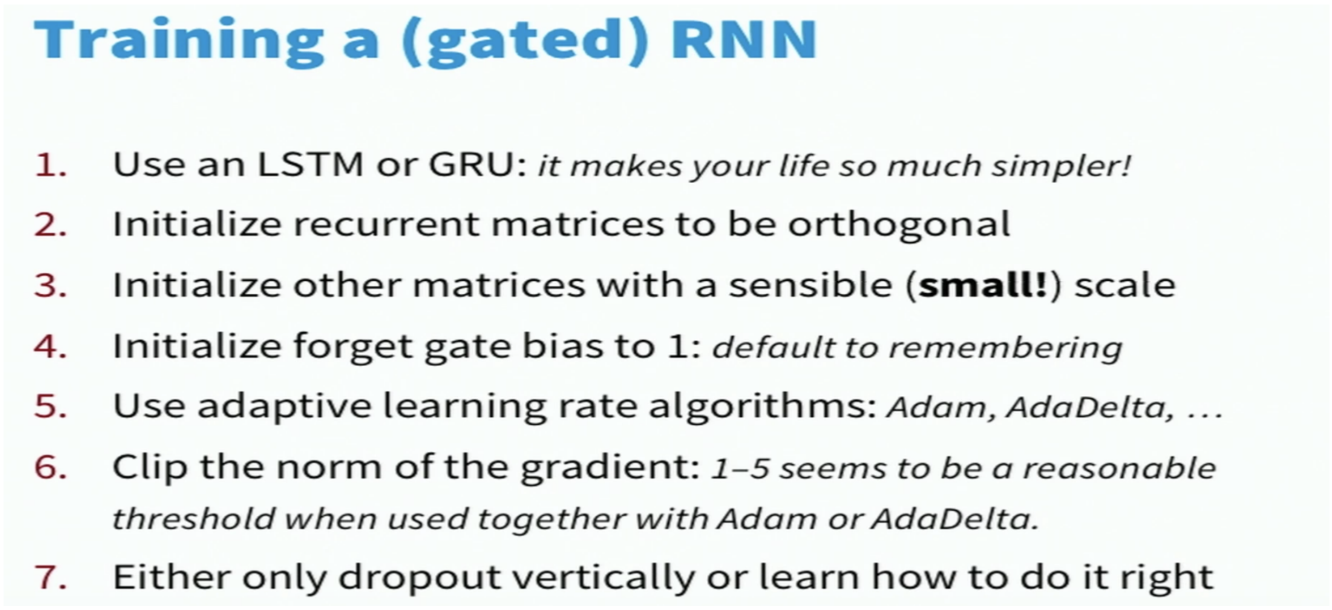


Gated Recurrent Unit

GRU …


Registers h
Execution
1. Select a readable subset r
2. Read the subset $r \odot h$
3. Select a writable subset u
4. Update the subset
$h \leftarrow u \odot \tilde{h} + (1 - u_t) \odot h$

- RNNs can quickly get deep over time → this introduces problems of vanishing and exploding gradients.
- Tips for training RNNs with LSTMs or GRUs



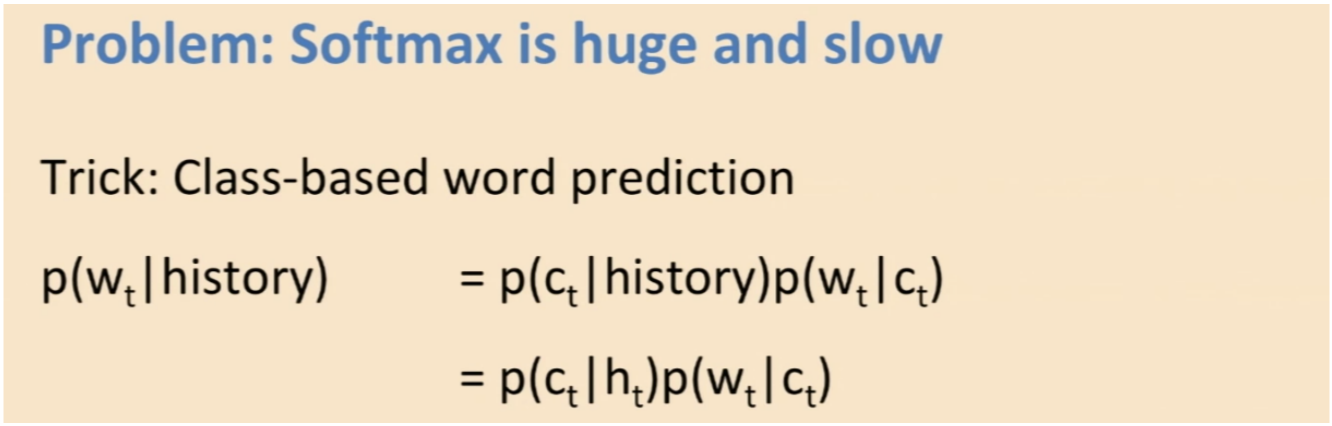
Training a (gated) RNN

1. Use an LSTM or GRU: *it makes your life so much simpler!*
2. Initialize recurrent matrices to be orthogonal
3. Initialize other matrices with a sensible (**small!**) scale
4. Initialize forget gate bias to 1: *default to remembering*
5. Use adaptive learning rate algorithms: *Adam, AdaDelta, …*
6. Clip the norm of the gradient: *1–5 seems to be a reasonable threshold when used together with Adam or AdaDelta.*
7. Either only dropout vertically or learn how to do it right

- The intuition for why it helps recurrent matrices to be orthogonal might be because that way it is forced to look at different components of the thing it multiplies.
- While using dropout for RNNs, you want to use the same dropout matrix for all time steps. If you instead do different dropouts at different time steps, then you will end up dropping out a lot of stuff and not remembering anything.
- Note that while using softmax for classification, if the weight norm of weights for one of the classes is initialized to a high value (if weights of a particular class are big), that might cause problems biasing towards that class.
- Curriculum learning is a good trick while training RNNs. You start with short seq2seq mappings, and gradually increase the length of sequences as training proceeds.
- Teacher training for RNNs feeds in label from previous step instead of predicted label.
- Ensembling + averaging the results gives a few percentage point inputs at the cost of a ton of compute.
- The reason vanishing and exploding gradients is a problem is because with very large or very small numbers our floating point ops in software become really bad estimates and the gradient isn't accurate

anymore. This makes it hard to learn long term dependencies. Information from a few words ago turns out to be often crucial in predicting the next word, so language modelling suffers. In general, long term dependencies suffer with vanishing and exploding gradient problem.

- one practical trick around the exploding gradient problems is to simply clip the magnitude of the gradient, if its getting too large → hopefully we move in a similar direction even with this clipping, even though a small movement because of clipped gradient.
- the above trick doesn't work so well for vanishing gradients because there we would be magnifying the gradient and overshooting or moving in the wrong direction
- One problem is that if our output layer has a output probability for all words, that's a huge output layer and computing it takes a ton of matrix multiplications. one idea to get around this is to predict the class a word belongs to and then predict the word given the class.

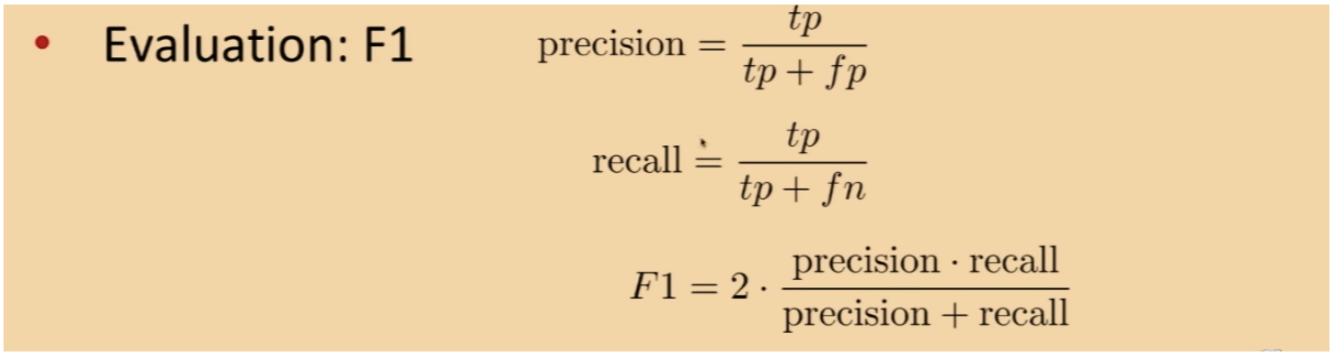


Problem: Softmax is huge and slow

Trick: Class-based word prediction

$$
\begin{aligned}
p(w_t \mid \text{history}) &= p(c_t \mid \text{history})\, p(w_t \mid c_t) \\
&= p(c_t \mid h_t)\, p(w_t \mid c_t)
\end{aligned}
$$

- RNNs give us much better performance on language model problems than simple feedforward nets. Also, on tasks like window classification, NER etc, with RNNs we can look at a larger history of words before coming to a conclusion.
- In general, language is very sequence based and things depend on lot on what has already been said and intuitively, RNNs are able to capture this.
- Bidirectional RNNs help us take advantage of both history and future so we can do for example, a fill in the blank type of task. or think about correcting a past word on Google keyboard based on future words.
- F1 metric is a way to tradeoff precision-recall



- Evaluation: F1

$$\text{precision} = \frac{tp}{tp + fp}$$

$$\text{recall} = \frac{tp}{tp + fn}$$

$$F1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

- Higher value of F1 is better. Max is 1. Lower F1 is bad.