Sequence-to-sequence Models

CIS 530, Computational Linguistics: Spring 2018

John Hewitt & Reno Kriz
University of Pennsylvania

Deep learning!

Now even deeper!

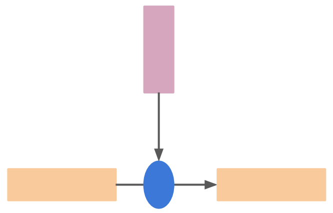
Some concepts drawn a bit transparently from Graham Neubig's excellent
Neural Machine Translation and Sequence-to-sequence Models: A Tutorial
https://arxiv.org/pdf/1703.01619.pdf

We’ve already seen RNNs for language modeling

🟧 The memory vector, or “state” .

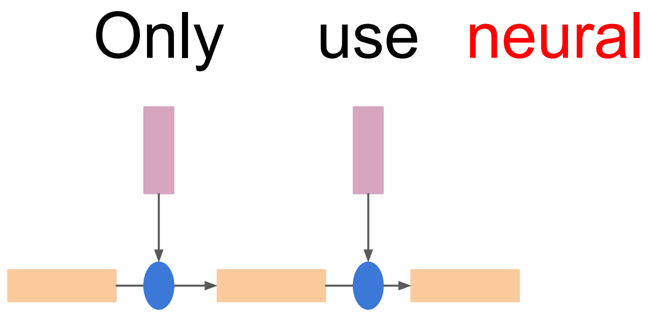
We've already seen RNNs for language modeling

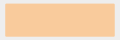
Only use



🟧	The memory vector, or "state" .
▮	The "word vector" representation of the word.
●	The RNN function, which combines the word vector and the previous state to create a new state.

We’ve already seen RNNs for language modeling

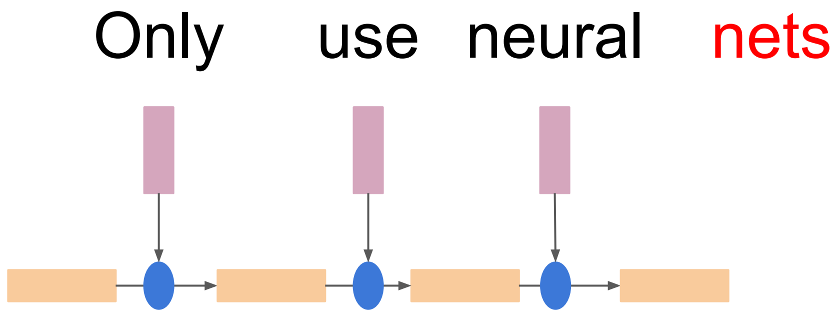


 The memory vector, or “state” .

The “word vector” representation of the word.

The RNN function, which combines the word vector and the previous state to create a new state.

We’ve already seen RNNs for language modeling

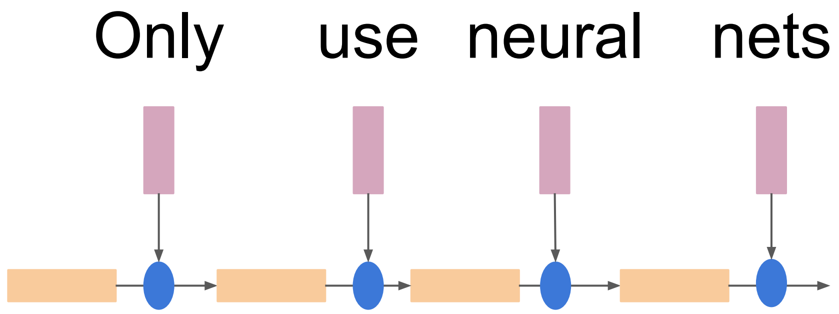


▬ The memory vector, or “state” .

▮ The “word vector” representation of the word.

● The RNN function, which combines the word vector and the previous state to create a new state.

We’ve already seen RNNs for language modeling



▬ The memory vector, or “state” .

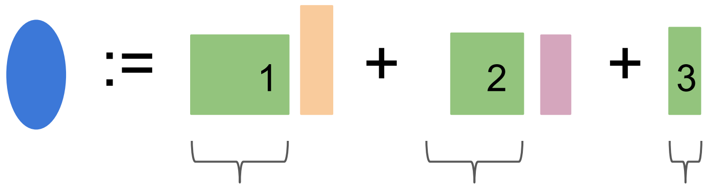
▮ The “word vector” representation of the word.

● The RNN function, which combines the word vector and the previous state to create a new state.

How does the RNN function work?

The RNN function takes the current RNN state and a word vector and produces a subsequent RNN state that "encodes" the sentence *so far*.

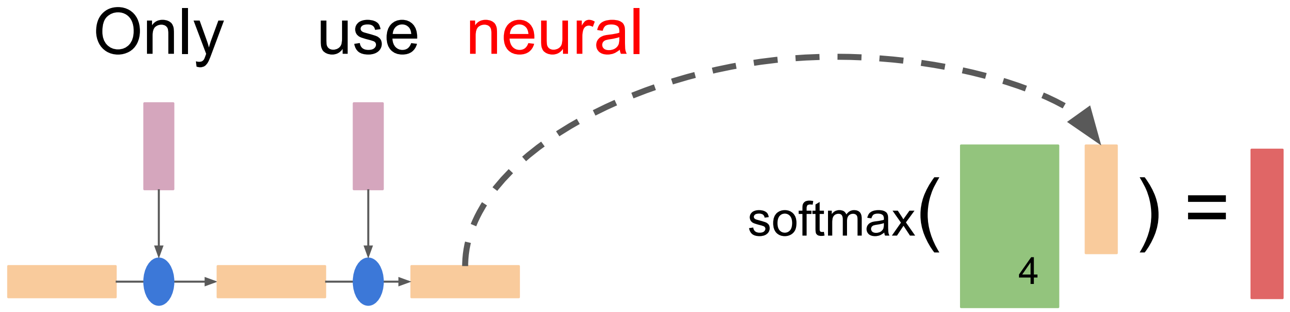
RNN function



🟩 Learned weights representing how to combine past information (the RNN memory) and current information (the new word vector.)

How does the prediction function work?

We've seen how RNNs "encode" word sequences. But how do they produce probability distributions over a vocabulary?

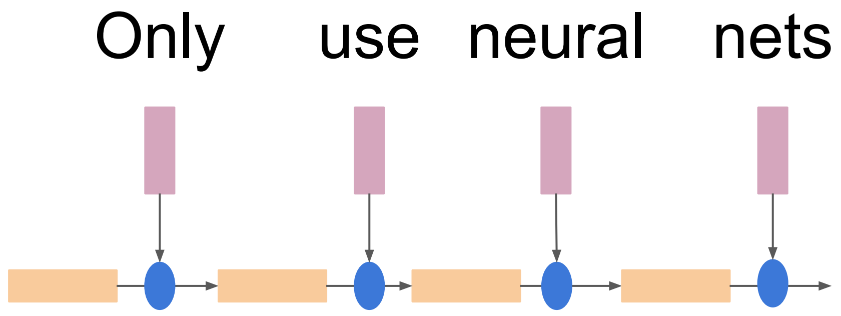


A probability distribution over the vocab, constructed from the RNN memory and 1 last transformation (in green.) The softmax function turns "scores" into a probability distribution.

Want to predict things other than the next word?

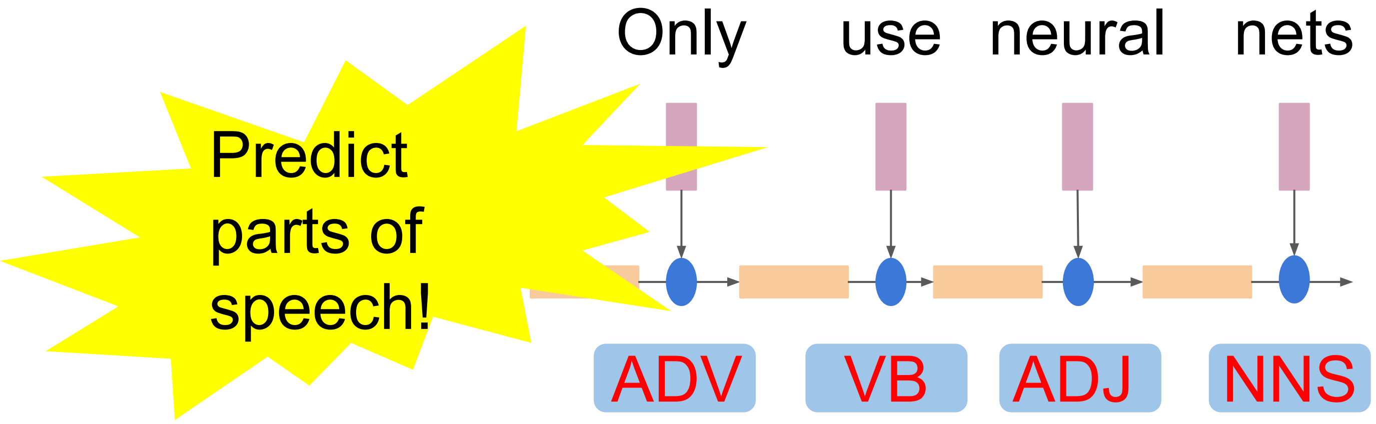
The model architecture (read: “design”) we’ve seen so far is frequently used in tasks other than language modeling, because modeling sequential information is useful in language, apparently.

Here’s our RNN encoder, representing the sentence.



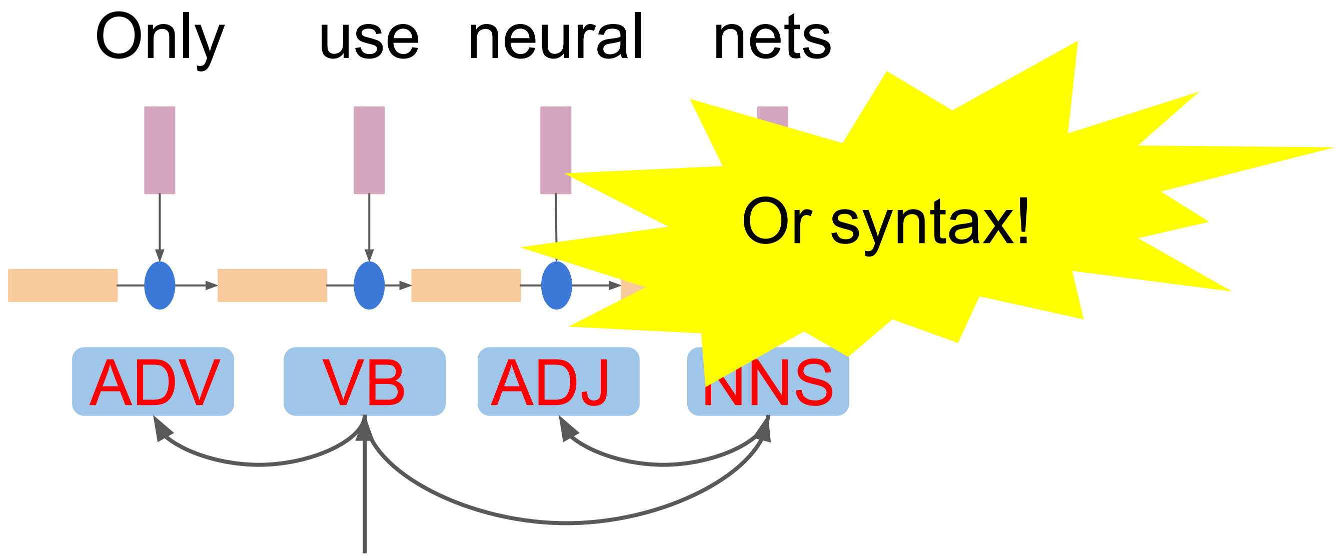
Want to predict things other than the next word?

The model architecture (read: “design”) we’ve seen so far is frequently used in tasks other than language modeling, because modeling sequential information is useful in language, apparently.



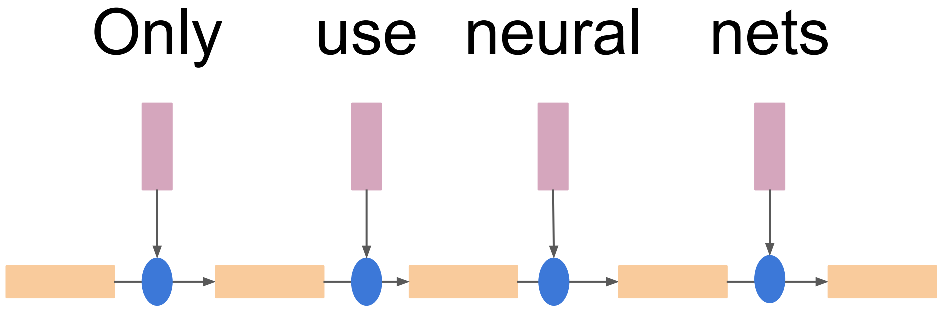
Want to predict things other than the next word?

The model architecture (read: “design”) we’ve seen so far is frequently used in tasks other than language modeling, because modeling sequential information is useful in language, apparently.



General idea: build a representation

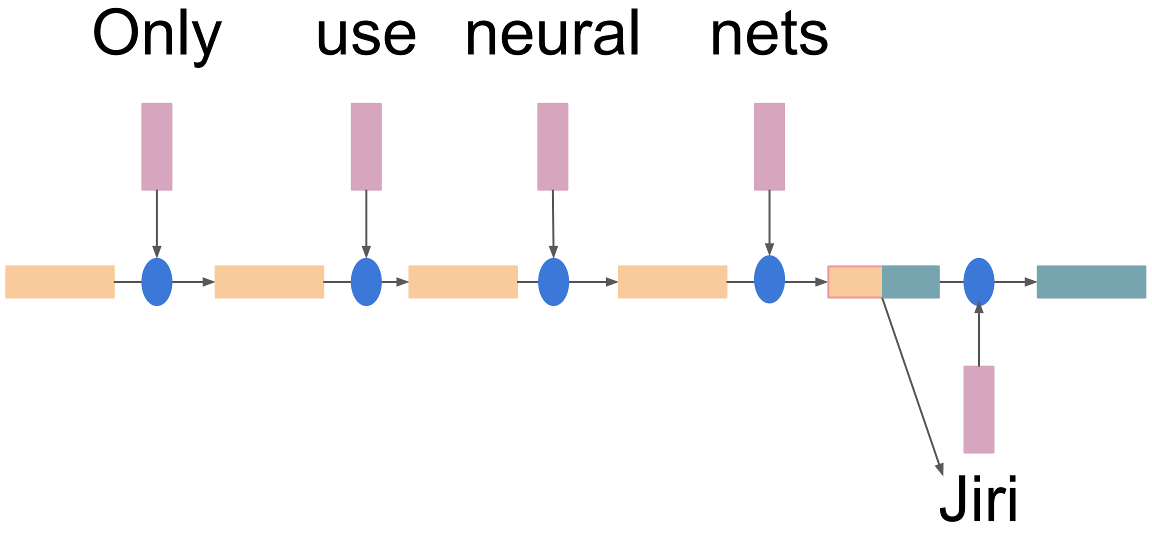
The method of building the representation is called an Encoder and is frequently an RNN.



Each memory vector in the encoder attempts to represent the sentence so far, but mostly represents the word most recently input.

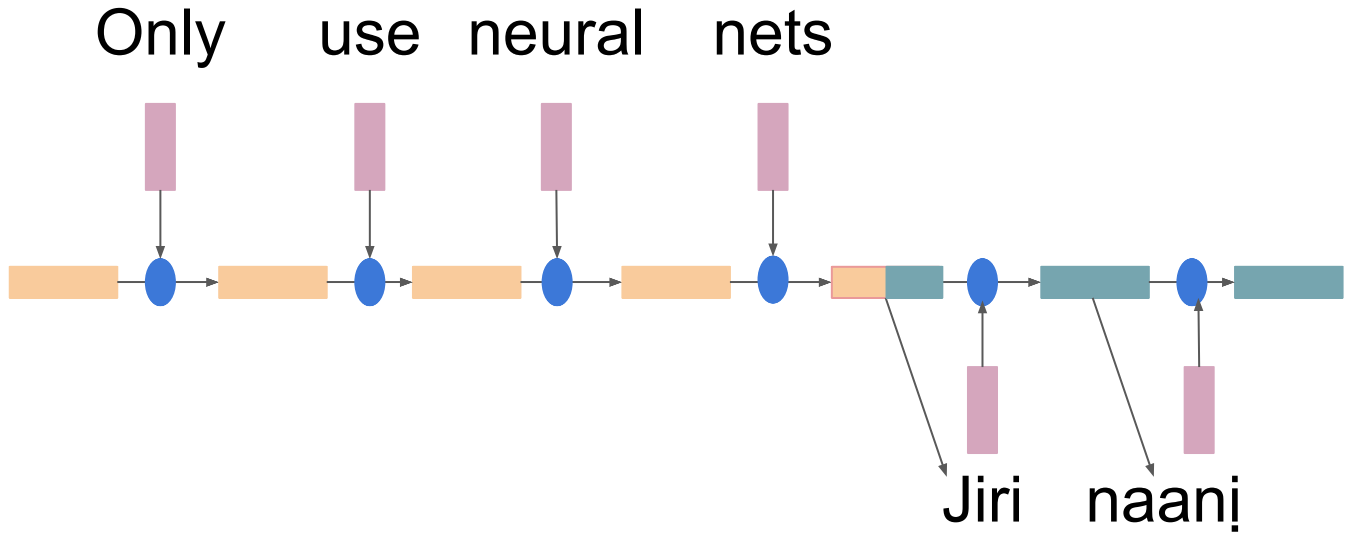
General idea: generate the output one token at a time

The model that takes the encoded representation and generates the output is called the Decoder, and, errrr, is also generally an RNN.



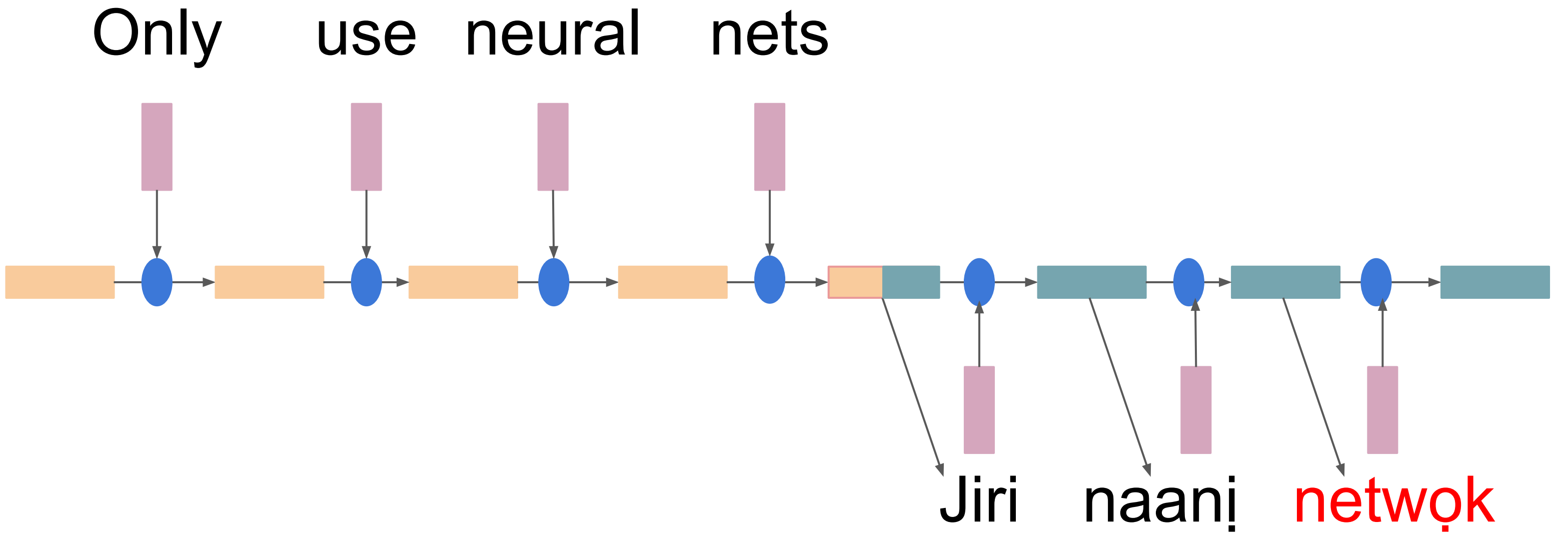
General idea: generate the output one token at a time

The model that takes the encoded representation and generates the output is called the Decoder, and, errrr, is also generally an RNN.



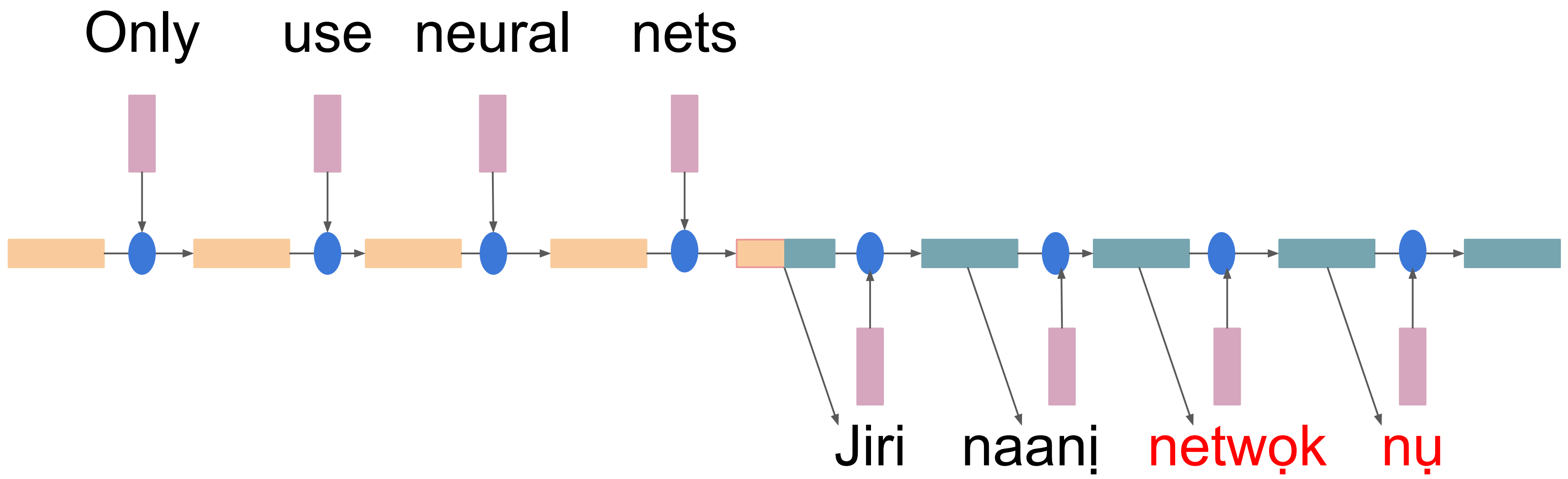
General idea: generate the output one token at a time

The model that takes the encoded representation and generates the output is called the Decoder, and, errrr, is also generally an RNN.



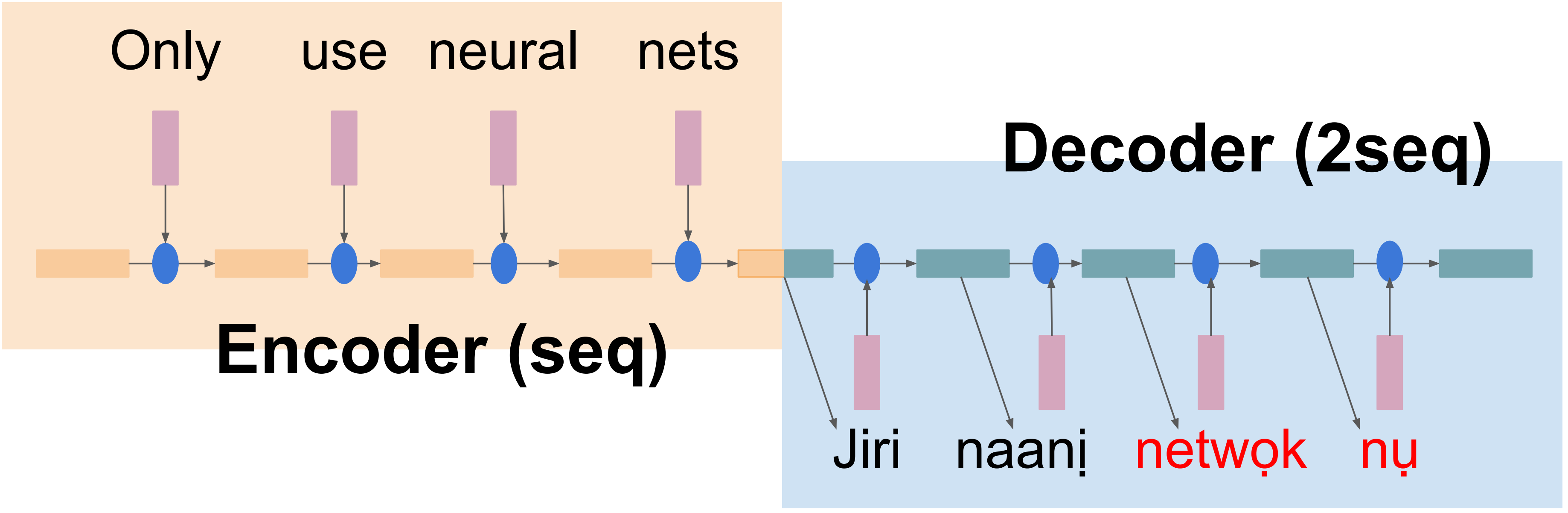
General idea: generate the output one token at a time

The model that takes the encoded representation and generates the output is called the Decoder, and, errrr, is also generally an RNN.



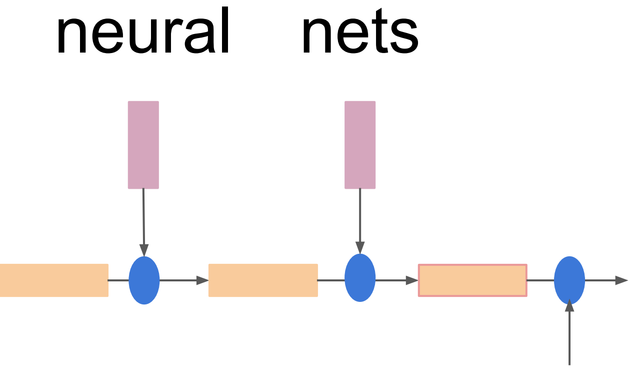
General idea: generate the output one token at a time

The model that takes the encoded representation and generates the output is called the Decoder, and, errrr, is also generally an RNN.



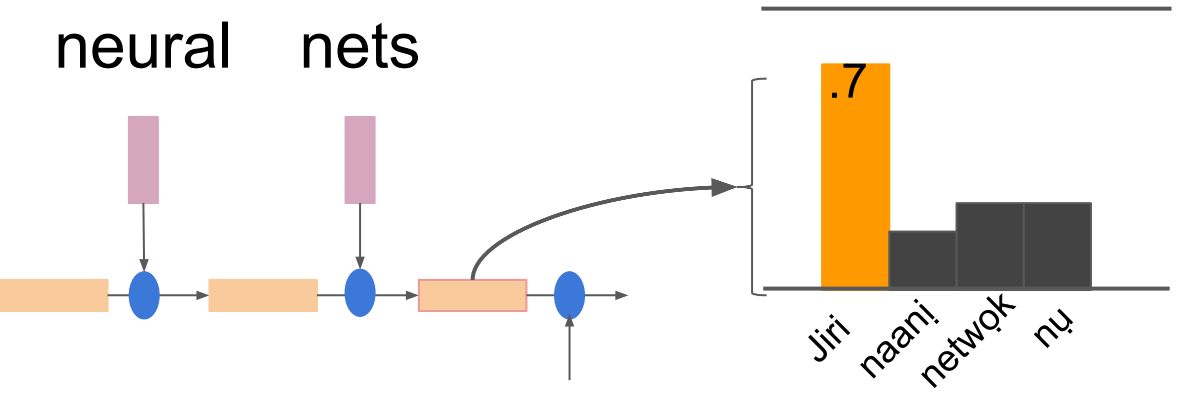
How is it trained?

In practice, training for a single sentence is done by “forcing” the decoder to generate gold sequences, and penalizing it for assigning the sequence a low probability. Losses for each token in the sequence are summed. Then, the summed loss is used to take a step in the right direction in all model parameters (including word embeddings!) (*stochastic gradient descent.)*



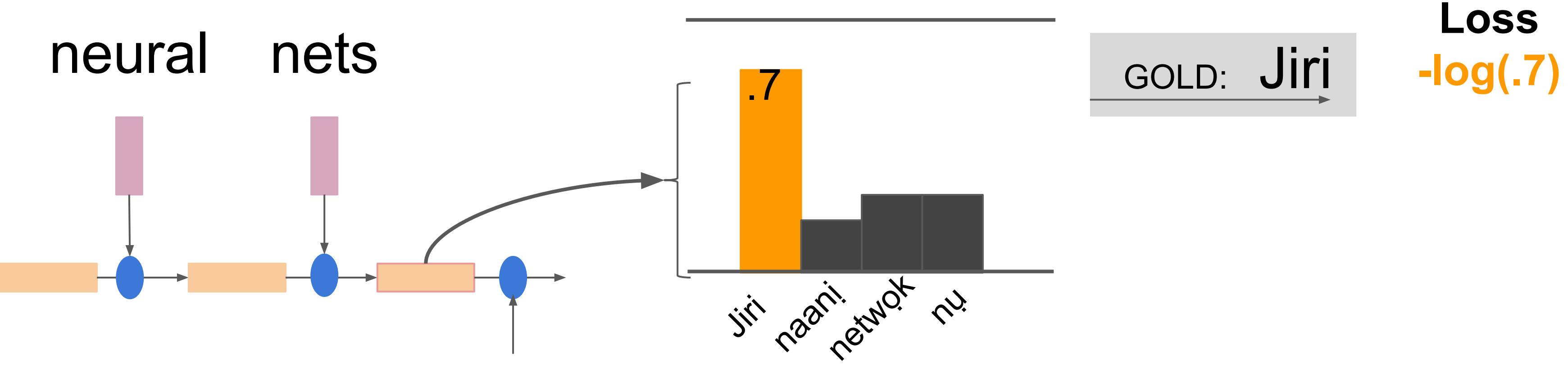
How is it trained?

In practice, training for a single sentence is done by “forcing” the decoder to generate gold sequences, and penalizing it for assigning the sequence a low probability. Losses for each token in the sequence are summed. Then, the summed loss is used to take a step in the right direction in all model parameters (including word embeddings!) (*stochastic gradient descent.)*



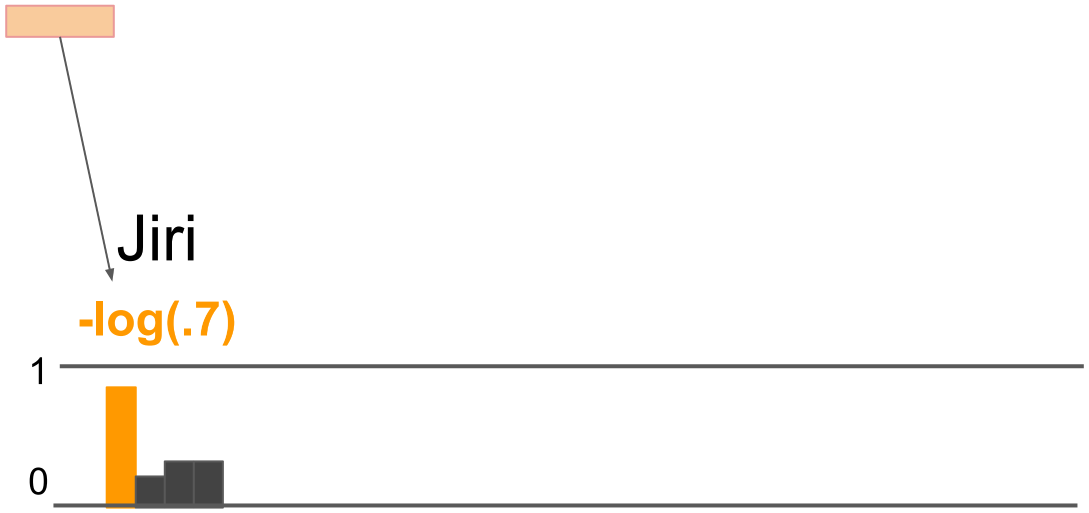
How is it formalized? How is it trained?

In practice, training for a single sentence is done by “forcing” the decoder to generate gold sequences, and penalizing it for assigning the sequence a low probability. Losses for each token in the sequence are summed. Then, the summed loss is used to take a step in the right direction in all model parameters (including word embeddings!) (*stochastic gradient descent.)*



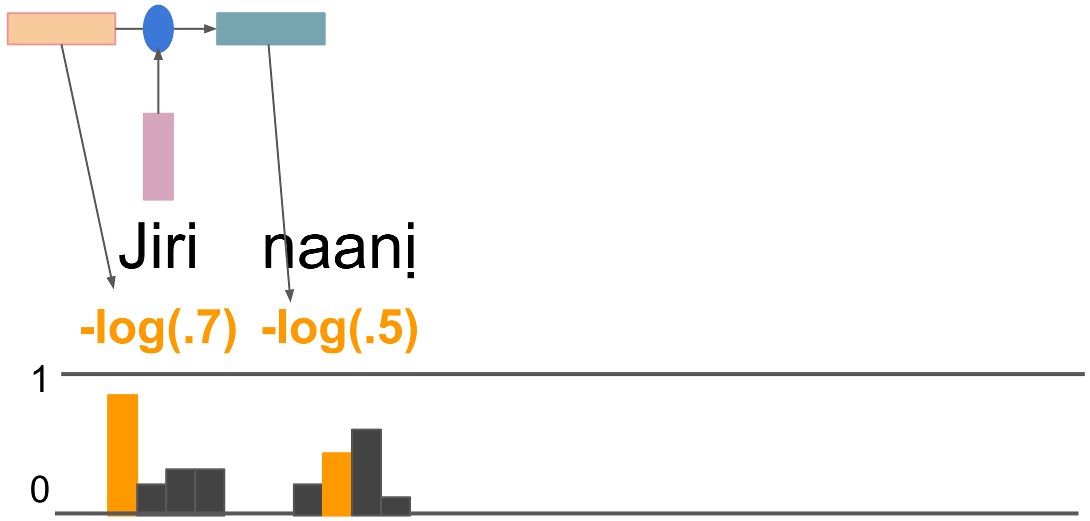
Sentence-level training

Almost all such networks are trained using **cross-entropy loss**. **At each step, the network produces a probability distribution over possible next tokens. This distribution is penalized from being different from the true distribution (e.g., a probability of 1 on the actual next token.)**



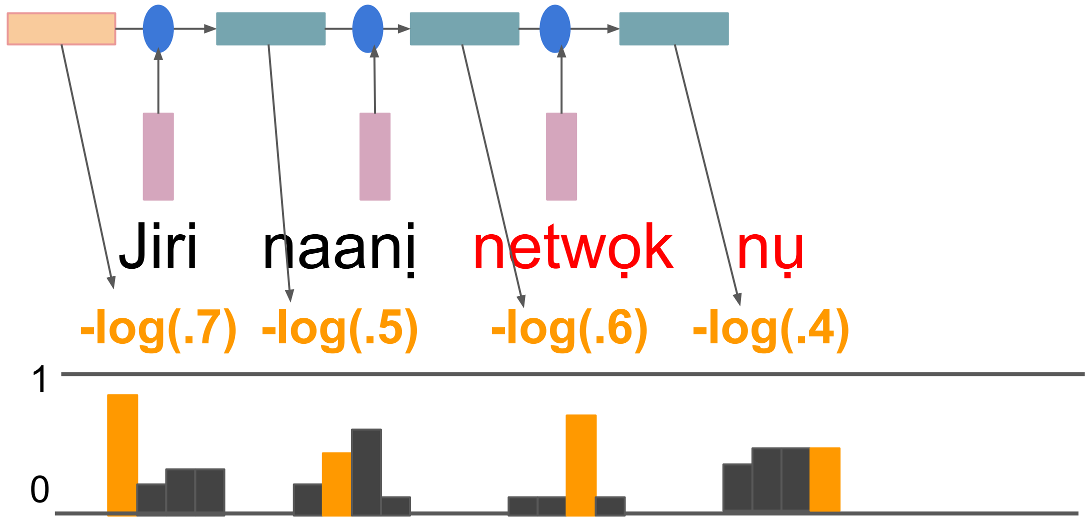
Sentence-level training

Almost all such networks are trained using **cross-entropy loss**. **At each step, the network produces a probability distribution over possible next tokens. This distribution is penalized from being different from the true distribution (e.g., a probability of 1 on the actual next token.)**



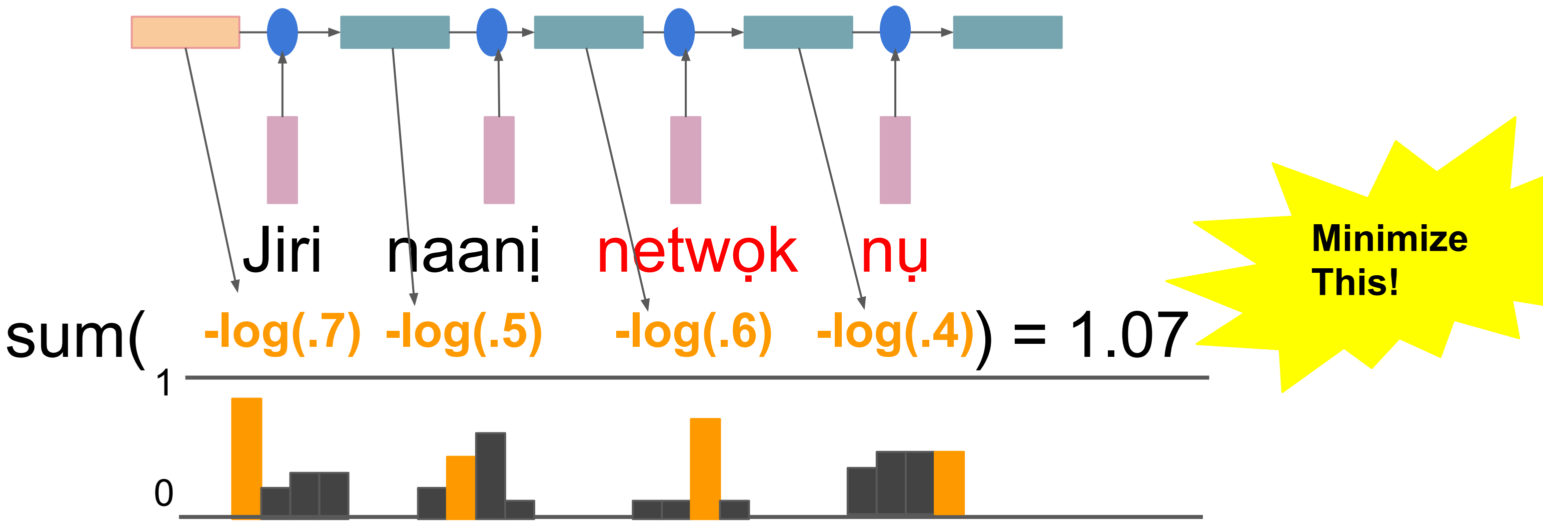
Sentence-level training

Almost all such networks are trained using **cross-entropy loss**. **At each step, the network produces a probability distribution over possible next tokens. This distribution is penalized from being different from the true distribution (e.g., a probability of 1 on the actual next token.)**



Sentence-level training

Almost all such networks are trained using **cross-entropy loss**. **At each step, the network produces a probability distribution over possible next tokens. This distribution is penalized from being different from the true distribution (e.g., a probability of 1 on the actual next token.)**



How is it formalized?

Let h_t be the RNN hidden state at timestep t: 🟧

How is it formalized?

Let h_t be the RNN hidden state at timestep t:

Let x_t be the input vector at timestep t:

How is it formalized?

Let h_t be the RNN hidden state at timestep t:

Let x_t be the input vector at timestep t:

The RNN equation posits 2 matrices and 1 vector as parameters:

> $\mathbf{W}^{hx}$ integrates input vector information.

How is it formalized?

Let h_t be the RNN hidden state at timestep t:

Let x_t be the input vector at timestep t:

The RNN equation posits 2 matrices and 1 vector as parameters:

- $\mathbf{W}^{hx}$ integrates input vector information.
- $\mathbf{W}^{hh}$ integrates information from the previous timestep.

How is it formalized?

Let h_t be the RNN hidden state at timestep t:

Let x_t be the input vector at timestep t:

The RNN equation posits 2 matrices and 1 vector as parameters:

- $\mathbf{W}^{hx}$ integrates input vector information.
- $\mathbf{W}^{hh}$ integrates information from the previous timestep.
- $\mathbf{b}^{h}$ is a bias term. (What function does this perform?)

How is it formalized?

Let h_t be the RNN hidden state at timestep t:

Let x_t be the input vector at timestep t:

The RNN equation posits 2 matrices and 1 vector as parameters:

> $\mathbf{W^{hx}}$ integrates input vector information.
>
> $\mathbf{W^{hh}}$ integrates information from the previous timestep.
>
> $\mathbf{b^{h}}$ is a bias term. (What function does this perform?)

The RNN equation is: $\mathbf{h_t = \tanh(W^{hx}x_t + W^{hh}h_{t-1} + b^h)}$

How is it formalized?

Glossing over this slide is totally reasonable. Also feel free to check your phone, ping your Bitcoin investment, see if your The Boring Company® (Not a) Flamethrower has shipped.

For prediction, we take the current hidden state, and use it as *features* in what is more or less a linear regression.

How is it formalized?

Glossing over this slide is totally reasonable. Also feel free to check your phone, ping your Bitcoin investment, see if your The Boring Company® (Not a) Flamethrower has shipped.

For prediction, we take the current hidden state, and use it as *features* in what is more or less a linear regression.

Let d_t be our decision (e.g., word, POS tag) at timestep t. Let D be the set of all possible decisions. Let $\mathbf{s}_{t-1}$ be the most recent *decoder* hidden state.

How is it formalized?

Glossing over this slide is totally reasonable. Also feel free to check your phone, ping your Bitcoin investment, see if your The Boring Company® (Not a) Flamethrower has shipped.

For prediction, we take the current hidden state, and use it as *features* in what is more or less a linear regression.

Let d_t be our decision (e.g., word, POS tag) at timestep t. Let D be the set of all possible decisions. Let $\mathbf{s}_{t-1}$ be the most recent *decoder* hidden state.

$$\mathbf{d_t = \mathit{argmax}_{d' \in D}\ p(\ d' \mid x_{1:n}, d_{1:t-1})}$$

How is it formalized?

Glossing over this slide is totally reasonable. Also feel free to check your phone, ping your Bitcoin investment, see if your The Boring Company® (Not a) Flamethrower has shipped.

For prediction, we take the current hidden state, and use it as *features* in what is more or less a linear regression.

Let d_t be our decision (e.g., word, POS tag) at timestep t. Let D be the set of all possible decisions. Let $\mathbf{s}_{t-1}$ be the most recent *decoder* hidden state.

$$d_t = \mathit{argmax}_{d' \in D}\ p(\, d' \mid x_{1:n}, d_{1:t-1})$$

$$p(\, * \mid x_{1:n}, d_{1:t-1}) = \mathrm{softmax}_D(W^{Dh} s_{t-1} + b^D)$$

How is it formalized?

Glossing over this slide is totally reasonable. Also feel free to check your phone, ping your Bitcoin investment, see if your The Boring Company® (Not a) Flamethrower has shipped.

For prediction, we take the current hidden state, and use it as *features* in what is more or less a linear regression.

Let d_t be our decision (e.g., word, POS tag) at timestep t. Let D be the set of all possible decisions. Let $\mathbf{s}_{t-1}$ be the most recent *decoder* hidden state.

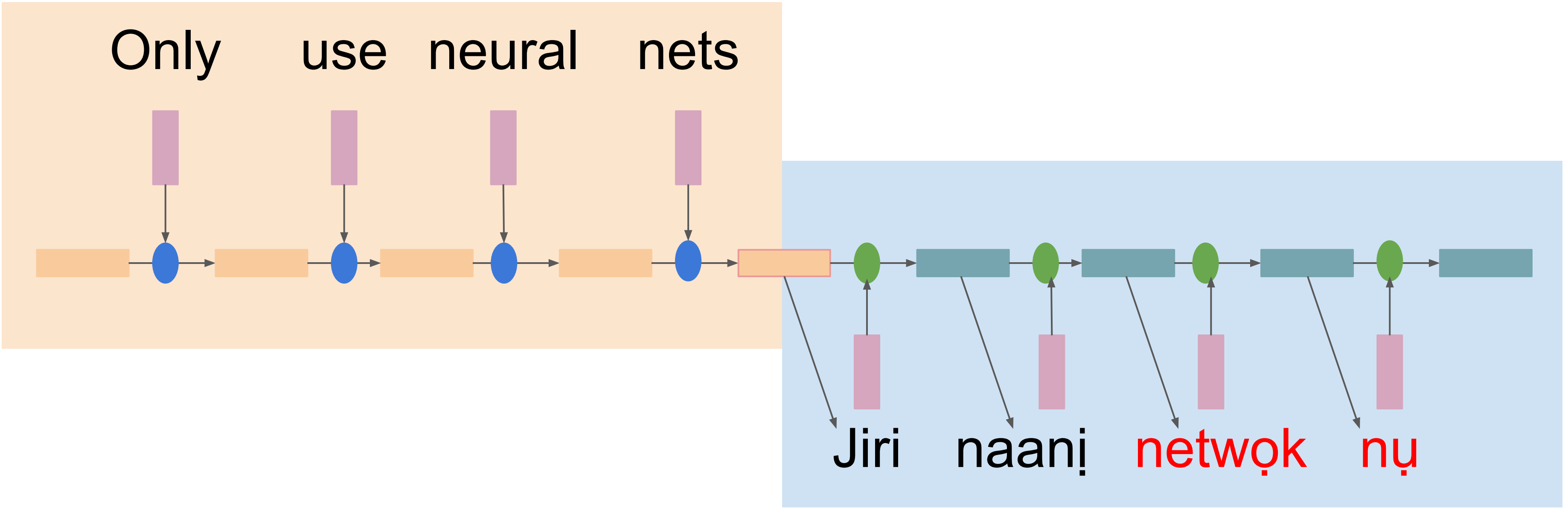
$$\mathbf{d_t = \mathit{argmax}_{\mathit{d'} \in D}\ p(\ d' \mid x_{1:n}, d_{1:t-1})}$$

$$\mathbf{p(\ * \mid x_{1:n}, d_{1:t-1}) = softmax_D(W^{Dh} s_{t-1} + b^D)}$$

Note that $\mathbf{W^{hD} s_{t-1} + b^D}$ produces a *vector of scores.* The **softmax** function normalizes scores to a probability distribution by exponentiating each dimension, and normalizing by the sum. For some choice k of K, $\mathbf{p(k) = e^{score(k)}} / \sum_{k' \in K} \mathbf{e^{\mathit{score(k')}}}$

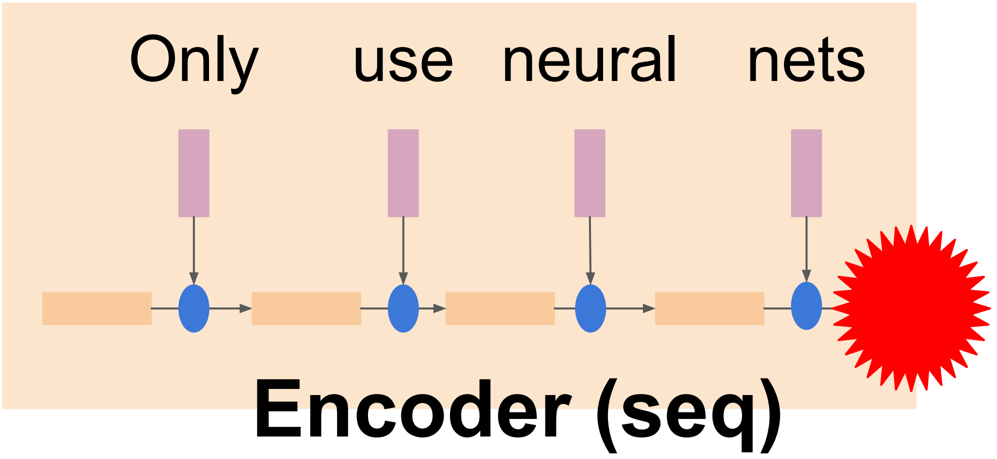
The information bottleneck and latent structure

Given the diagram below, what problem do you foresee when translating progressively longer sentences?



The information bottleneck and latent structure

We are trying to encode *variable-length* structure (e.g., variable-length sentences) in a fixed-length memory (e.g., only the 300 dimensions of your hidden state.)

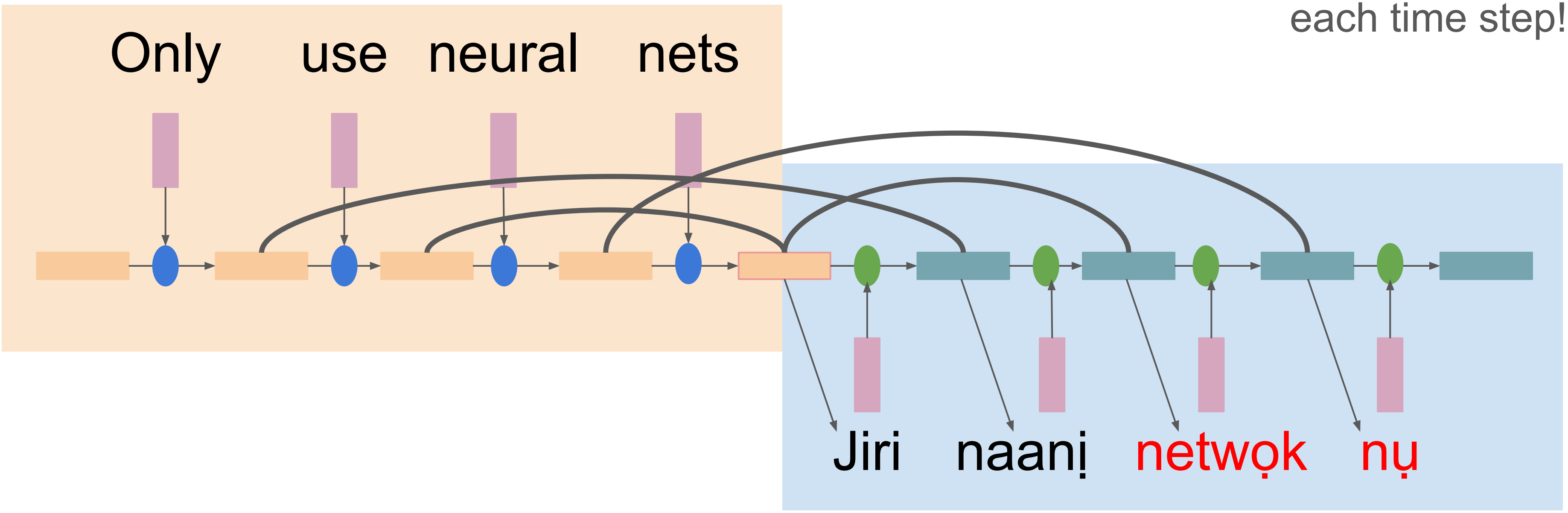


The last encoder hidden state is the bottleneck -- all information in the source sentence must pass through it to get to the decoder.

Finding a solution to this problem was the final advance that made neural MT competitive with previous approaches.

The information bottleneck and latent structure

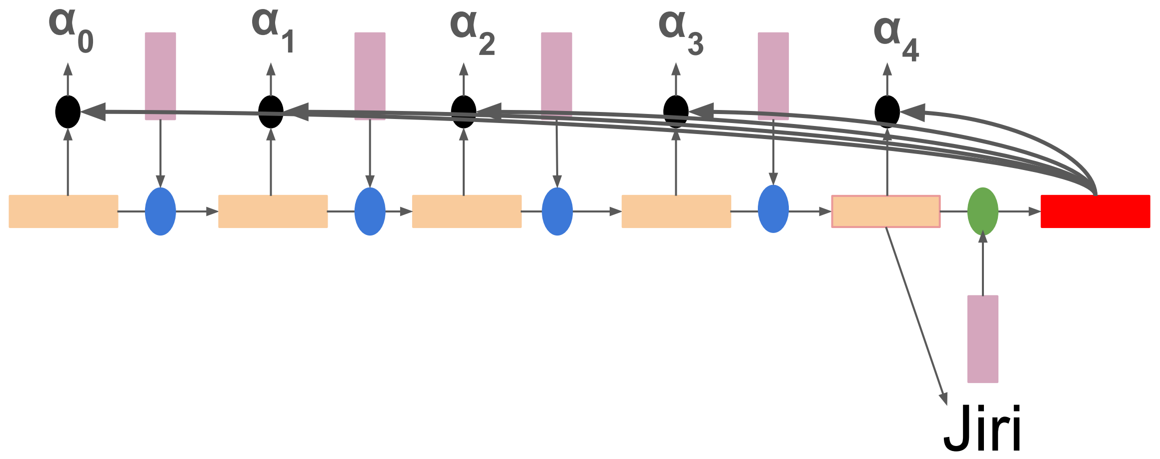
The key insight is related to the word alignment work we did last week. We allow the decoder to look at *any* encoder state, and let it learn which are important at each time step!



Learning to pay attention

Attention **summarizes the encoder**, focusing on specific parts/words.

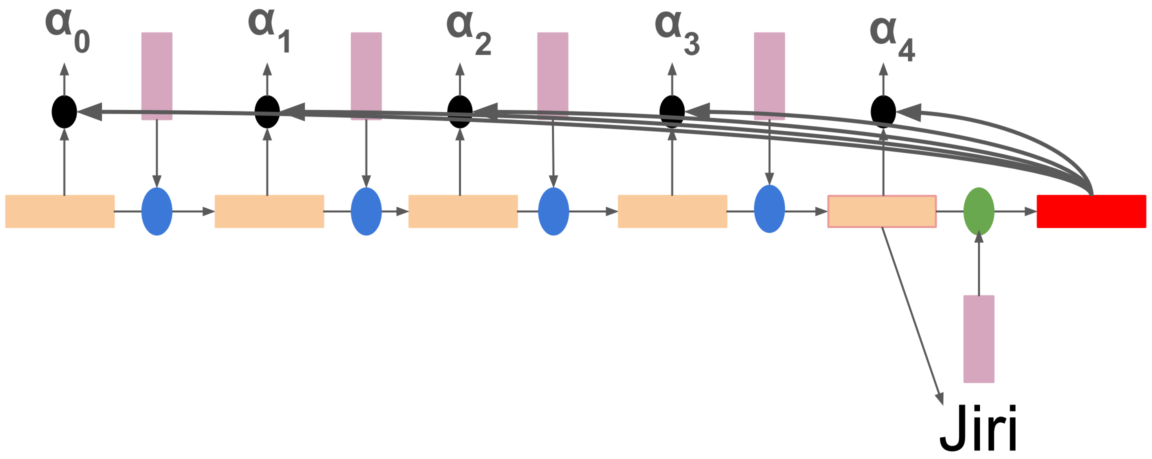
Step 1: Take the decoder state, and compute an *affinity* $\boldsymbol{\alpha}_i$ with all encoder states.



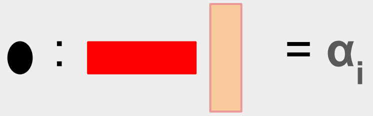
Learning to pay attention

Attention **summarizes the encoder**, focusing on specific parts/words.

Step 1: Take the decoder state, and compute an *affinity* $\boldsymbol{\alpha}_i$ with all encoder states.



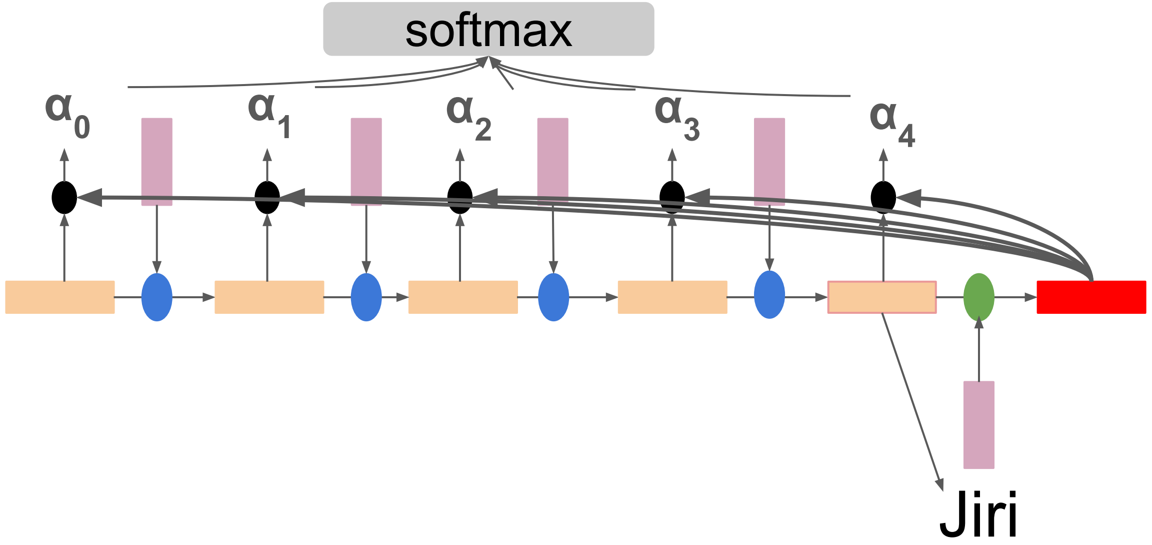
The affinity function, ● , is a dot product, or something similar.



Learning to pay attention

Attention **summarizes the encoder**, focusing on specific parts/words.

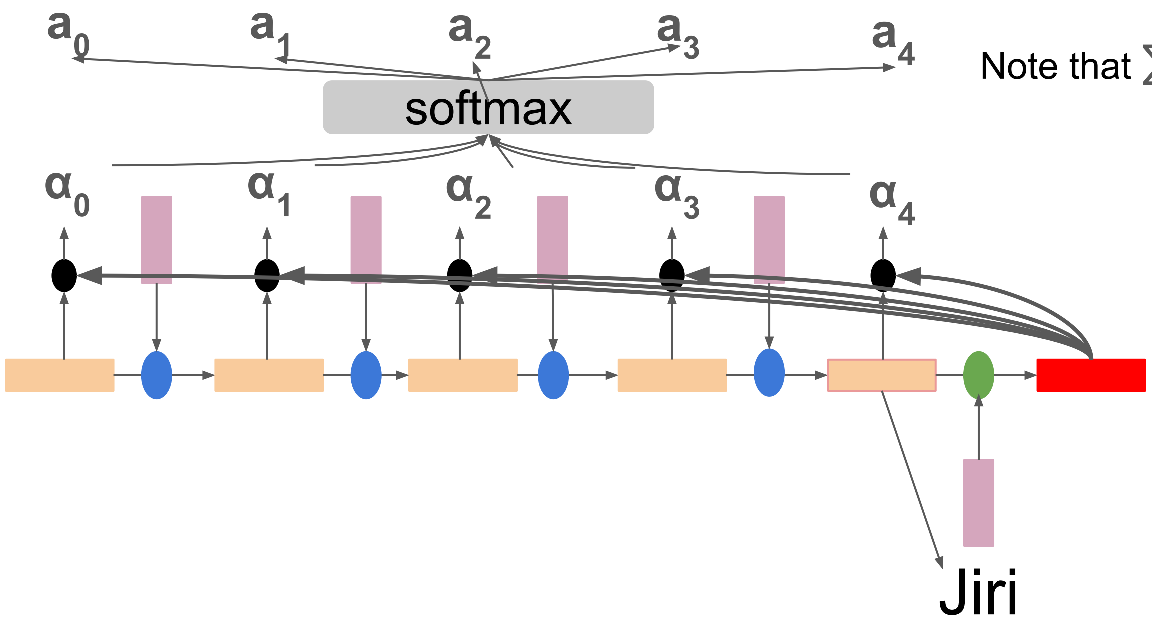
Step 2: Normalize the scores to sum to 1 by the softmax function.



Learning to pay attention

Attention **summarizes the encoder**, focusing on specific parts/words.

Step 2: Normalize the scores to sum to 1 by the softmax function.



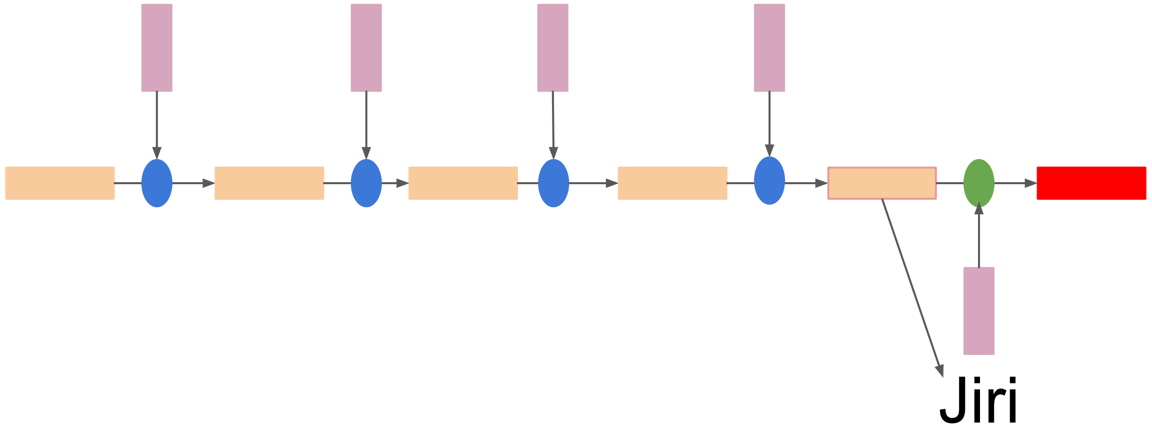
Note that $\sum_{i=1,2,3,4} a_i = 1$

Learning to pay attention

Attention **summarizes the encoder**, focusing on specific parts/words.

Step 3: Average the encoder states, weighted by the **a** distribution.

a_0 1 + a_1 1 + a_2 1 + a_3 1 + a_4 1 = 🟧



This weighted average

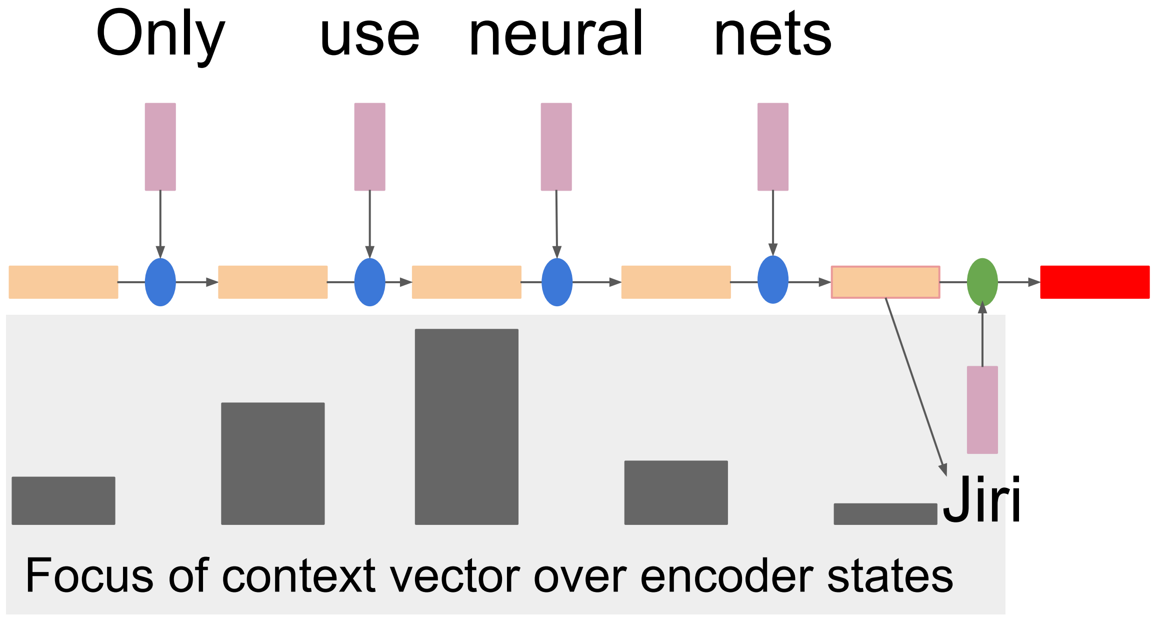
🟧

Is called the **context vector.**

Learning to pay attention

Attention **summarizes the encoder**, focusing on specific parts/words.

Step 3: Average the encoder states, weighted by the **a** distribution.

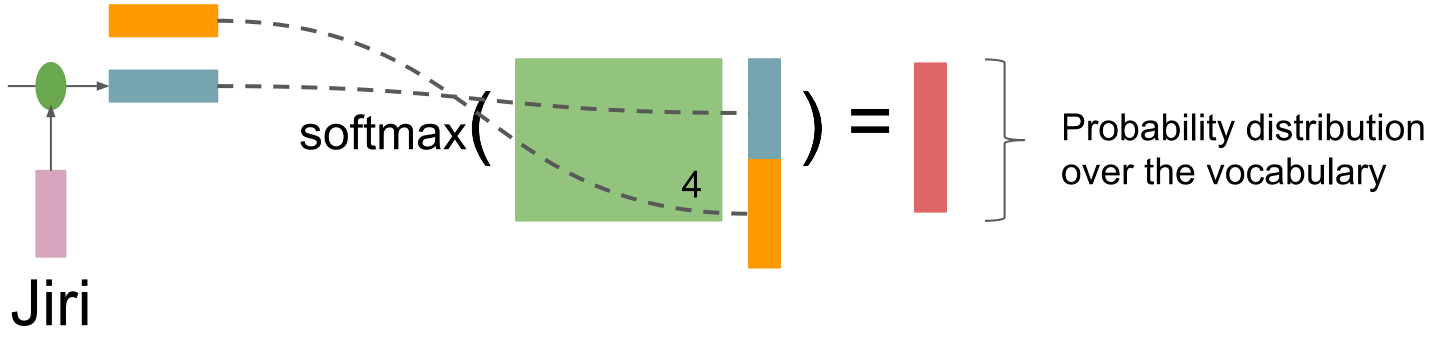


In this example, since “Jiri” means “use”, the attention will focus on the vectors around “use”.

Learning to pay attention

Attention **summarizes the encoder**, focusing on specific parts/words.

Step 4: Use the context vector at prediction, concatenating it to the decoder state.



This vector has the current decoder information, , but also a focused summary of the encoder, .

Attention Formalization

Glossing over this slide is totally reasonable. Feel free to check your phone, ping your Bitcoin investment, see if your The Boring Company® (Not a) Flamethrower has shipped.

Attention computes the **affinity between the decoder state and all encoder states**.There are many affinity computation methods, but they're all like a **dot product**.

Let there are *n* encoder states. The affinity between encoder state *i* and the decoder state is $\boldsymbol{\alpha}_i$. The encoder states are $\mathbf{h}_{1:n}$, and the decoder state is $\mathbf{s}_{t-1}$.

Let $\boldsymbol{\alpha}_i = f(\mathbf{h}_i, \mathbf{s}_{t-1}) = \mathbf{h}_i^T \mathbf{s}_{t-1}$

Let weights $\mathbf{a} = \text{softmax}(\boldsymbol{\alpha})$.

Let the context $\mathbf{c} = \sum_{i=1:n} \mathbf{h}_i \mathbf{a}_i$. *(Note that this is a weighted average.)*

Attention Formalization

Glossing over this slide is totally reasonable. Feel free to check your phone, ping your Bitcoin investment, see if your The Boring Company® (Not a) Flamethrower has shipped.

Attention is used at prediction as extra information in the final prediction.

Reminder, we let the context $\mathbf{c} = \sum_{i=1:n} \mathbf{h}_i \mathbf{a}_i$. *(Weighted average of encoder states.)*

Attention Formalization

Glossing over this slide is totally reasonable. Feel free to check your phone, ping your Bitcoin investment, see if your The Boring Company® (Not a) Flamethrower has shipped.

Attention is used at prediction as extra information in the final prediction.

Reminder, we let the context $\mathbf{c} = \sum_{i=1:n} \mathbf{h}_i \mathbf{a}_i$. *(Weighted average of encoder states.)*

Let the notation [**s**;**c**] mean the concatenation of vectors **s** and **c**.

$$\mathbf{d}_t = \mathit{argmax}_{d' \in D}\ p(\, d' \mid x_{1:n}, d_{1:t-1})$$

(same as before, without attention)

Attention Formalization

Glossing over this slide is totally reasonable. Feel free to check your phone, ping your Bitcoin investment, see if your The Boring Company® (Not a) Flamethrower has shipped.

Attention is used at prediction as extra information in the final prediction.

Reminder, we let the context $\mathbf{c} = \sum_{i=1:n} \mathbf{h}_i \mathbf{a}_i$. *(Weighted average of encoder states.)*

Let the notation [**s**;**c**] mean the concatenation of vectors **s** and **c**.

$$\mathbf{d_t = \mathit{argmax}_{d' \in D}\ p(\ d' \mid x_{1:n}, d_{1:t-1})}$$

$$\mathbf{p(\ *\ \mid x_{1:n}, d_{1:t-1}) = softmax_D(W^{D(2h)}[s_{t-1};c]+b^D)}$$

So, the only difference is that the final prediction uses the context vector concatenated to the decoder state to make the prediction.

Empirical considerations

There are a lot of “hyperparameter” choices that can greatly affect the quality of your model. **In short, take parameters from papers/tutorials, and grid search (try many combinations of parameters) around them.**

RNN variants: LSTMs have a different (much better) recurrent equation.

Hidden state sizes: larger: more memory! Requires more data.

Embedding sizes: more representation power! Requires more data.

Learning rate: the step size you take in learning your parameters! Start this “large”, and cut it in half when your training stops improving development set performance.

Empirical considerations

There are a lot of “hyperparameter” choices that can greatly affect the quality of your model. **In short, take parameters from papers/tutorials, and grid search (try many combinations of parameters) around them.**

Regularization: “dropout” prevents overfitting by making each node in your hidden state unavailable for an observation with a given probability. Try some values around .2 to .3.

Batch size: The number of observations to group together before performing a parameter update step. Larger batches: less fine-grained training, many more observations per minute, especially on GPU.

Case study: text simplification

Text simplification is the process in which a text is transformed into an equivalent text that can be more easily read by a broader audience (Saggion, 2017).

Simplification can be used as a preprocessing tool for improving performance of many NLP end-tasks such as parsing, SRL, summarization, Information Retrieval etc.

Case study: text simplification

Text simplification is the process in which a text is transformed into an equivalent text that can be more easily read by a broader audience (Saggion, 2017).

Simplification can be used as a preprocessing tool for improving performance of many NLP end-tasks such as parsing, SRL, summarization, Information Retrieval etc.

> “There’s just one major hitch: the primary purpose of education is to develop citizens with a wide variety of skills.”

$$\downarrow$$

> “The purpose of education is to develop many skills.”

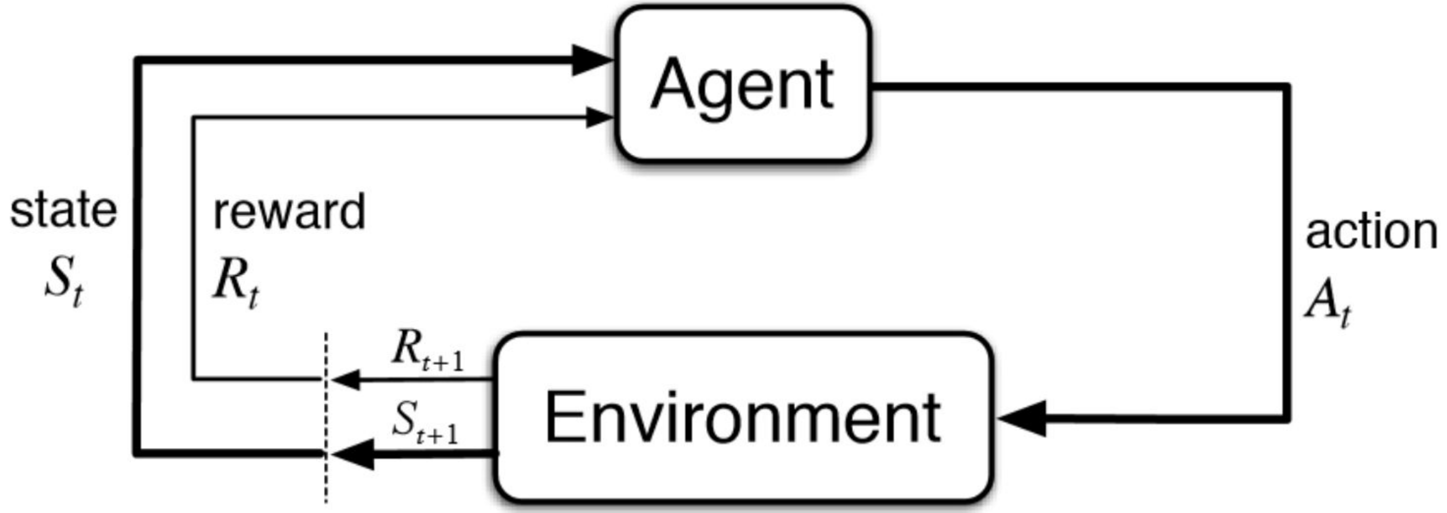
Case study: text simplification

Text simplification can be thought of in part as monolingual machine translation.

Problem: The most common rewrite operation is copying from the complex sentence to the simple sentence.

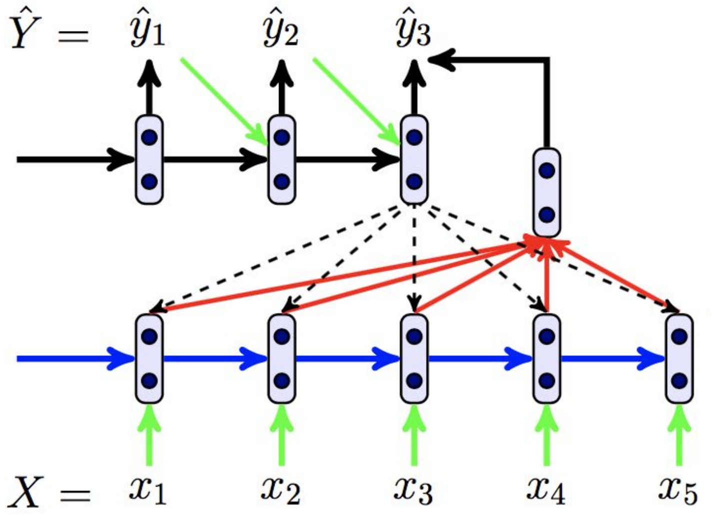
- One solution: Add in reinforcement learning (Zhang and Lapata 2017), to encourage the model to use other rewrite operations, such as deletion, substitution, word reordering.

A brief introduction to Reinforcement Learning



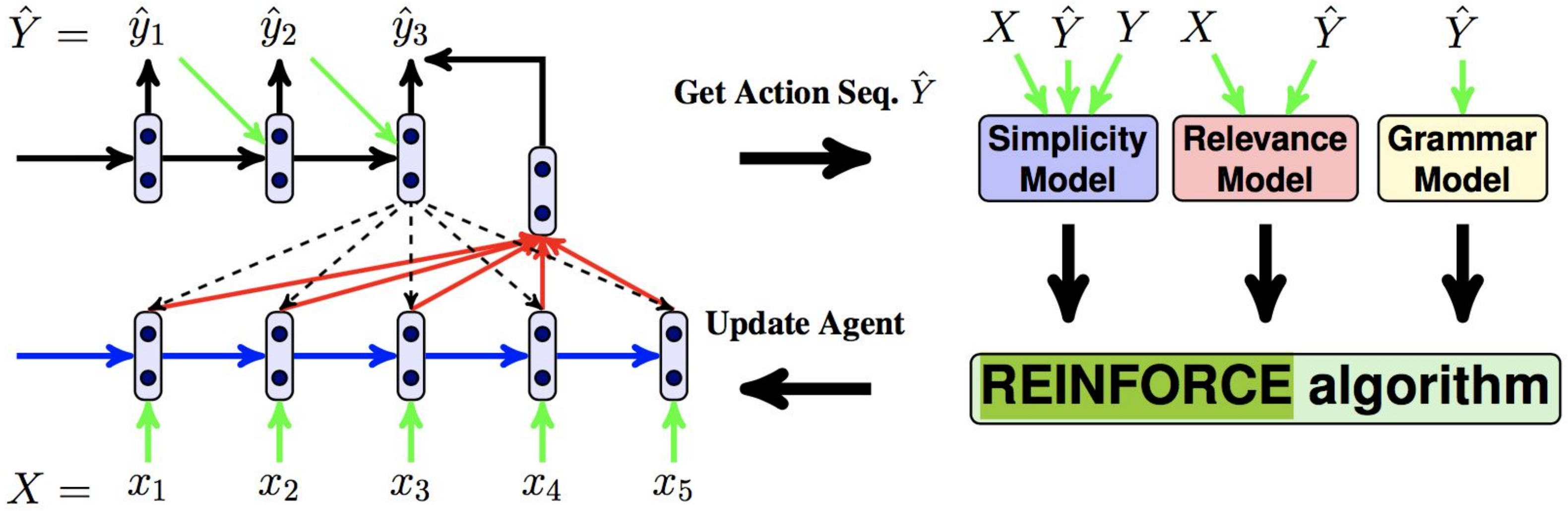
The reinforcement learning framework (Sutton and Barto, 1998)

Case study: text simplification


$\hat{Y} = \hat{y}_1 \; \hat{y}_2 \; \hat{y}_3$
$X = x_1 \; x_2 \; x_3 \; x_4 \; x_5$

Basic encoder-decoder model, from (Zhang and Lapata, 2017).

Case study: text simplification



$\hat{Y} = \hat{y}_1 \quad \hat{y}_2 \quad \hat{y}_3$
Get Action Seq. $\hat{Y}$
$X \quad \hat{Y} \quad Y \quad X \quad \hat{Y} \quad \hat{Y}$
Simplicity Model
Relevance Model
Grammar Model
REINFORCE algorithm
Update Agent
$X = x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5$

Encoder-Decoder model with reinforcement learning (Zhang and Lapata, 2017).

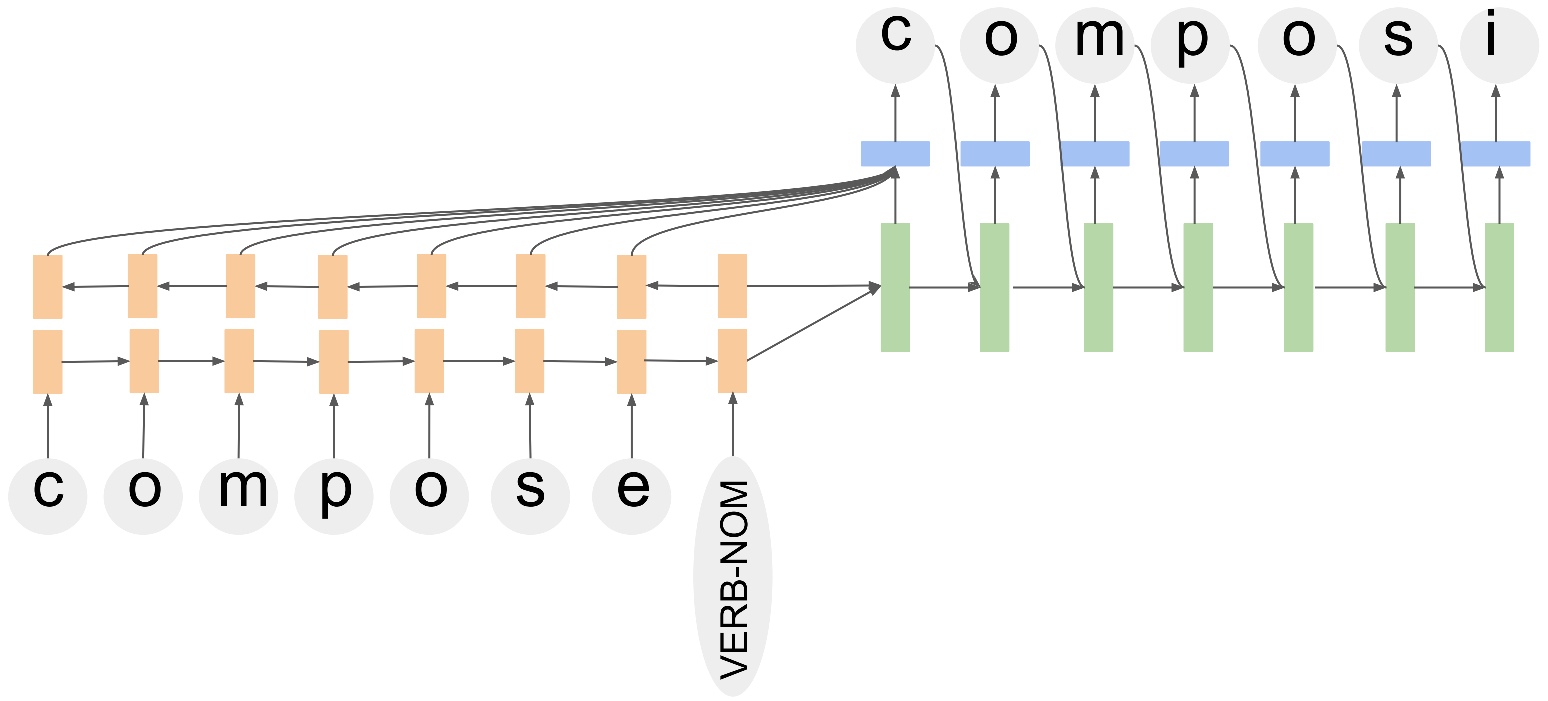
Derivational morphology

- Process of generating new words from existing words
  - Changes semantic meaning
  - Often a new part-of-speech

employ	V -> N, Agent	employer
employ	V -> N, Passive	employee
employ	V -> N, Result	employment
employ	V -> Adj, Potential	employable
employable	V -> Adj -> N, Stative	employability

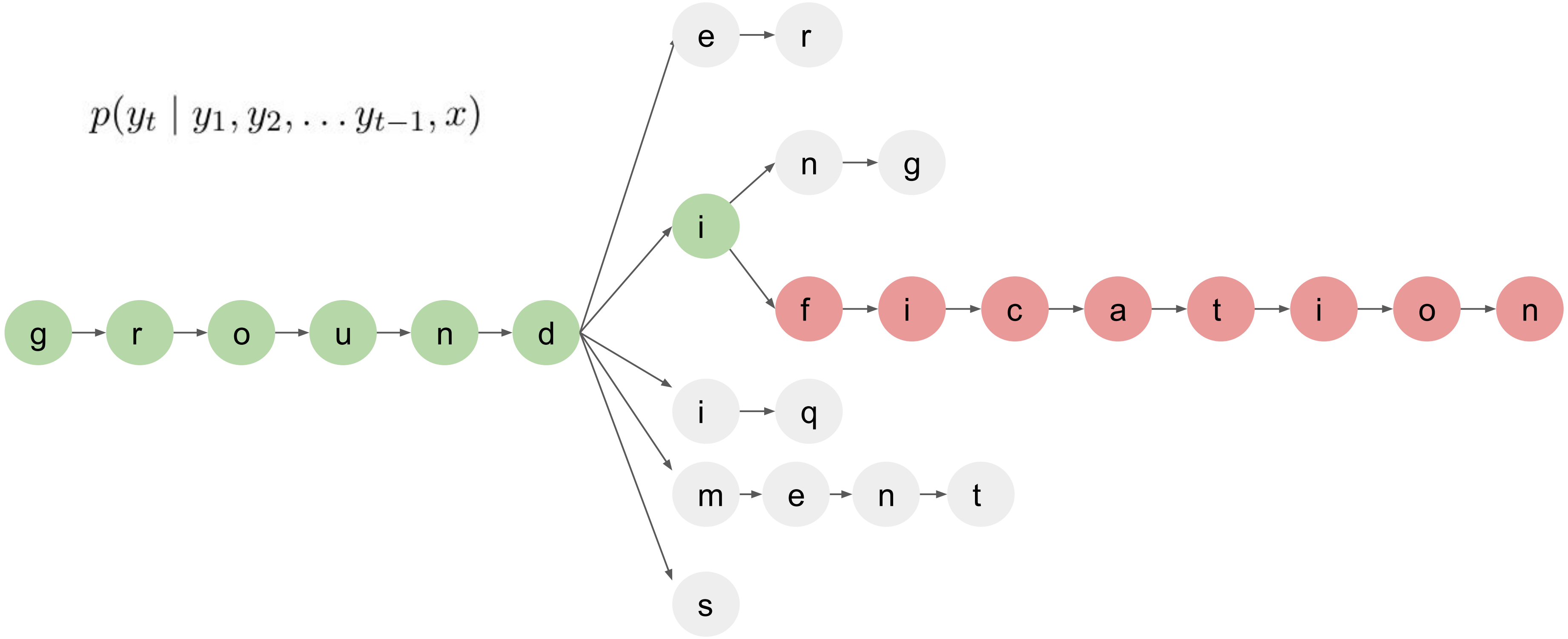
Encoder (seq) **Decoder (2seq)**

Derivational morphology



Derivational morphology: search

$$p(y_t \mid y_1, y_2, \ldots y_{t-1}, x)$$



Reference Sheet

🟧 🟦 The memory vector, or “state” . Color denotes whether encoder or decoder.

🟪 The “word vector” representation of the word.

🔵 The RNN function, which combines the word vector and the previous state to create a new state.

🟩 A learned parameter matrix

$\mathbf{W}^{hx}$ integrates input vector information.

$\mathbf{W}^{hh}$ integrates information from the previous timestep.

$\mathbf{b}^{h}$ is a bias term.

d_t is our decision at timestep t.

The RNN equation is:

$$\mathbf{h}_t = \tanh(\mathbf{W}^{hx}\mathbf{x}_t + \mathbf{W}^{hh}\mathbf{h}_{t-1} + \mathbf{b}^{h})$$

$$d_t = \mathit{argmax}_{d' \in D}\ p(\, d' \mid x_{1:n}, d_{1:t-1})$$

$$p(\, * \mid x_{1:n}, d_{1:t-1}) =$$

$$\mathrm{softmax}_D(\mathbf{W}^{Dh}\mathbf{s}_{t-1} + \mathbf{b}^{D})$$