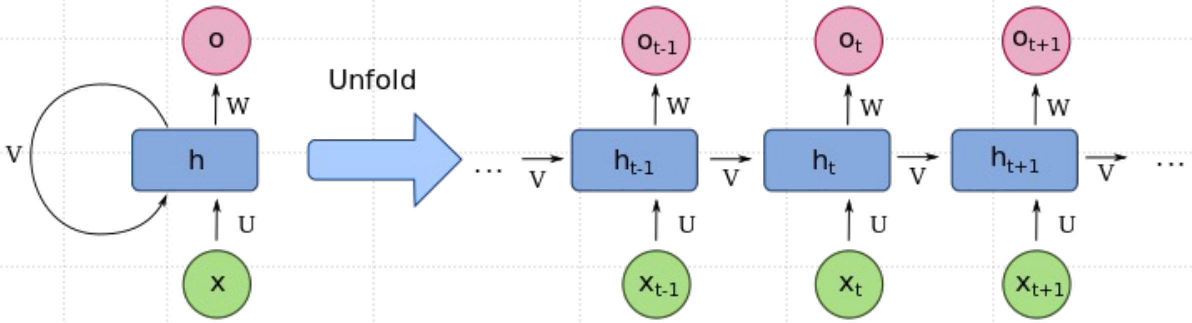
Introduction to Large Language Models



CSS 100

Sean Trott

Spring 2024

Language models: the basics

A **language model** assigns a probability to a word or sequences of words, typically in some context and order.

- An *N*-gram language model bases these probabilities on the number of times a given word *w* has been observed in a context of size *N*.

Please turn your homework ____.

"in" "refrigerator"

$$\mathrm{P}(\text{“}in\text{”}|homework) = \frac{C(\text{“}homework\ in\text{”})}{C(\text{“}homework\text{”})}$$

Language models: the basics

A **language model** assigns a probability to a word or sequences of words, typically in some context and order.

A **large language model (LLM)** is a neural network with many parameters trained on a word-prediction task—i.e., a language model using a neural network.

- “Large” = lots of parameters + training data.
- Given a context, a language model learns to fill in the blank.
- Like other neural networks, LLMs do this by updating their weights.

An LLM is **self-supervised**: uses structure of language as its own training signal.

Language models: the basics

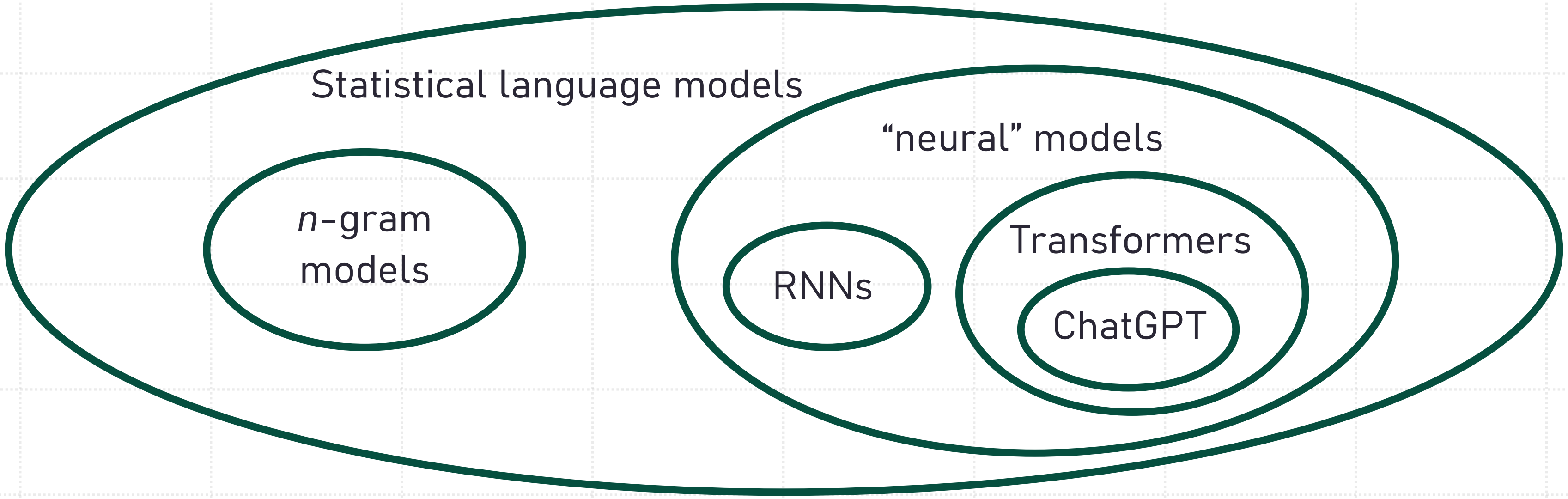
A **language model** assigns a probability to a word or sequences of words, typically in some context and order.

A **large language model (LLM)** is a neural network with many parameters trained on a word-prediction task—i.e., a language model using a neural network.

- "Large" = lots of parameters + training data.
- Given a context, a language model learns to fill in the blank.
- Like other neural networks, LLMs do this by updating their weights.

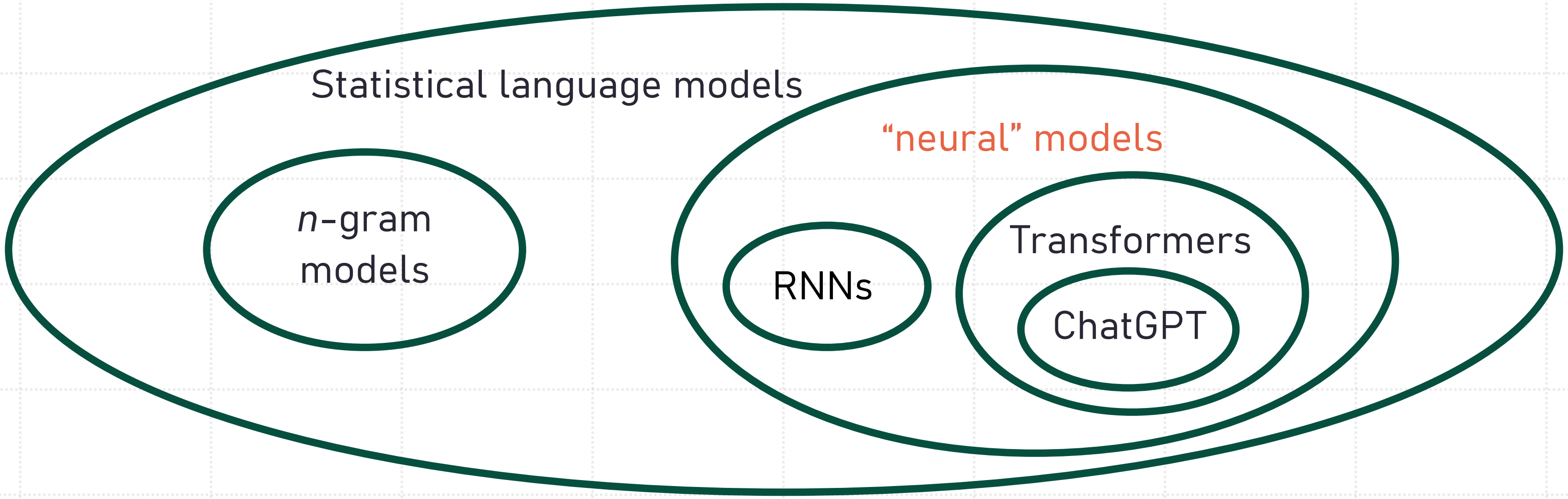
A brief taxonomy

- Many approaches to **language modeling**.
- All revolve around **statistical learning** in some way.



A brief taxonomy

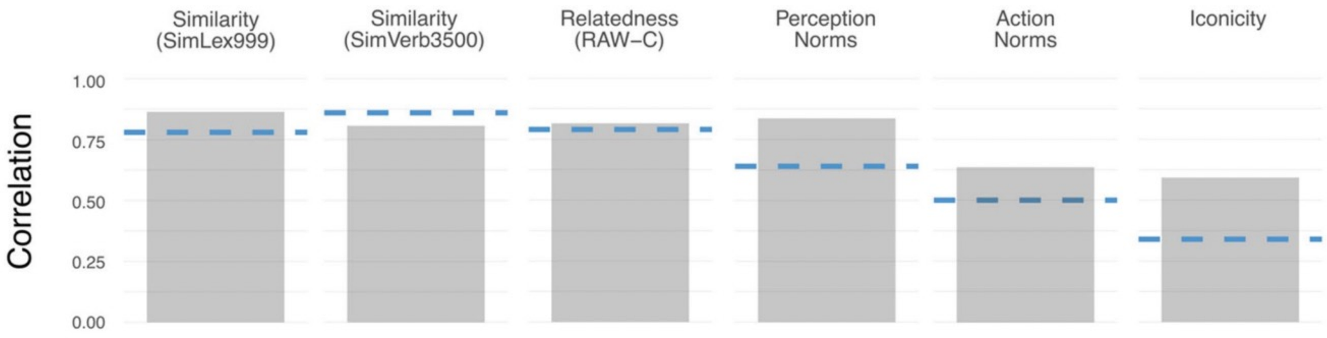
- Many approaches to **language modeling**.
- All revolve around **statistical learning** in some way.



How does CSS intersect with LLMs?

- LLMs are poised to impact society.
- LLMs are very impressive—but also hard to interpret.
- LLMs can also accelerate scientific research.

Each of these are related to CSS.



Lecture plan

- Review: embeddings.
- Common architectures:
  - Feedforward language model.
  - Recurrent neural network.
  - Transformer architecture.
- Next time: LLMs in Python!

Lecture plan

- Review: embeddings.
- Common architectures:
  - Feedforward language model.
  - Recurrent neural network.
  - Transformer architecture.
- Next time: LLMs in Python!

Introducing vector semantics

In **vector semantics**, a word is represented by a vector: an array of numbers that place the word in some N-dimensional space.

Words with similar meanings should be "nearby" in space.

Because vectors are *numbers*, they can also be manipulated and transformed.

But where do the vectors come from?

Word counts: a naïve approach

- **Basic premise**: we can represent words as <u>vectors</u> reflecting how they <u>distribute</u>.

A **co-occurrence matrix** is a way of representing how often words occur in different contexts.

Term-document matrix

	As You Like It	Twelfth Night	Julius Caesar	Henry IV
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

How often does a given word occur in different Shakespeare plays (our "corpus")?

Introducing word embeddings

A **word embedding** is a short, dense vector, where “dense” means that most dimensions are non-zero.

- In NLP, dense vectors usually work better than sparse vectors.
  - Easier to fit a classifier to 300-D embeddings than 100000-D vectors.
- Dense vectors also seem to capture synonymy better.
  - Forcing vectors to represent words with fewer dimensions means that each dimension has more information.
- In 2013, the ***word2vec* package** was introduced for learning word embeddings.

Note: a key issue is often **context window size**—how many words to include in "context"?

Pt. 1: The *word2vec* classifier

- Goal: we want to train a **classifier** to learn the probability that some *context c* is an actual context of *word w*.

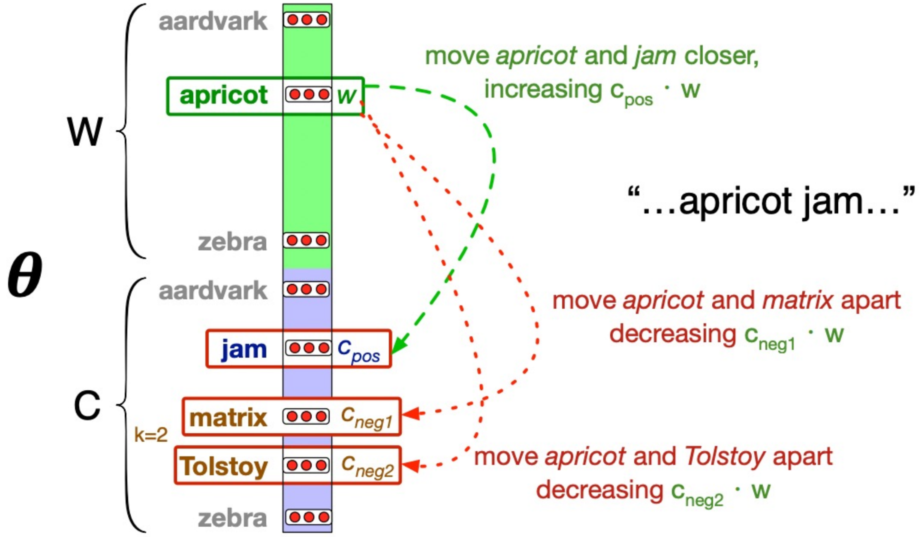
$$P(+|w,c)$$

- Intuition: two words are likely to co-occur if they have **similar embeddings** (i.e., a high dot product).
- So given *w* and context word(s) *c*, the classifier should **assign a probability** based on the similarity of their embeddings.

That means we need to <u>learn</u> an embedding for each word in our vocabulary.

Pt. 2a: Learning—the intuition.

- First, gather **training data**: examples of *(w, c)* that do and don't co-occur.
- Then, initialize **random** embeddings for each word in vocabulary.
- Iteratively **update** embeddings so (+|w,c) are closer, and (-|w,c) are farther apart.

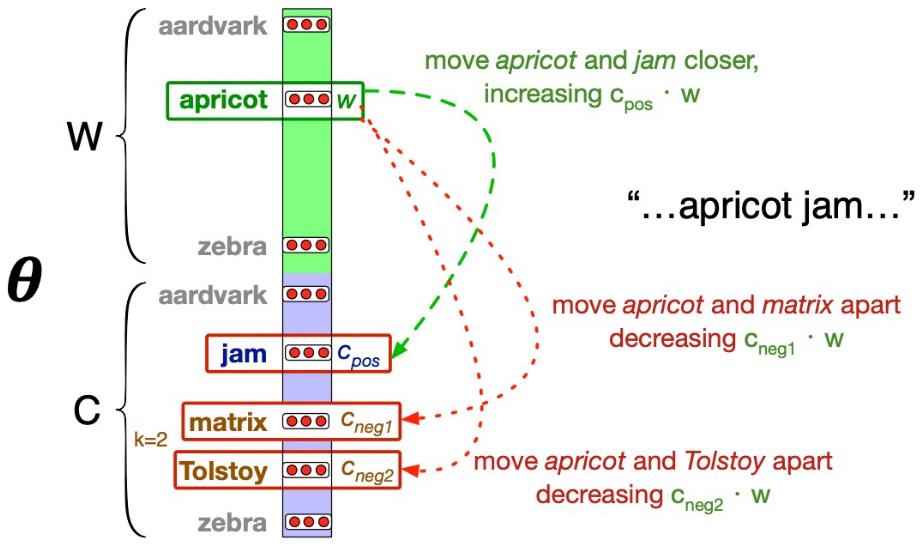


Technically, algorithm learns *two* embeddings for each word:

- W: represents word when it's the target.
- C: represents word when it's the context.

Pt. 2a: Learning—the intuition.

- First, gather **training data**: examples of *(w, c)* that do and don't co-occur.
- Then, initialize **random** embeddings for each word in vocabulary.
- Iteratively **update** embeddings so (+|w,c) are closer, and (-|w,c) are farther apart.



Continue this process until further improvements reach diminishing returns.

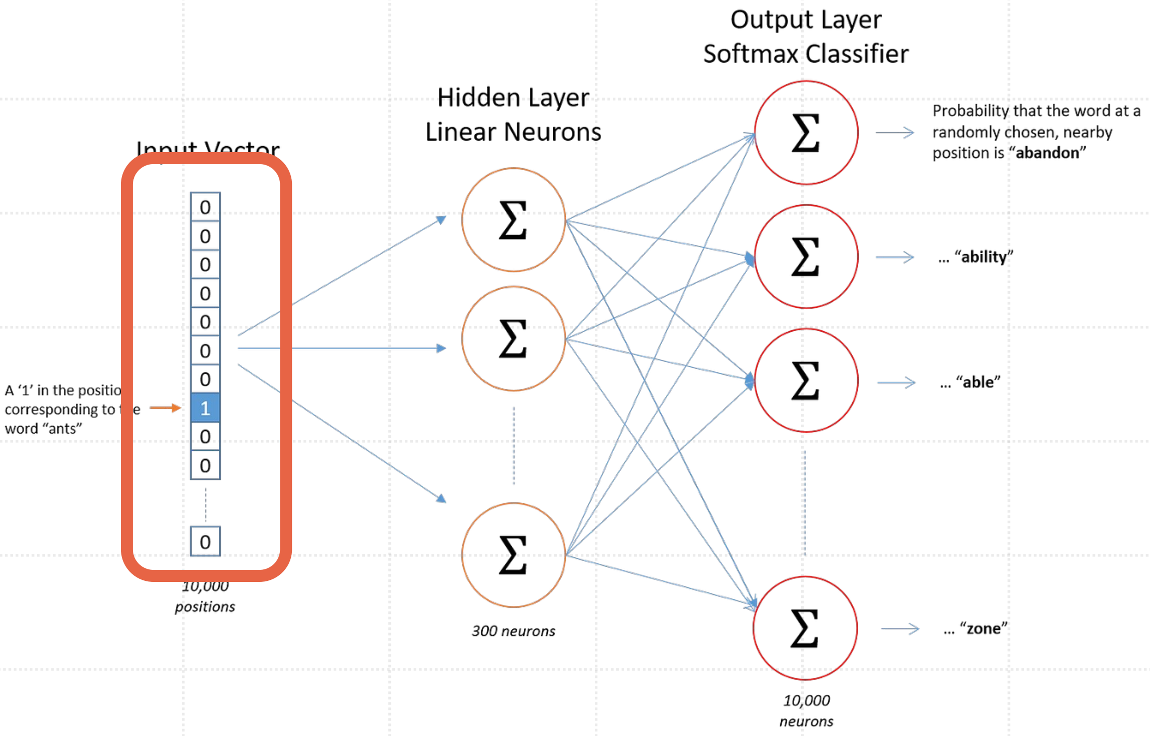
Once process is finished, **W** is our final matrix of dense embeddings for each word.

https://towardsdatascience.com/skip-gram-neural-network-from-scratch-485f2e688238

Pt. 2b: Learning—the details.

- Technically, *word2vec* uses a **simple neural network**.

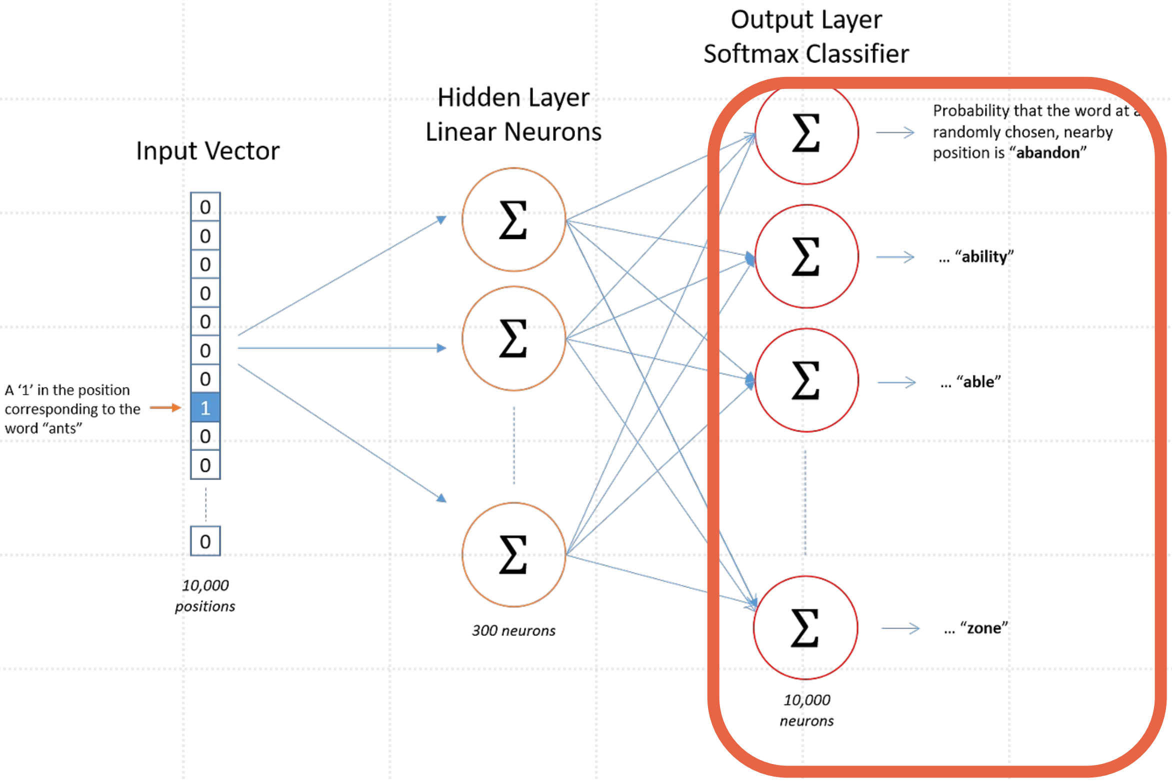
Input uses **one-hot encoding**: a vector of length “V” (size of vocabulary), which is all 0s except for a single 1.



https://towardsdatascience.com/skip-gram-neural-network-from-scratch-485f2e688238

Pt. 2b: Learning—the details.

- Technically, *word2vec* uses a **simple neural network**.



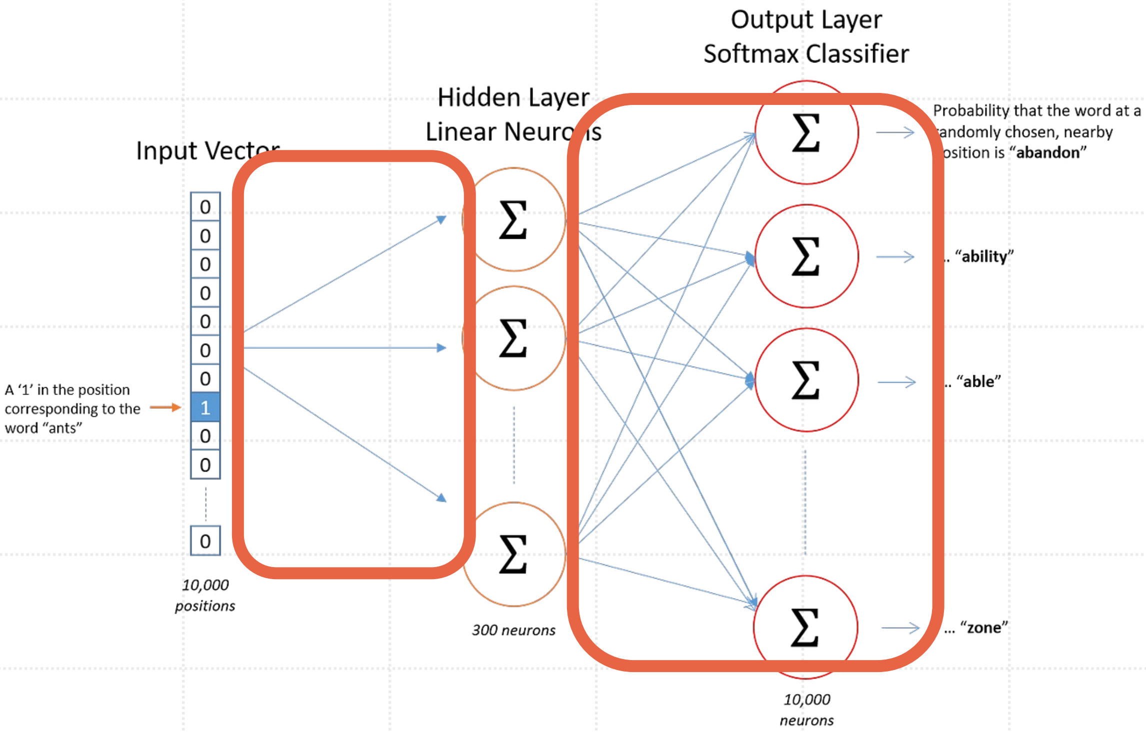
Output contains neurons for each possible word in vocabulary—learn **probability distribution** over these words.

https://towardsdatascience.com/skip-gram-neural-network-from-scratch-485f2e688238

Pt. 2b: Learning—the details.

- Technically, *word2vec* uses a **simple neural network**.

Learn **weights** from each input vector onto **hidden layer**.



The value of these weights is adjusted according to accuracy of predictions.

https://towardsdatascience.com/skip-gram-neural-network-from-scratch-485f2e688238

Pt. 2b: Learning—the details.

- Technically, *word2vec* uses a **simple neural network**.
- Skip-gram: goal is to predict **context** from a word.
- Skip-gram with negative sampling (SGNS): turns skip-gram into a binary classification task.
- Learning is done using **stochastic gradient descent (SGD).**
  - Keep changing embeddings until loss (error) is minimized.

What do the dimensions of these embeddings “represent”?

https://towardsdatascience.com/skip-gram-neural-network-from-scratch-485f2e688238

Pt. 2b: Learning—the details.

- Technically, *word2vec* uses a **simple neural network**.
- Skip-gram: goal is to predict **context** from a word.
- Skip-gram with negative sampling (SGNS): turns skip-gram into a binary classification task.
- Learning is done using **stochastic gradient descent (SGD).**
  - Keep changing embeddings until loss (error) is minimized.

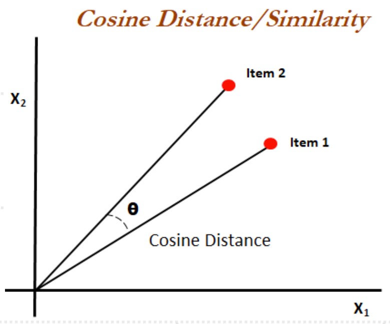
What do the dimensions of these embeddings “represent”?

They’re not directly interpretable! They don’t represent anything themselves.

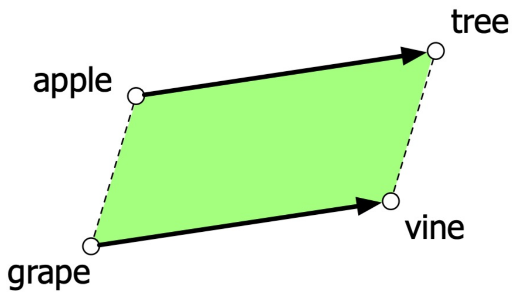
What are embeddings good for?

- The *word2vec* algorithm is used to produce **dense, static embeddings**.
- We can use these embeddings for <u>many</u> different tasks.

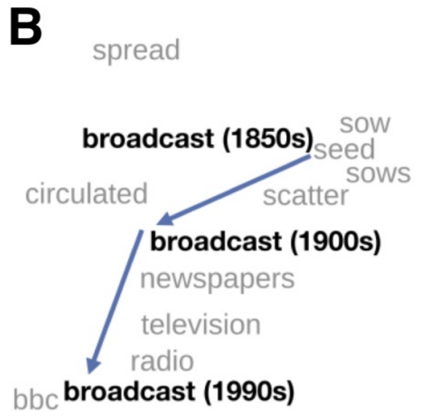
Measuring word similarity



Finding word analogies



Tracking change over time



Lecture plan

- Review: embeddings.
- Common architectures:
  - Feedforward language model.
  - Recurrent neural network.
  - Transformer architecture.
- Next time: LLMs in Python!

Feed-forward NLMs

A **feed-forward neural language model** uses a feed-forward neural network to assign probabilities to word w_t using representations of previous words.

Feed-forward NLMs

A **feed-forward neural language model** uses a feed-forward neural network to assign probabilities to word w_t using representations of previous words.

How might we use the neural network architecture for the **language modeling** task?

Feed-forward NLMs

A **feed-forward neural language model** uses a feed-forward neural network to assign probabilities to word w_t using representations of previous words.

- Neural network tries to predict w_t using context (previous words).
- neural network uses **embeddings** to represent those contextual words—rather than the words themselves.

How/why might using embedding representations help with the prediction task?

Feed-forward NLMs

And thanks for all the ____

context w_t

Feed-forward NLMs

And thanks for all the ____

$\underbrace{\textit{for all the}}_{\text{context}}$ $\underset{w_t}{\uparrow}$

To simplify, let's assume "context" is just the previous three words.

What type of *N*-gram model would this be?

Feed-forward NLMs

And thanks $\underbrace{\textit{for all the}}_{\text{context}}$ $\underset{\substack{\uparrow \\ w_t}}{\_\_\_\_}$

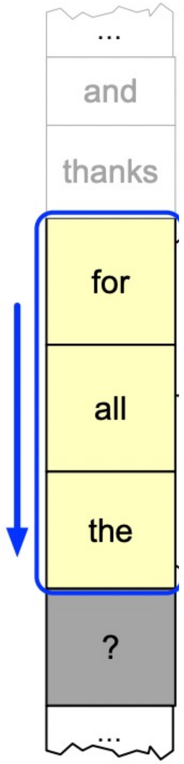
To simplify, let's assume "context" is just the previous three words.

Note: This is a **fixed context window**—we decide ahead of time how many words we want to include in the context.

Feed-forward NLMs

And thanks for all the ____

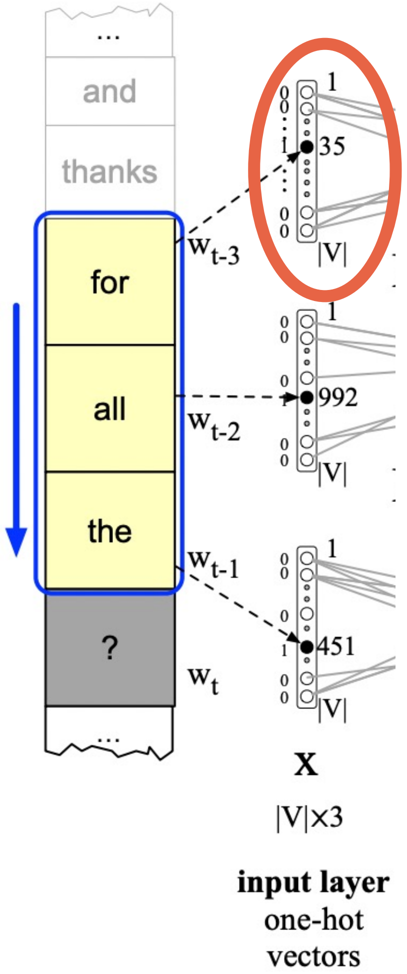
Words in the context are represented using **one-hot encodings**.



Feed-forward NLMs

And thanks for all the ____

Words in the context are represented using **one-hot encodings**.

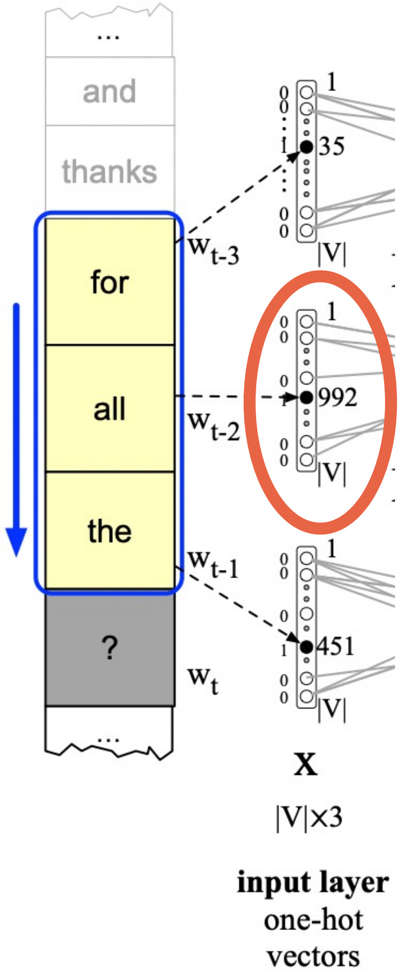


The word “for” is the 35th word in our vocabulary (V).

Feed-forward NLMs

And thanks for all the ____

Words in the context are represented using **one-hot encodings**.

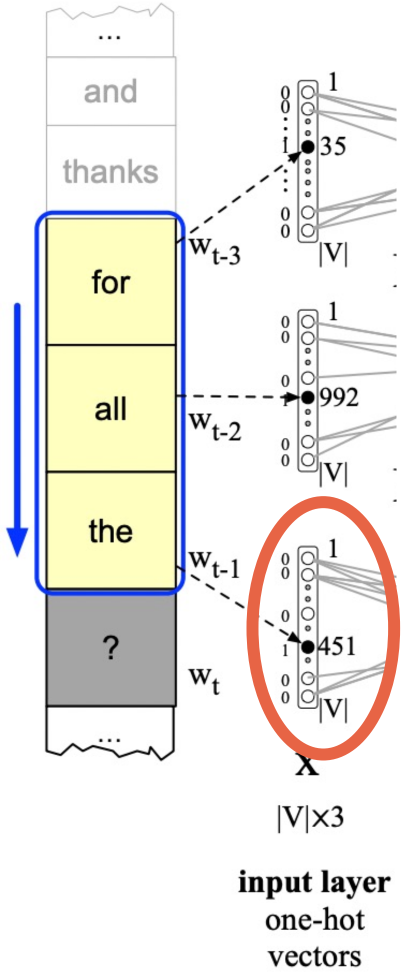


The word "all" is the 992nd word in our vocabulary (V).

Feed-forward NLMs

And thanks for all the ____

Words in the context are represented using **one-hot encodings**.



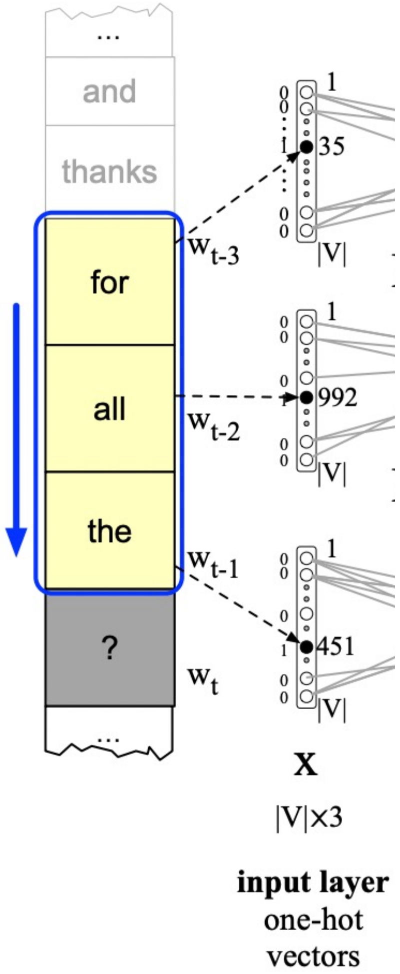
The word "the" is the 451st word in our vocabulary (V).

Feed-forward NLMs

And thanks for all the ____

Words in the context are represented using **one-hot encodings**.

Each one-hot encoding is multiplied by an **embedding matrix** (**E**).

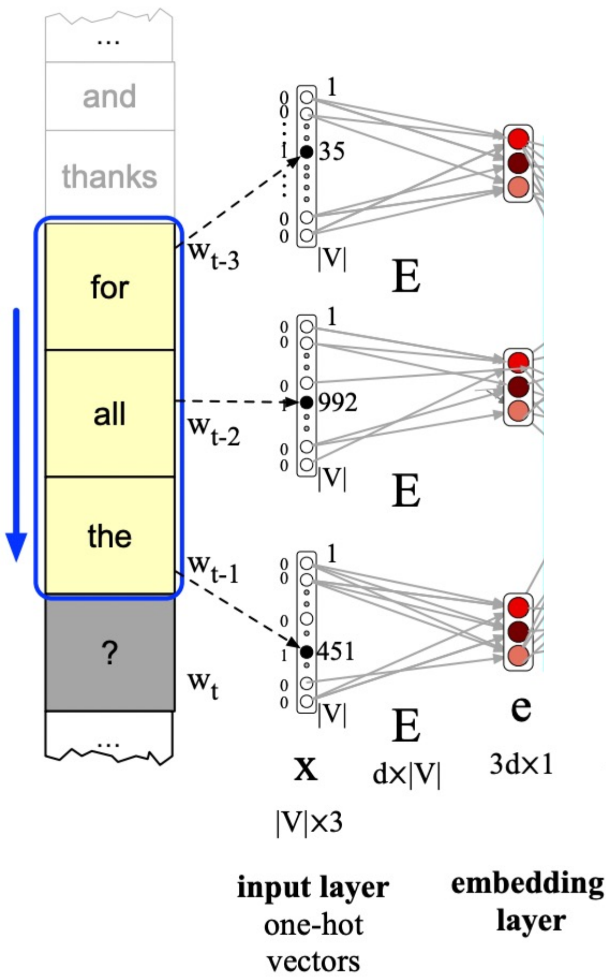


Feed-forward NLMs

And thanks for all the ____

Words in the context are represented using **one-hot encodings**.

Each one-hot encoding is multiplied by an **embedding matrix** (**E**).



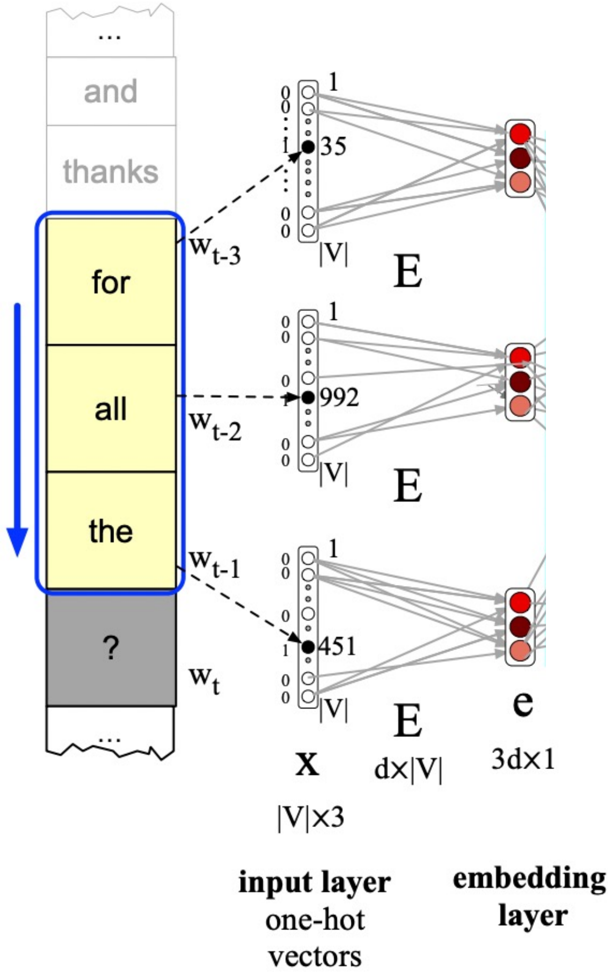
Feed-forward NLMs

And thanks for all the ____

Words in the context are represented using **one-hot encodings**.

Each one-hot encoding is multiplied by an **embedding matrix** (E).

Embeddings are combined, then multiplied by (learned) **weights** to obtain hidden layer activations.



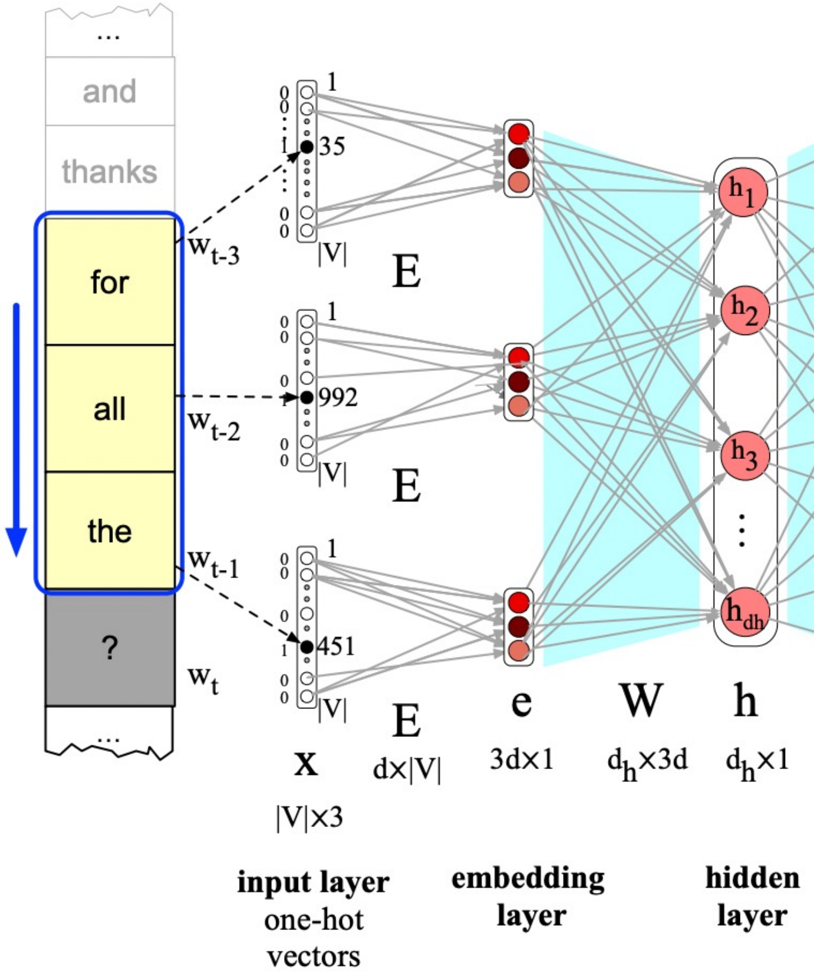
Feed-forward NLMs

And thanks for all the ____

Words in the context are represented using **one-hot encodings**.

Each one-hot encoding is multiplied by an **embedding matrix** (**E**).

Embeddings are combined, then multiplied by (learned) **weights** to obtain hidden layer activations.



These hidden units are <u>learned</u> <u>“representations”</u> of the immediate context that help with the prediction task.

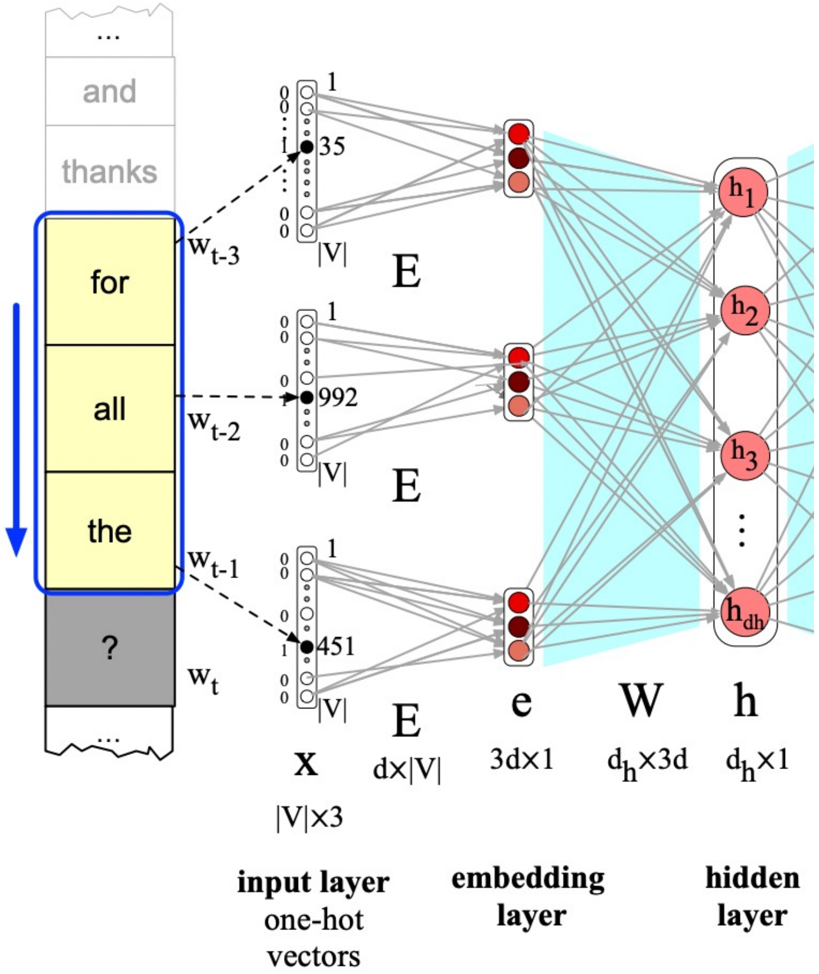
Feed-forward NLMs

And thanks for all the ____

Words in the context are represented using **one-hot encodings**.

Each one-hot encoding is multiplied by an **embedding matrix** (E).

Embeddings are combined, then multiplied by (learned) <u>**weights**</u> to obtain <u>hidden layer activations</u>.



Lots of work trying to **interpret** what these units learn…

Then, multiply hidden layer by weight matrix **U**, and apply **softmax**, to obtain probability distribution over next word.

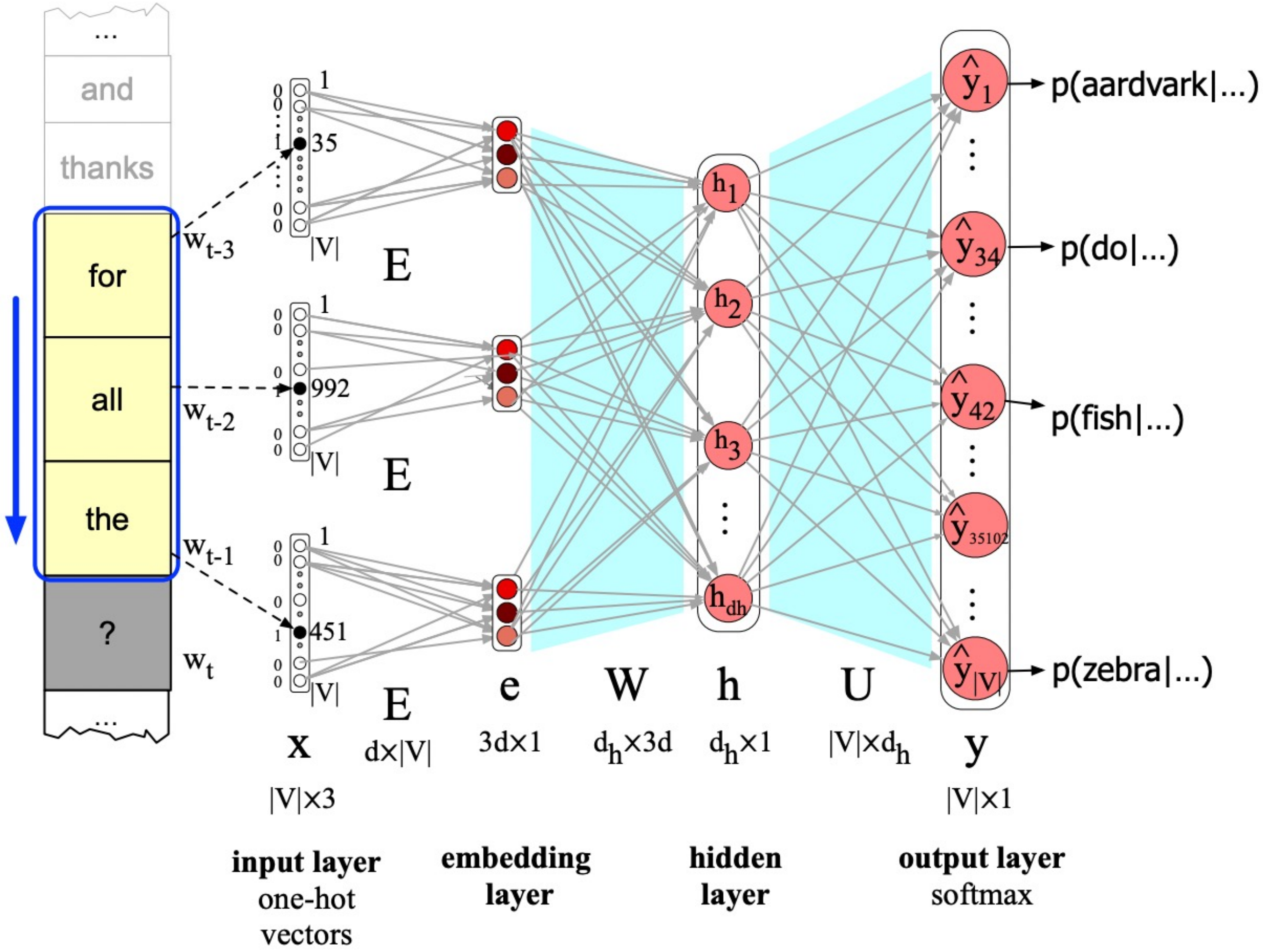
Feed-forward NLMs

And thanks for all the ____

Words in the context are represented using **one-hot encodings**.

Each one-hot encoding is multiplied by an **embedding matrix** (**E**).

Embeddings are combined, then multiplied by (learned) **weights** to obtain hidden layer activations.

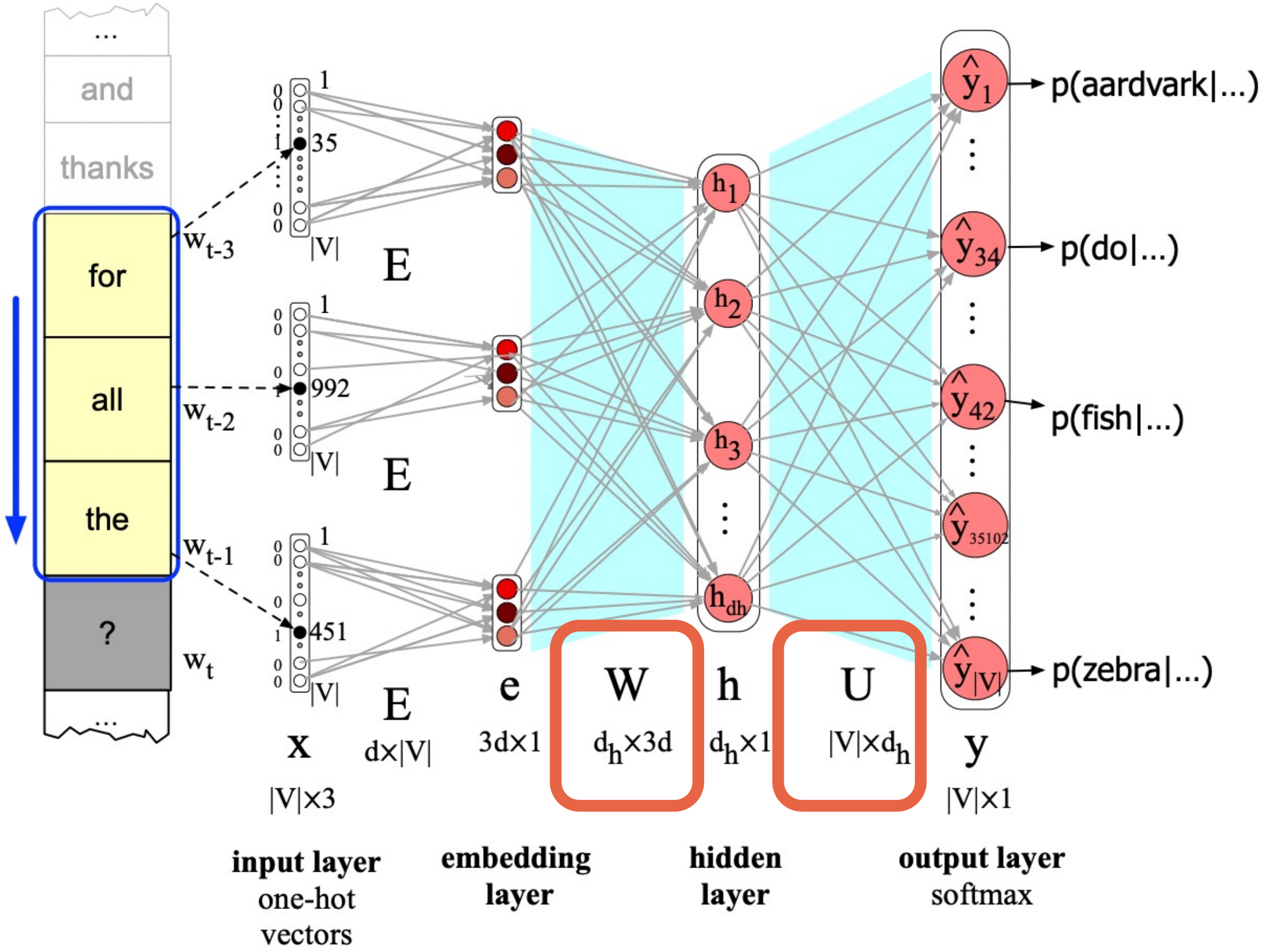


Feed-forward NLMs

And thanks for all the ____

All these weights are **learned**—i.e., update them to make better and better predictions.

But how does this actually work?



Training feed-forward NLMs

During *training*, the goal is to learn **parameters** ("weights") to make the predictions *Y'* as close as possible to actual values *Y*.

First, we define a **loss function**.

- <u>Higher</u> probability to true answer: <u>lower</u> loss
- <u>Lower</u> probability to true answer: <u>higher</u> loss

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\log \hat{\mathbf{y}}_c$$

When there's only a single "right answer", we can use the **negative log likelihood** assigned to the true answer.

Training feed-forward NLMs

During *training*, the goal is to learn **parameters** ("weights") to make the predictions *Y'* as close as possible to actual values *Y*.

First, we define a **loss function**.

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\log \hat{\mathbf{y}}_c$$

*And thanks for all the **fish***

What probability did we assign to **fish**, the true completion?

Basic intuition: a "good" model should've assigned 100% probability to *fish*!

Training feed-forward NLMs

During *training*, the goal is to learn **parameters** (“weights”) to make the predictions *Y’* as close as possible to actual values *Y*.

First, we define a **loss function**.

$$L_{CE}(\hat{\mathbf{y}}, \mathbf{y}) = -\log \hat{\mathbf{y}}_c$$

Update the **parameters** to *minimize* this loss.

Because neural networks have <u>many parameters</u>, this requires using a technique called “<u>**error back-propagation**</u>” (or “backprop”).

Lecture plan

- Review: embeddings.
- Common architectures:
  - Feedforward language model.
  - Recurrent neural network.
  - Transformer architecture.
- Next time: LLMs in Python!

Language unfolds over time

- Language is a <u>temporal</u> phenomenon.
- When we process (hear, see, read) language, it <u>unfolds</u> bit-by-bit.

This

Language unfolds over time

- Language is a <u>temporal</u> phenomenon.
- When we process (hear, see, read) language, it <u>unfolds</u> bit-by-bit.

class

Language unfolds over time

- Language is a temporal phenomenon.
- When we process (hear, see, read) language, it unfolds bit-by-bit.

is

Language unfolds over time

- Language is a temporal phenomenon.
- When we process (hear, see, read) language, it unfolds bit-by-bit.

about

Language unfolds over time

- Language is a temporal phenomenon.
- When we process (hear, see, read) language, it unfolds bit-by-bit.

language

Language unfolds over time

- Language is a temporal phenomenon.
- When we process (hear, see, read) language, it unfolds bit-by-bit.

models

Language unfolds over time

- Language is a temporal phenomenon.
- When we process (hear, see, read) language, it unfolds bit-by-bit.

and

Language unfolds over time

- Language is a <u>temporal</u> phenomenon.
- When we process (hear, see, read) language, it <u>unfolds</u> bit-by-bit.

cognitive

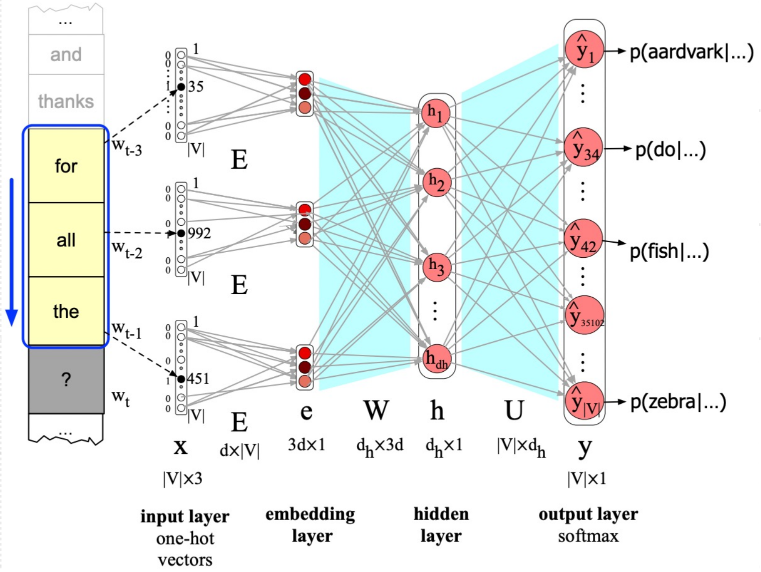
Language unfolds over time

- Language is a <u>temporal</u> phenomenon.
- When we process (hear, see, read) language, it <u>unfolds</u> bit-by-bit.

science.

Language unfolds over time

- Language is a temporal phenomenon.
- When we process (hear, see, read) language, it unfolds bit-by-bit.
- Yet the feed-forward models we've discussed use a fixed window to represent context.



Even though "for all the" unfolds over time, this model has simultaneous access to each word at the same time.

This isn't really how language works!

Also presents other challenges—how big should this window be?

Language unfolds over time

- Language is a temporal phenomenon.
- When we process (hear, see, read) language, it unfolds bit-by-bit.
- Yet the feed-forward models we've discussed use a fixed window to represent context.
- Ideally, we could incorporate the temporal nature of language into the very structure of our neural network.
- This is what **recurrent neural networks (RNNs)** aim to do.
- **"Recurrent" connections** are a way to model the role of context without needing fixed-size windows.

Finding Structure in Time

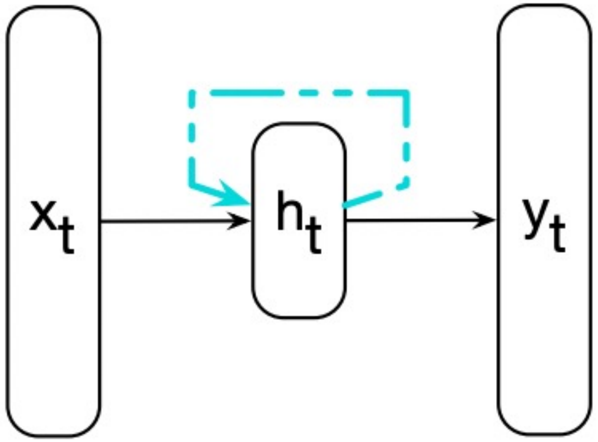
JEFFREY L. ELMAN

University of California, San Diego

Jeff Elman

Recurrent neural networks

A **recurrent neural network (RNN)** is any network with a "cycle" in its connections, i.e., such that the value of some unit depends (directly or indirectly) on its *earlier activity*.

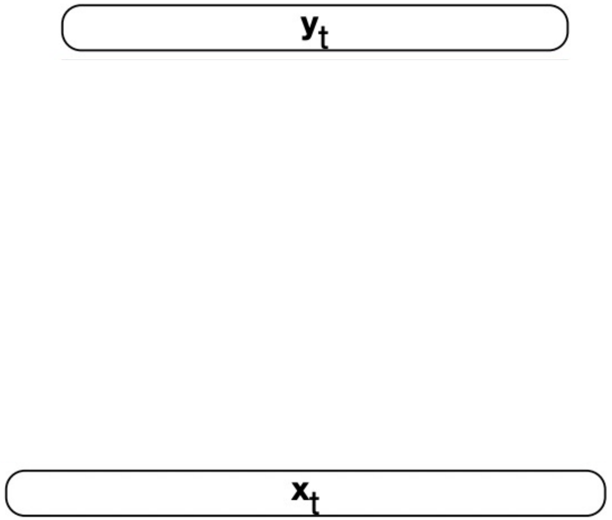


- The **Elman net** (1990) is one very influential implementation.
- In addition to *feed-forward weights*, the hidden layer contains **recurrent connections** (i.e., to itself).
- Sequences are presented **one unit (e.g., word) at a time**.
- This recurrent connection acts as a kind of "**memory**", connecting current state to previous states.
- No need for fixed context windows!

Forward inference in RNNs

<u>"Forward inference"</u> refers to mapping an input **(x)** to a predicted output **(y).**

Output at time *t*

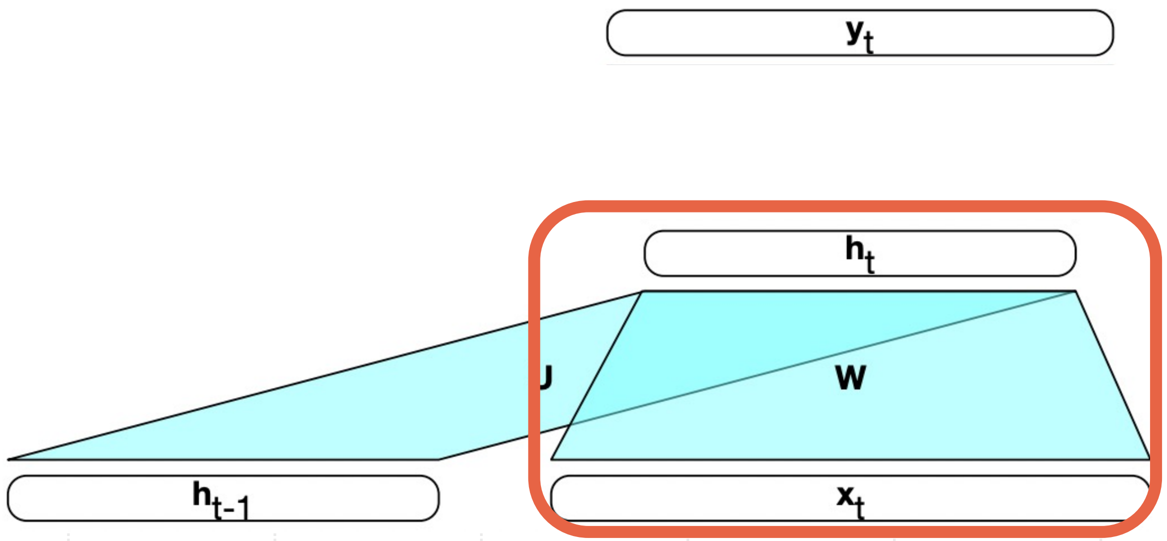


Input at time *t*

Forward inference in RNNs

<u>"Forward inference"</u> refers to mapping an input **(x)** to a predicted output **(y).**

Output at time *t*

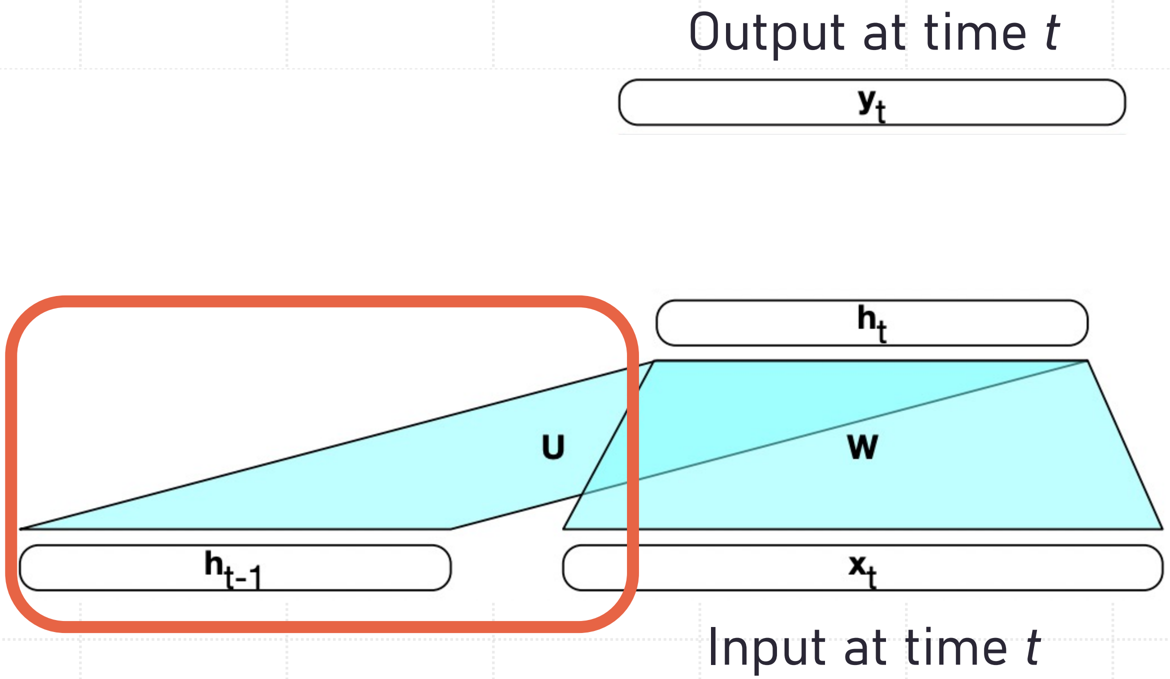


Input at time *t*

- Multiply **input** by weight matrix **W.**

Forward inference in RNNs

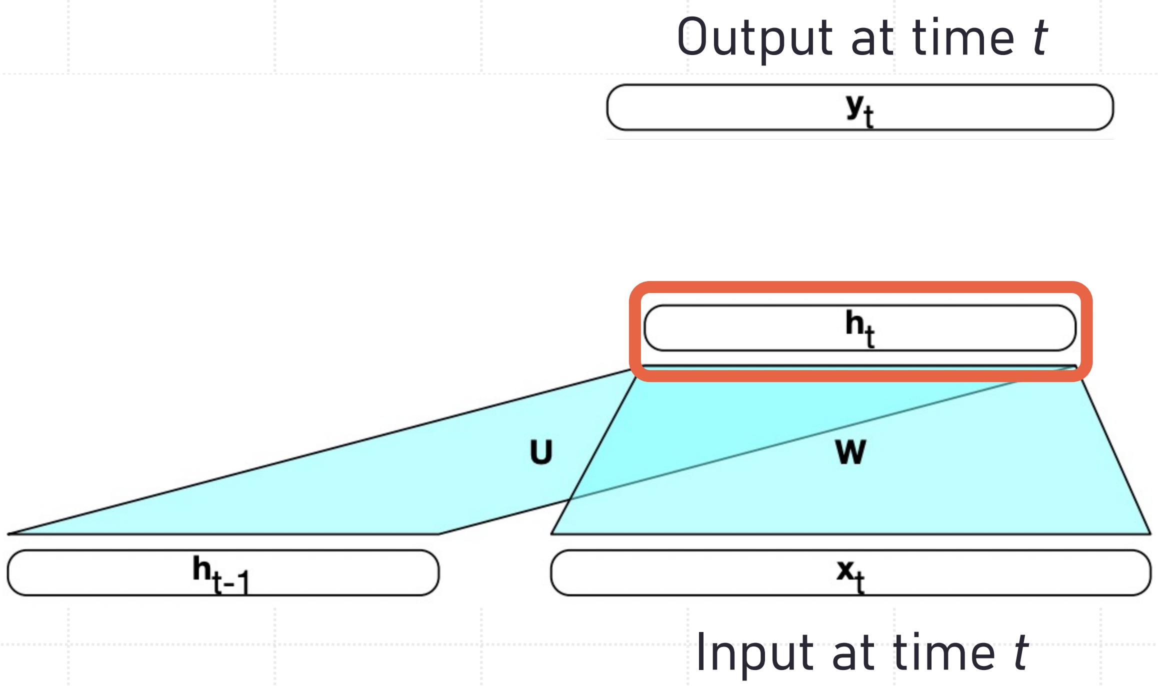
<u>"Forward inference"</u> refers to mapping an input **(x)** to a predicted output **(y).**



- Multiply **input** by weight matrix **W.**
- Multiply *previous* hidden layer activation ($\boldsymbol{h_{t-1}}$) by weight matrix **U.**

Forward inference in RNNs

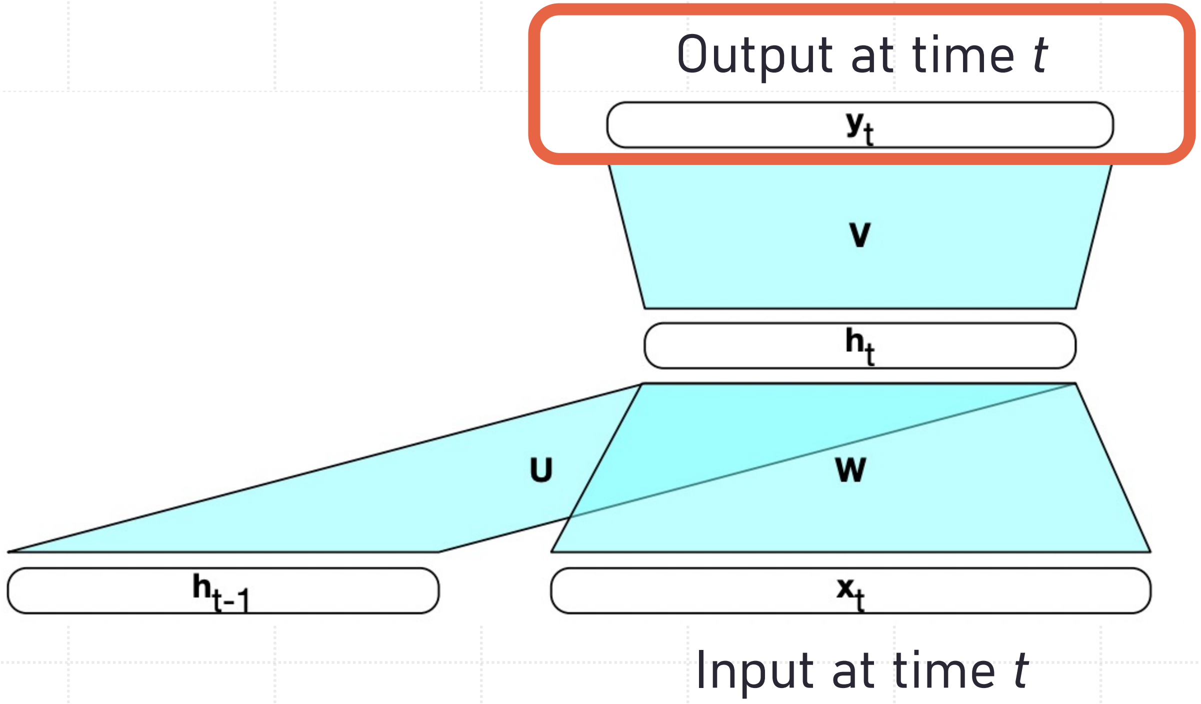
<u>"Forward inference"</u> refers to mapping an input **(x)** to a predicted output **(y).**



- Multiply **input** by weight matrix **W.**
- Multiply *previous* hidden layer activation (h_{t-1}) by weight matrix **U.**
- Add these together and pass through **activation function**.

Forward inference in RNNs

"Forward inference" refers to mapping an input **(x)** to a predicted output **(y).**



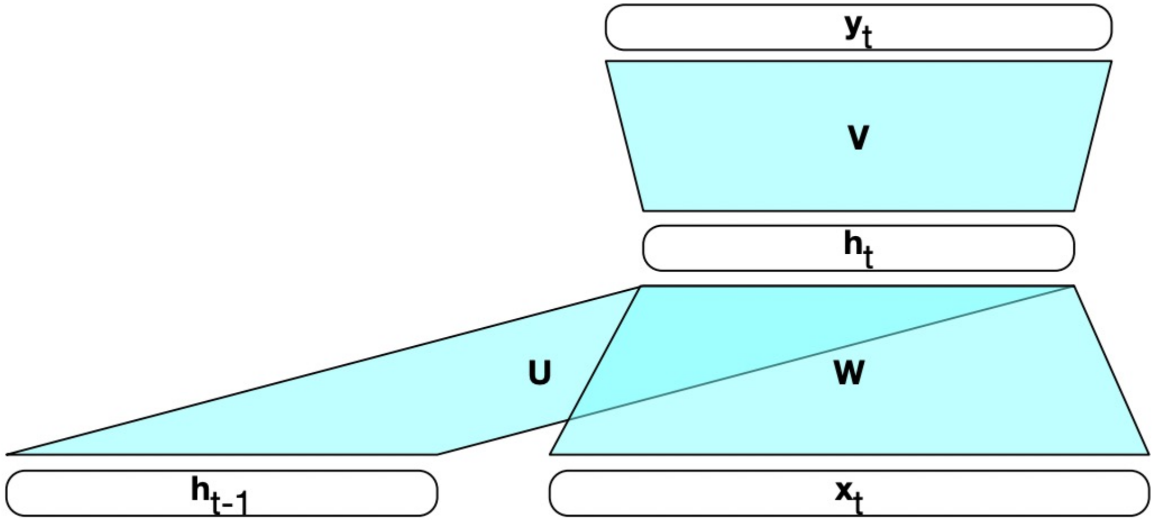
- Multiply **input** by weight matrix **W.**
- Multiply *previous* hidden layer activation (h_{t-1}) by weight matrix **U.**
- Add these together and pass through **activation function**.
- Multiply $\boldsymbol{h_t}$ by weight matrix **V**.
- Apply **softmax** to obtain output probabilities.

What's <u>similar</u> to a feed-forward network?
What's <u>different</u>?

Forward inference in RNNs

"Forward inference" refers to mapping an input **(x)** to a predicted output **(y).**

Output at time *t*



Input at time *t*

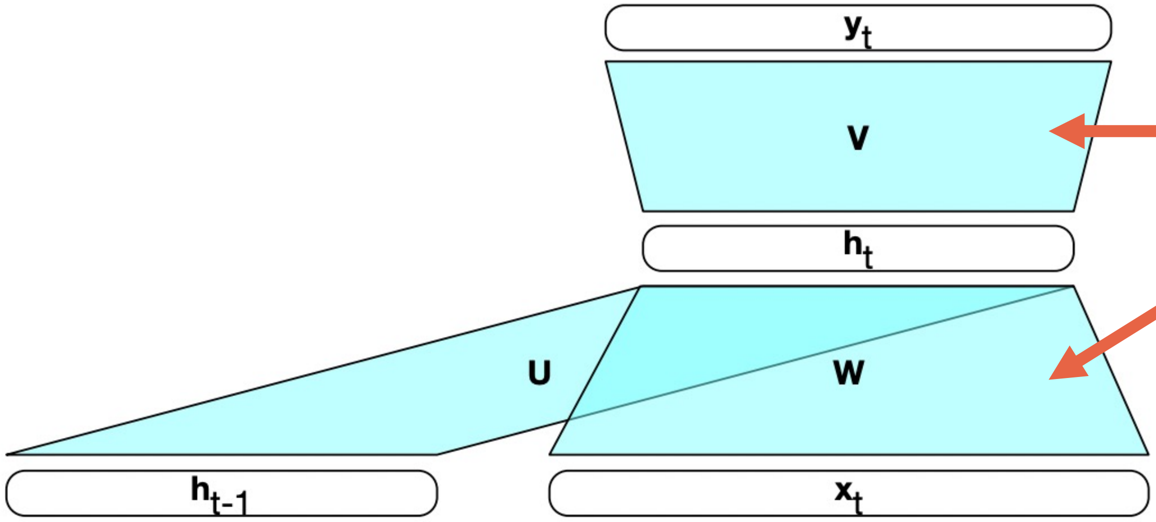
- Multiply **input** by weight matrix **W.**
- Multiply *previous* hidden layer activation (h_{t-1}) by weight matrix **U.**
- Add these together and pass through **activation function**.
- Multiply $\boldsymbol{h_t}$ by weight matrix **V**.
- Apply **softmax** to obtain output probabilities.

What's similar to a feed-forward network?
What's different?

Forward inference in RNNs

"Forward inference" refers to mapping an input **(x)** to a predicted output **(y).**

Output at time *t*



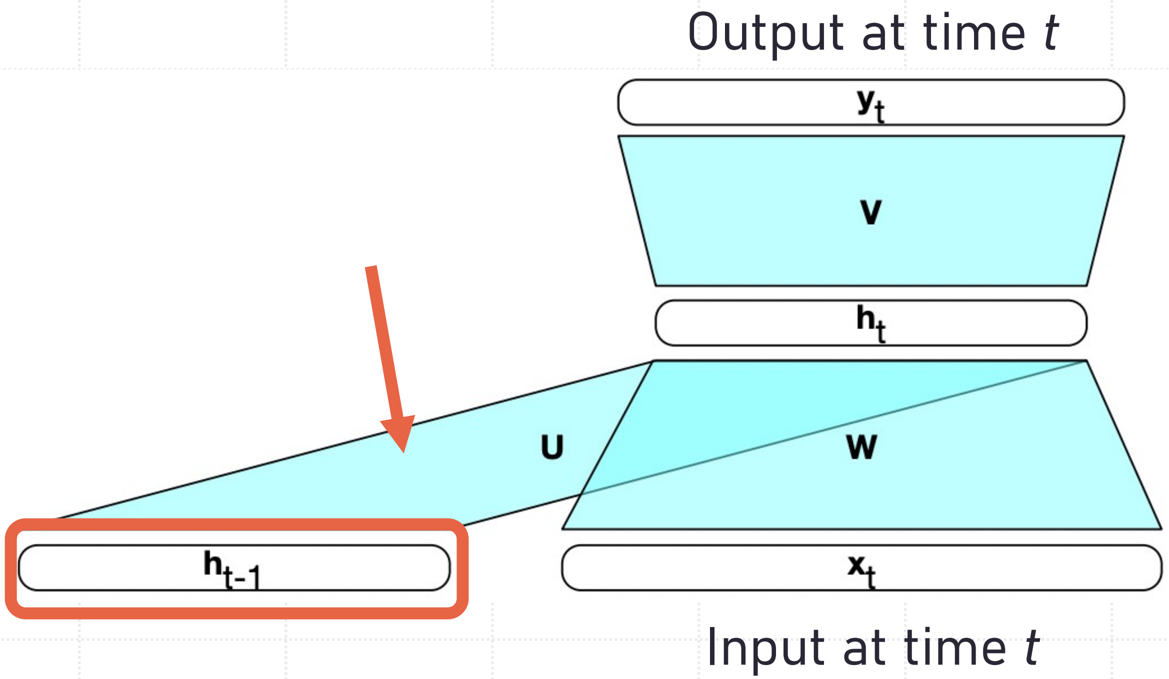
Input at time *t*

Like a feed-forward network, we multiply $\mathbf{x_t}$ and $\mathbf{h_t}$ by weight matrices (**W** and **V**) to obtain hidden and output activations.

What's similar to a feed-forward network?
What's different?

Forward inference in RNNs

"Forward inference" refers to mapping an input **(x)** to a predicted output **(y).**



Like a feed-forward network, we multiply $\mathbf{x_t}$ and $\mathbf{h_t}$ by weight matrices (**W** and **V**) to obtain hidden and output activations.

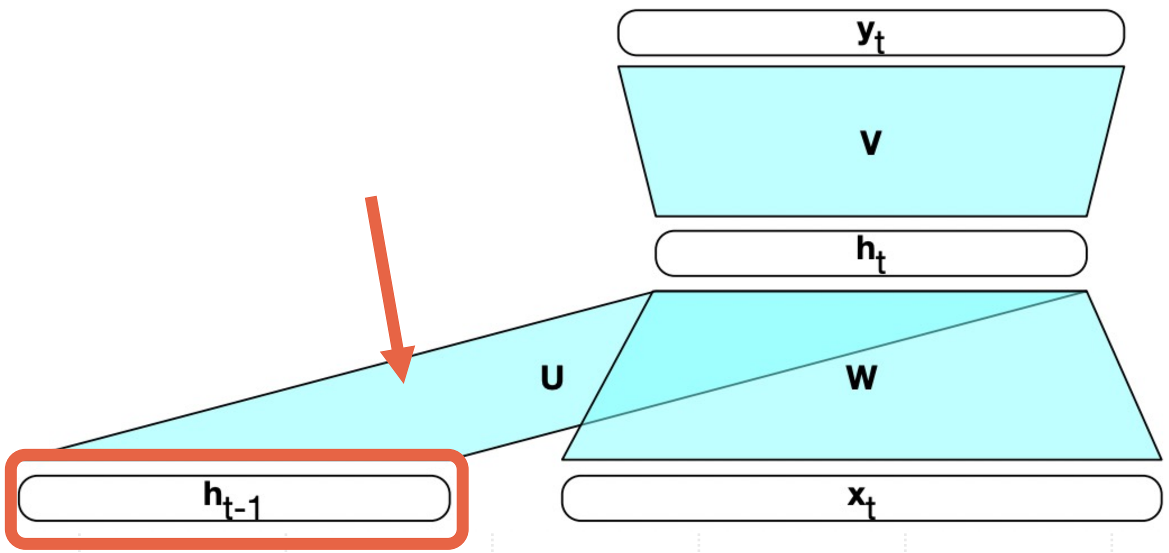
Unlike a feed-forward network, we "remember" the previous state's activations **($\mathbf{h_{t-1}}$)**, and incorporate that into calculation of $\mathbf{h_t}$.

We can also represent this algorithmically
(e.g., in pseudo-code).

Forward inference in RNNs

"Forward inference" refers to mapping an input **(x)** to a predicted output **(y).**

Output at time *t*



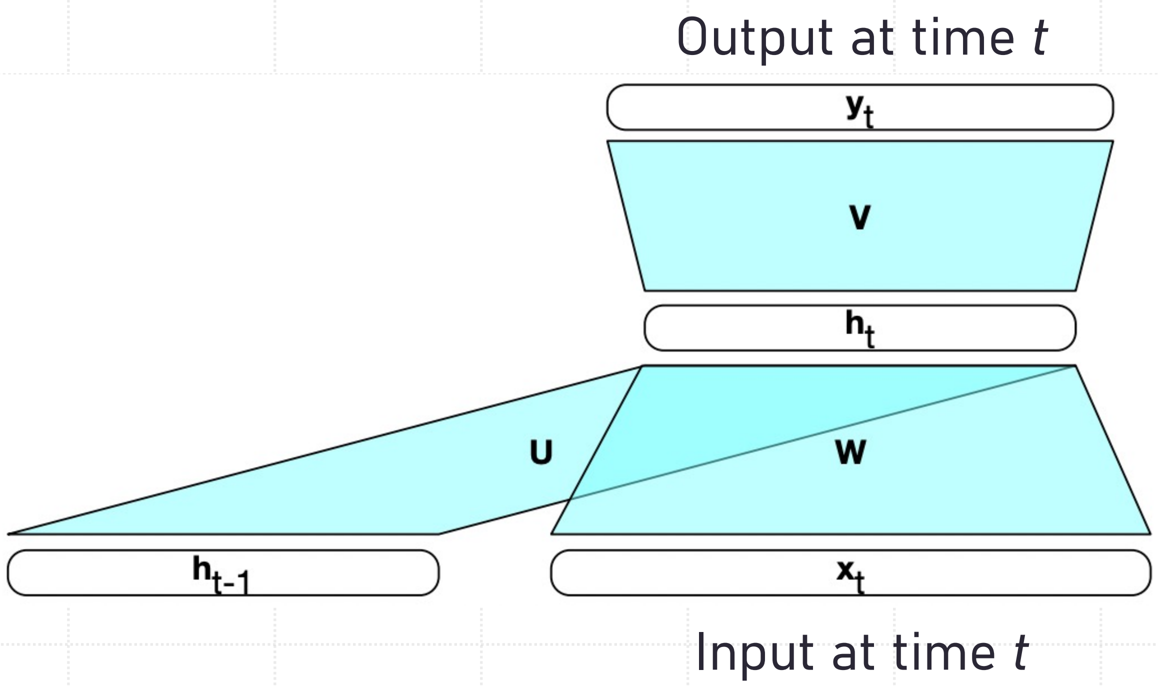
Input at time *t*

Like a feed-forward network, we multiply $\mathbf{x_t}$ and $\mathbf{h_t}$ by weight matrices (**W** and **V**) to obtain hidden and output activations.

Unlike a feed-forward network, we "remember" the previous state's activations **($\mathbf{h_{t-1}}$)**, and incorporate that into calculation of $\mathbf{h_t}$.

Forward inference in RNNs

<u>"Forward inference"</u> refers to mapping an input **(x)** to a predicted output **(y).**



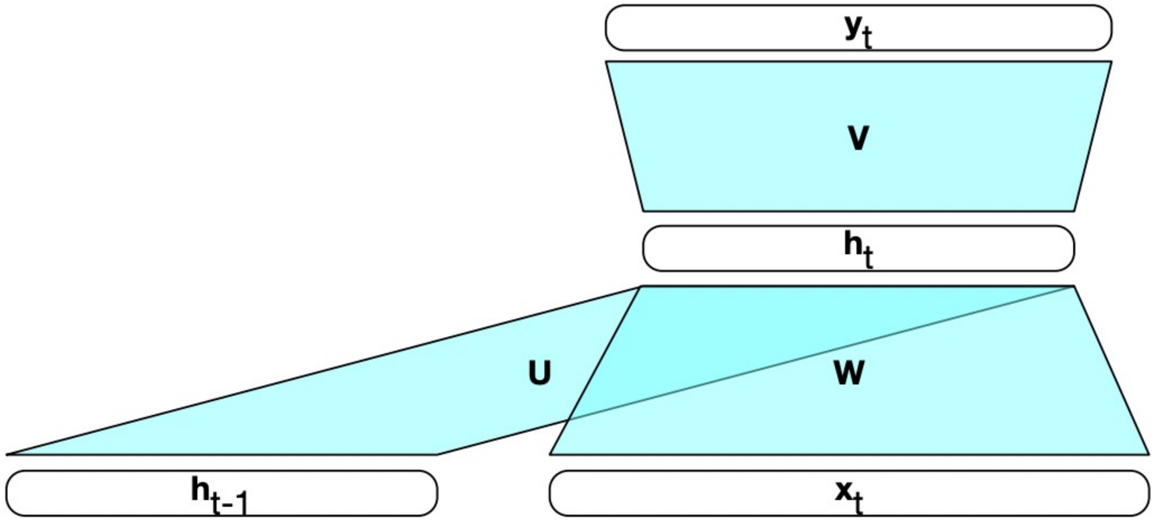
function FORWARDRNN($\mathbf{x}$, *network*) **returns** output sequence $\mathbf{y}$

$\mathbf{h}_0 \leftarrow 0$
for $i \leftarrow 1$ **to** LENGTH($\mathbf{x}$) **do**
  $\mathbf{h}_i \leftarrow g(\mathbf{U}\mathbf{h}_{i-1} + \mathbf{W}\mathbf{x}_i)$
  $\mathbf{y}_i \leftarrow f(\mathbf{V}\mathbf{h}_i)$
return y

Forward inference in RNNs

"Forward inference" refers to mapping an input **(x)** to a predicted output **(y).**

Output at time *t*



Input at time *t*

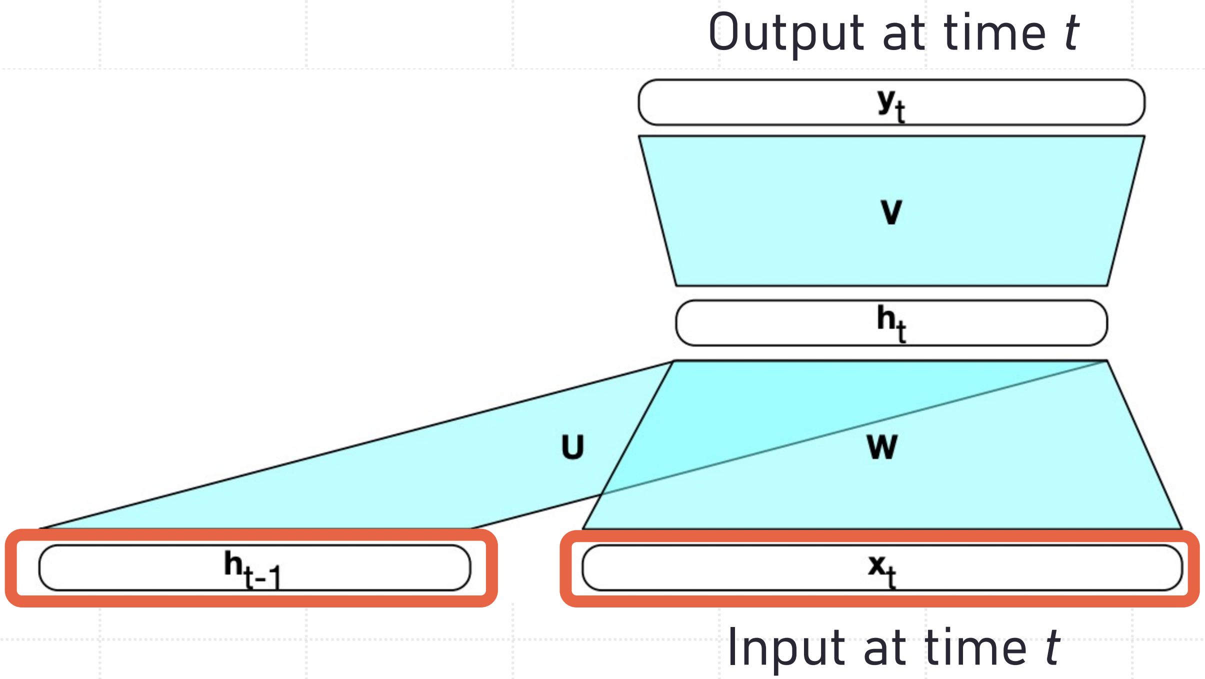
function FORWARDRNN($\mathbf{x}$, *network*) **returns** output sequence $\mathbf{y}$

$\mathbf{h}_0 \leftarrow 0$
for $i \leftarrow 1$ **to** LENGTH($\mathbf{x}$) **do**
  $\mathbf{h}_i \leftarrow g(\mathbf{U}\mathbf{h}_{i-1} + \mathbf{W}\mathbf{x}_i)$
  $\mathbf{y}_i \leftarrow f(\mathbf{V}\mathbf{h}_i)$
return y

Process input **incrementally**.

Forward inference in RNNs

<u>"Forward inference"</u> refers to mapping an input **(x)** to a predicted output **(y).**



function FORWARDRNN($\mathbf{x}$, *network*) **returns** output sequence $\mathbf{y}$

$\mathbf{h}_0 \leftarrow 0$
for $i \leftarrow 1$ **to** LENGTH($\mathbf{x}$) **do**
  $\boxed{\mathbf{h}_i \leftarrow g(\mathbf{U}\mathbf{h}_{i-1} + \mathbf{W}\mathbf{x}_i)}$
  $\mathbf{y}_i \leftarrow f(\mathbf{V}\mathbf{h}_i)$
return y

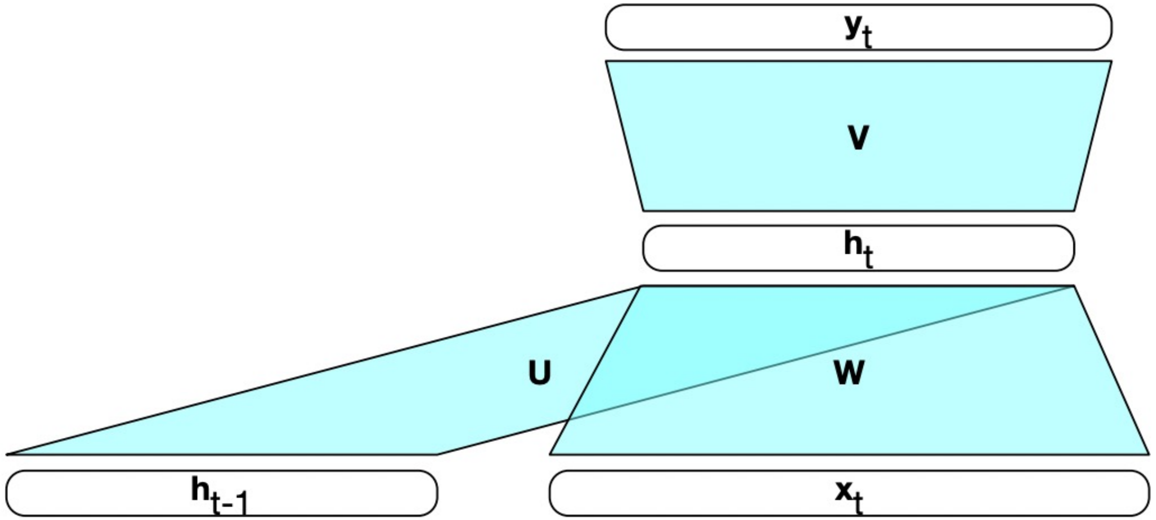
Current hidden state is a function of current **input** and **previous** hidden state.

It is also helpful to visualize this by **"unrolling"** the network across time.

Forward inference in RNNs

"Forward inference" refers to mapping an input **(x)** to a predicted output **(y).**

Output at time *t*



Input at time *t*

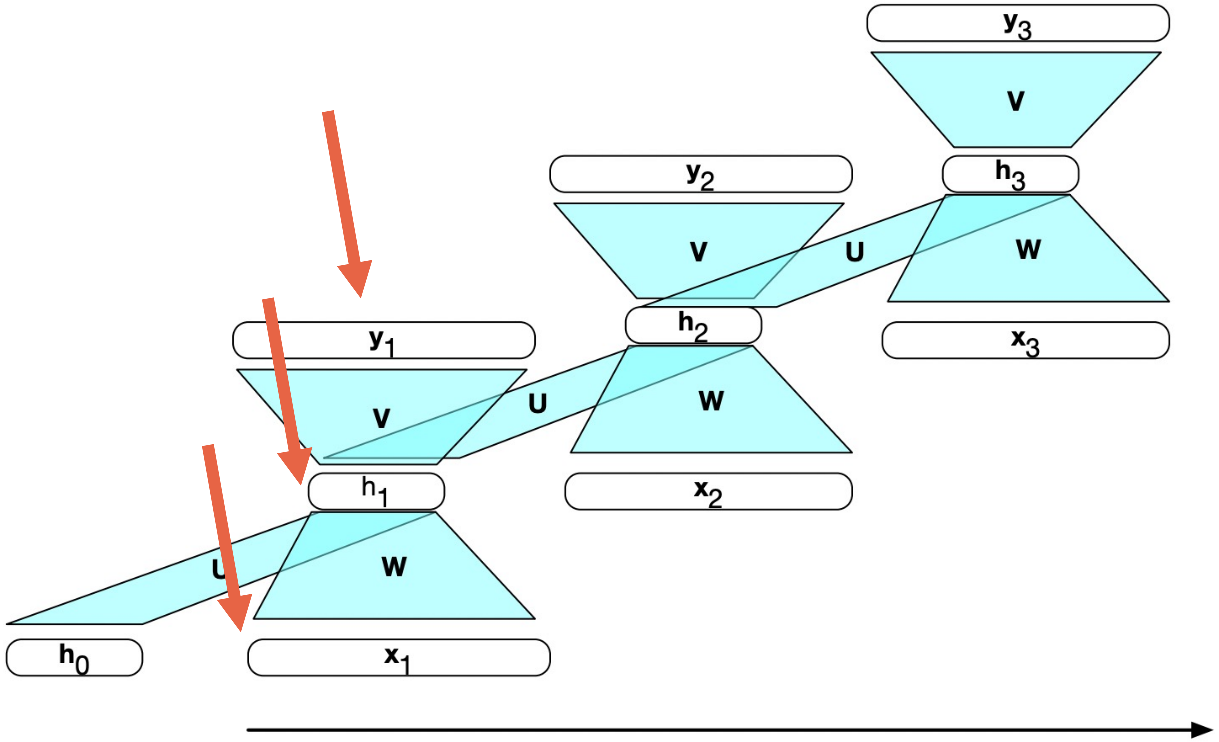
function FORWARDRNN($\mathbf{x}$, *network*) **returns** output sequence $\mathbf{y}$

$\mathbf{h}_0 \leftarrow 0$
for $i \leftarrow 1$ **to** LENGTH($\mathbf{x}$) **do**
  $\mathbf{h}_i \leftarrow g(\mathbf{U}\mathbf{h}_{i-1} + \mathbf{W}\mathbf{x}_i)$
  $\boxed{\mathbf{y}_i \leftarrow f(\mathbf{V}\mathbf{h}_i)}$
return y

Current prediction is a function of current hidden state.

Forward inference in RNNs

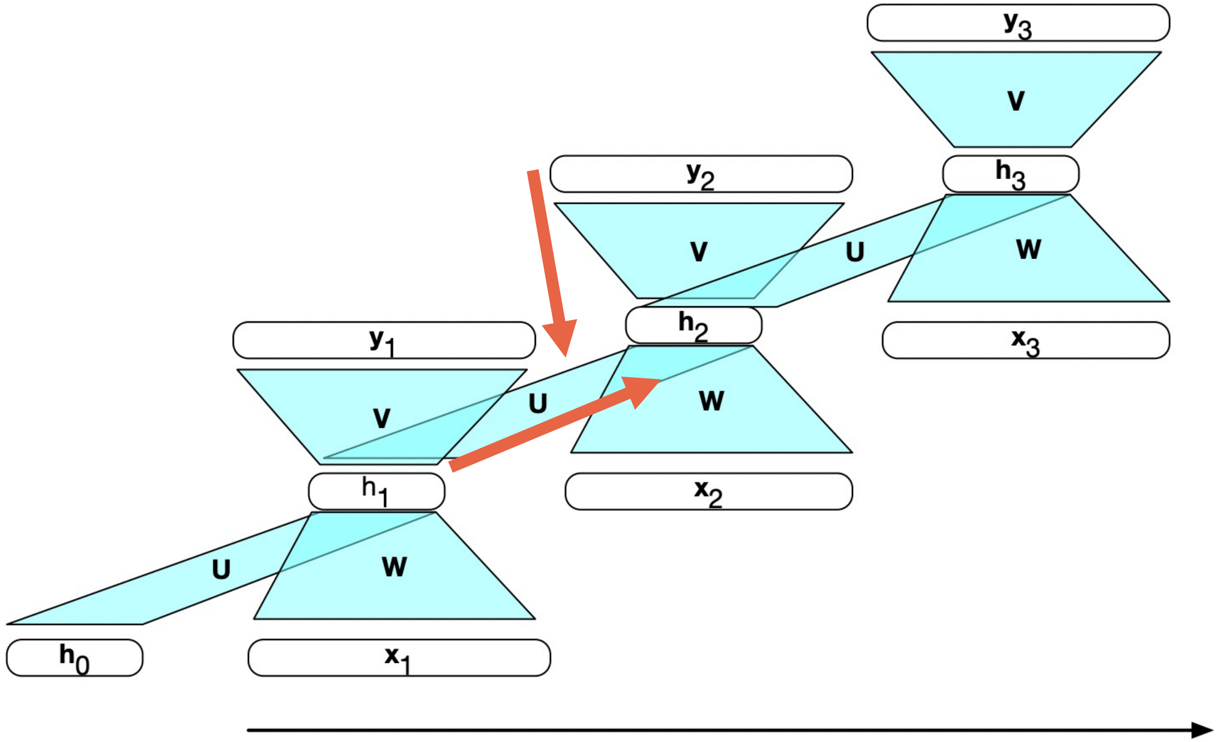
<u>"Forward inference"</u> refers to mapping an input **(x)** to a predicted output **(y).**



For each input token, we obtain a <u>predicted output</u>—and also a <u>hidden state</u>.

Forward inference in RNNs

<u>"Forward inference"</u> refers to mapping an input **(x)** to a predicted output **(y).**

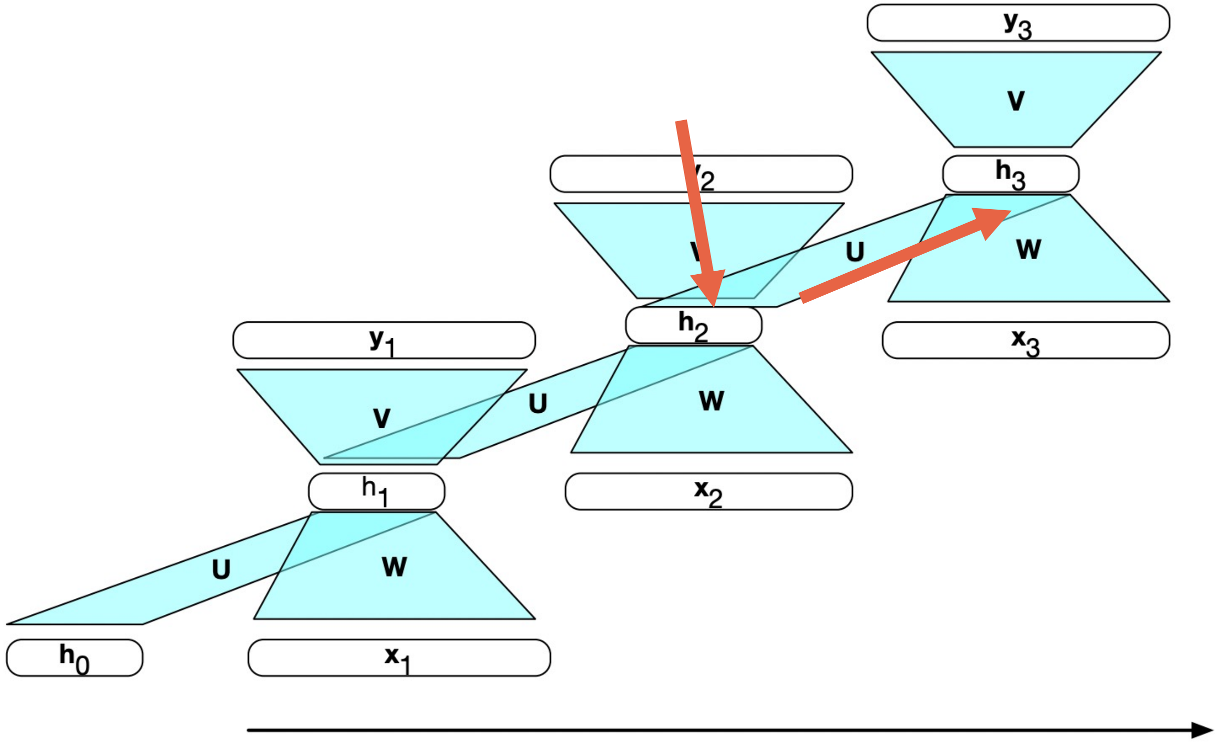


For each input token, we obtain a <u>predicted output</u>—and also a <u>hidden state</u>.

These <u>hidden states</u> are used to influence hidden states at the *next time step*.

Forward inference in RNNs

<u>"Forward inference"</u> refers to mapping an input **(x)** to a predicted output **(y).**

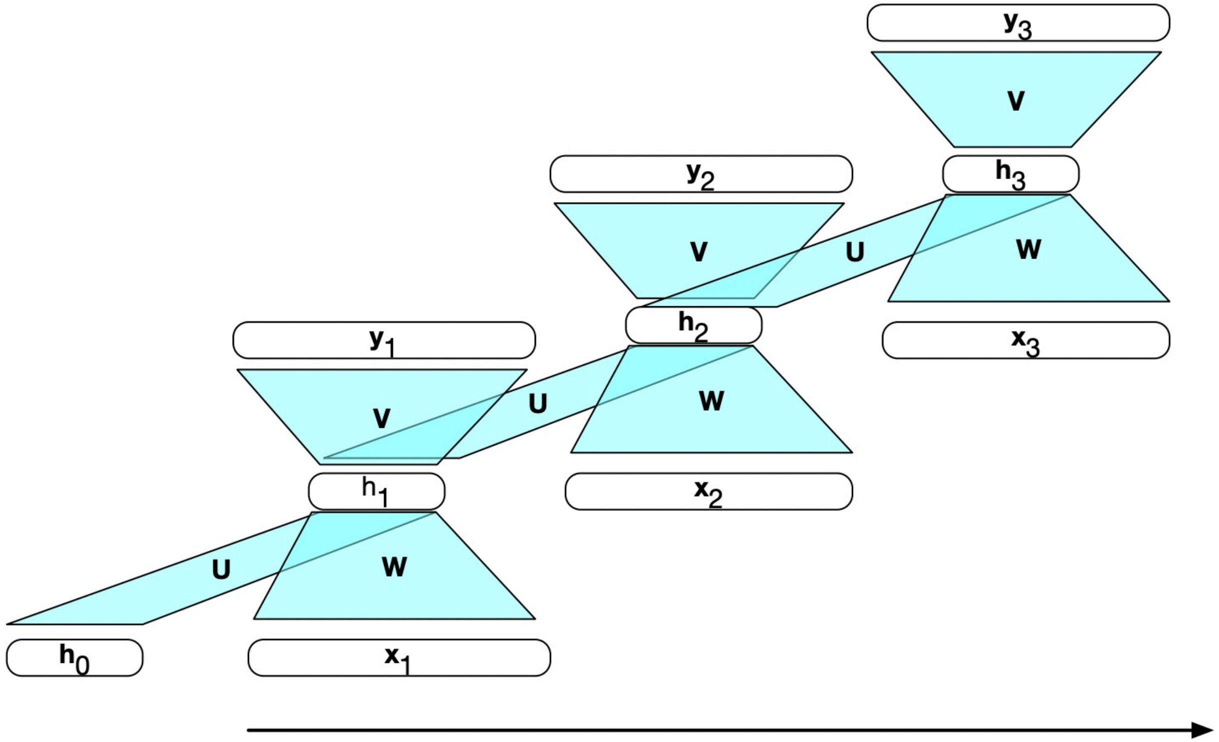


For each input token, we obtain a <u>predicted output</u>—and also a <u>hidden state</u>.

These <u>hidden states</u> are used to influence hidden states at the *next time step*.

Forward inference in RNNs

<u>"Forward inference"</u> refers to mapping an input **(x)** to a predicted output **(y).**



For each input token, we obtain a <u>predicted output</u>—and also a <u>hidden state</u>.

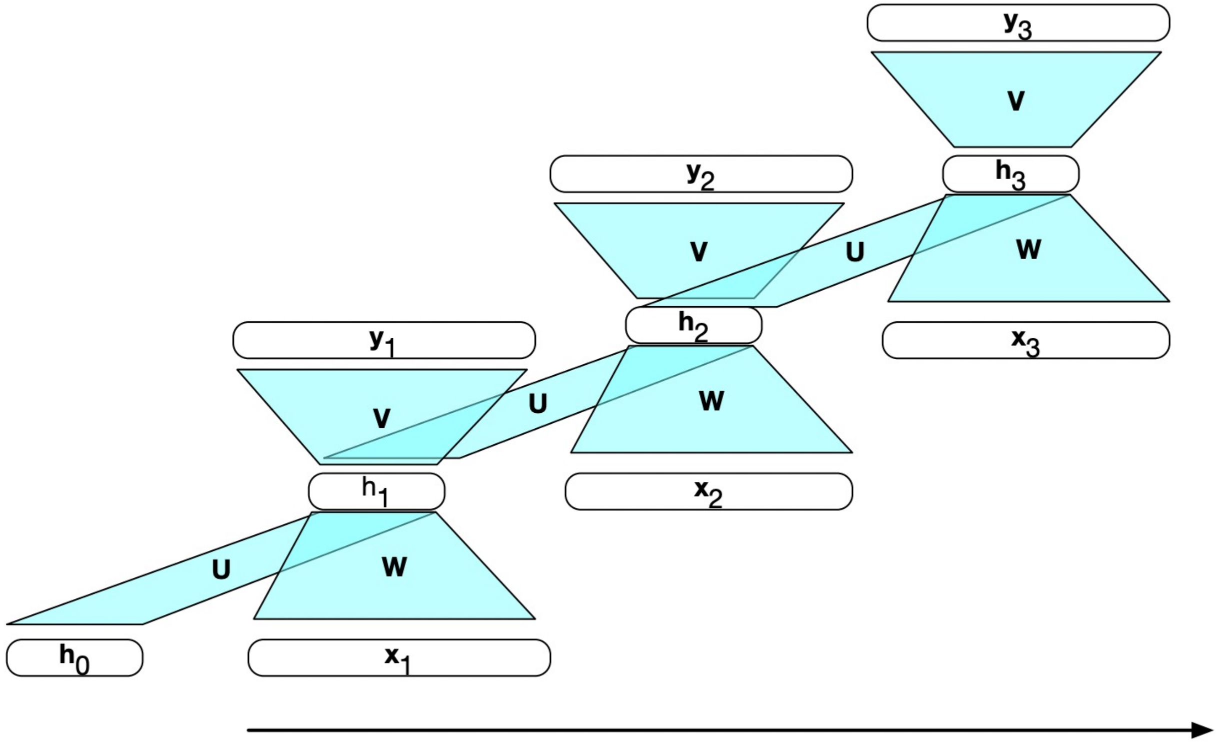
These <u>hidden states</u> are used to influence hidden states at the *next time step*.

Weight matrices stay the same—but $\mathbf{h}_{t-1}$ will change in <u>context</u>.

Note: applying backprop will require "unrolling" network, because updates should include the effect on *future predictions*.

Forward inference in RNNs

"Forward inference" refers to mapping an input **(x)** to a predicted output **(y).**



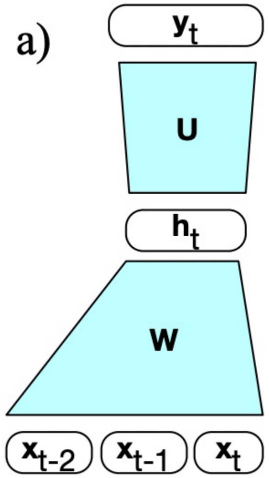
A conceptually elegant way to capture the **temporal** structure of language.

Also captures the effect of context—**"context" is just the state of the system at time *t***.

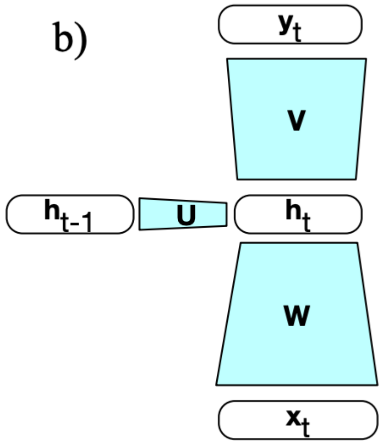
RNNs as language models

- RNNs are naturally suited to modeling language—and avoid some issues of feed-forward neural networks.

Feed-forward LM



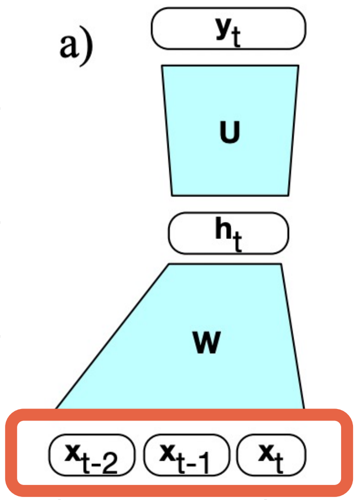
RNN LM



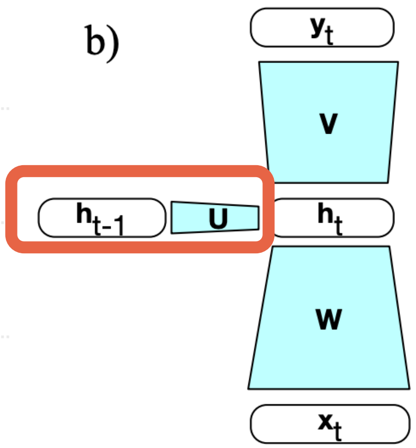
RNNs as language models

- RNNs are naturally suited to modeling language—and avoid some issues of feed-forward neural networks.

Feed-forward LM



RNN LM

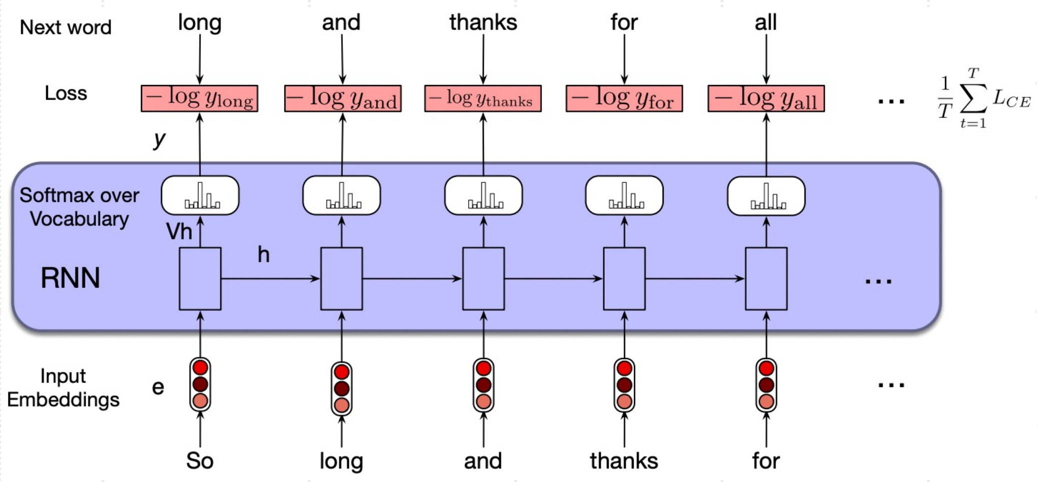


- Doesn't require **fixed context window**.
- "Context" is captured by $\mathbf{h}_{t-1}$.

In both cases, "output" is <u>probability distribution</u> over upcoming word token.

RNNs as language models

- RNNs are naturally suited to modeling language—and avoid some issues of feed-forward neural networks.
- Like other LMs, RNNs can be trained using **self-supervision** (language acts as its own training signal).



At each time step, we compute **loss**—the negative log probability assigned to the correct word y_t

We also "force" input to be the correct sequence (ignoring model's previous predictions)—this is called **teacher forcing**.

Using RNNs as generative models

A **generative language model** uses the probabilities assigned to upcoming tokens to actually generate novel sequences of text.

Intuitively, how might this work?

Using RNNs as generative models

A **generative language model** uses the probabilities assigned to upcoming tokens to actually generate novel sequences of text.

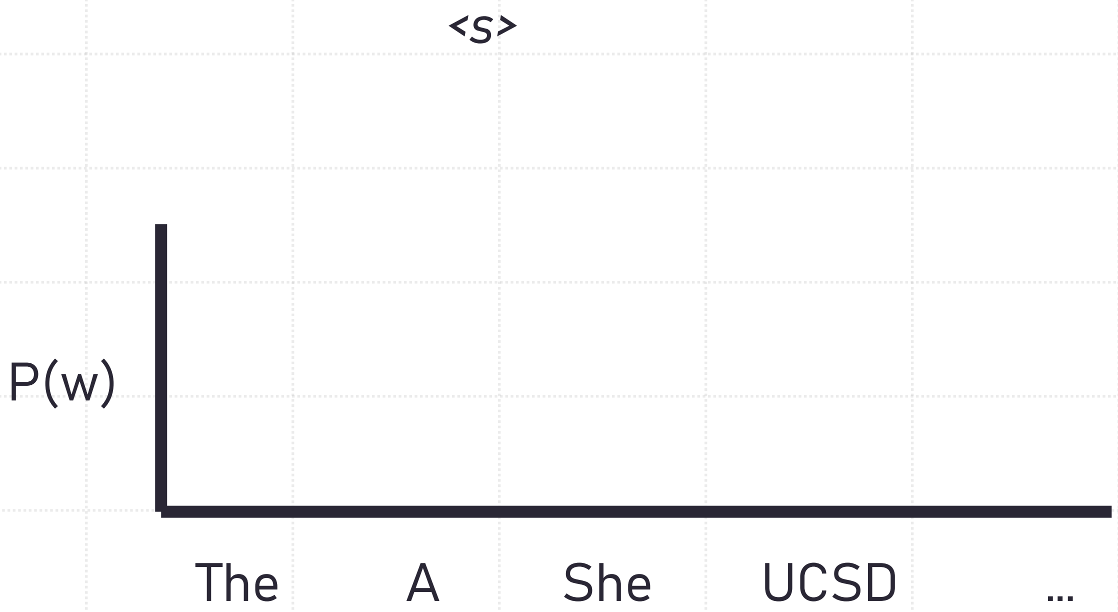
- Start with an initial token/sequence.

<s>

Using RNNs as <u>generative models</u>

A **<u>generative language model</u>** uses the probabilities assigned to upcoming tokens to actually <u>generate</u> novel sequences of text.

- Start with an initial token/sequence.
- Run through RNN, obtain probabilities over next word.

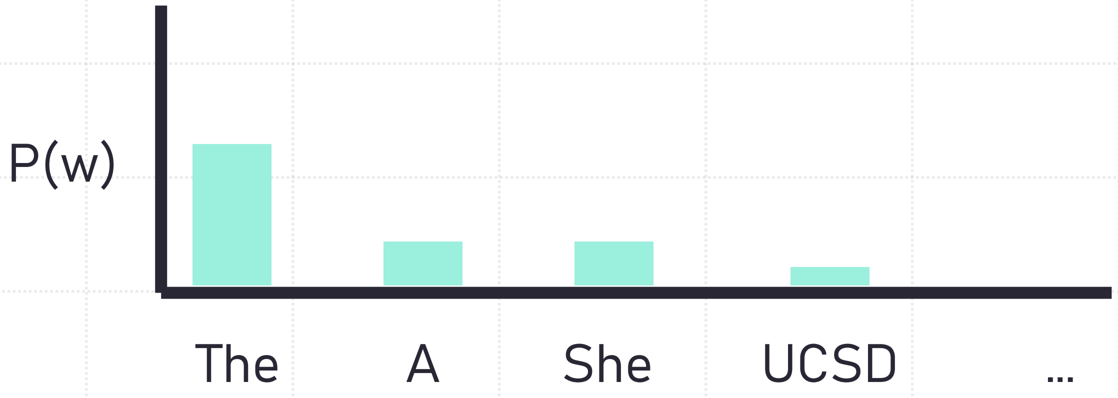


Using RNNs as <u>generative models</u>

A **generative language model** uses the probabilities assigned to upcoming tokens to actually generate novel sequences of text.

- Start with an initial token/sequence.
- Run through RNN, obtain probabilities over next word.

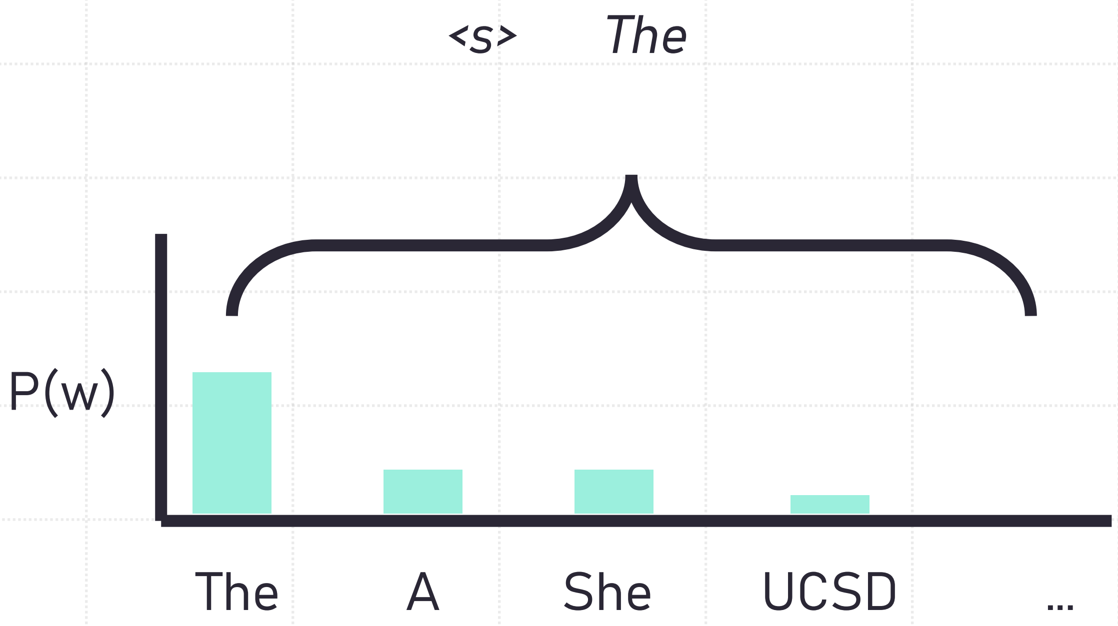
<s>



Using RNNs as generative models

A **generative language model** uses the probabilities assigned to upcoming tokens to actually generate novel sequences of text.

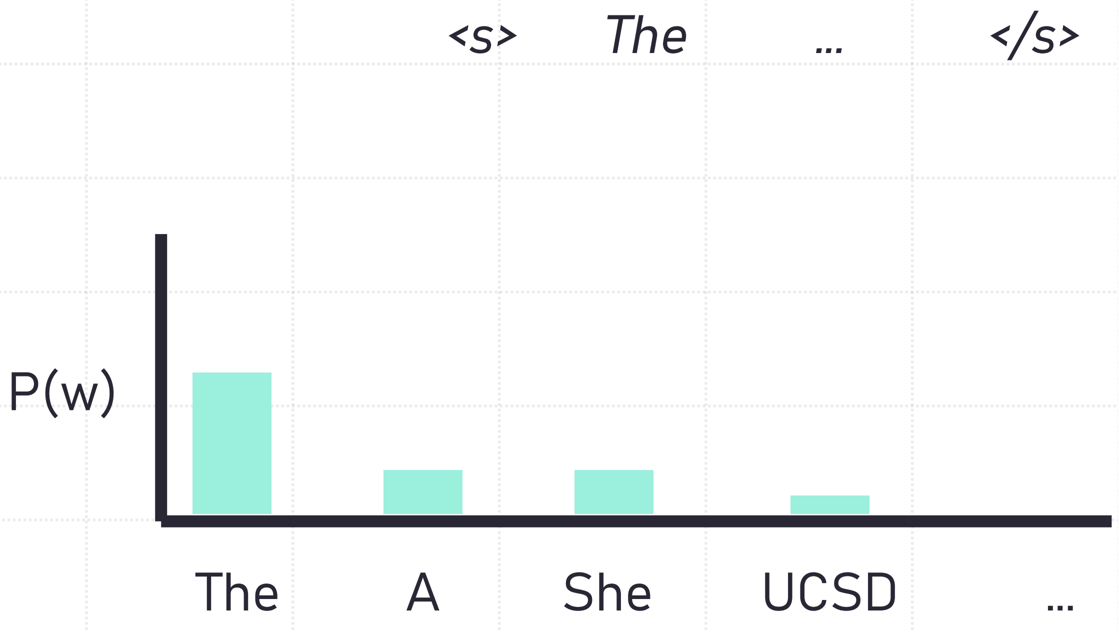
- Start with an initial token/sequence.
- Run through RNN, obtain probabilities over next word.
- **Sample** from this distribution.



Using RNNs as generative models

A **generative language model** uses the probabilities assigned to upcoming tokens to actually generate novel sequences of text.

- Start with an initial token/sequence.
- Run through RNN, obtain probabilities over next word.
- **Sample** from this distribution.
- **Repeat** until </s>.



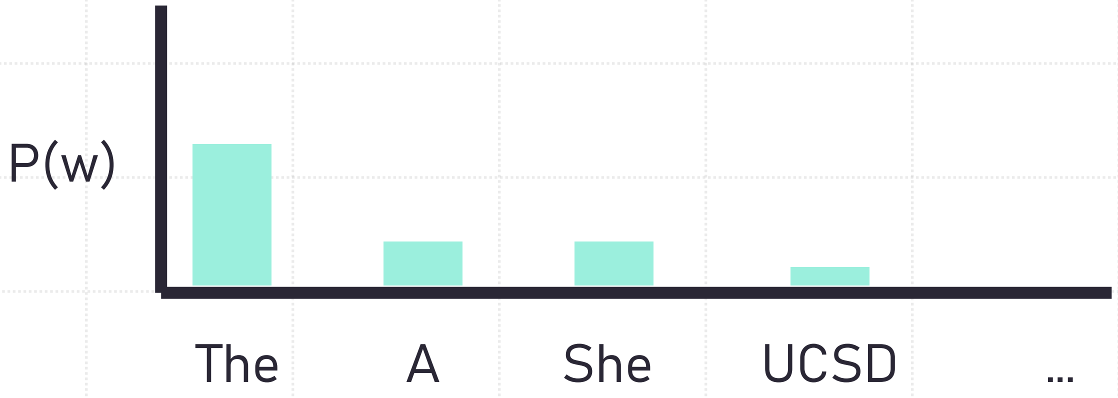
Using RNNs as <u>generative models</u>

A **<u>generative language model</u>** uses the probabilities assigned to upcoming tokens to actually <u>generate</u> novel sequences of text.

- Start with an initial token/sequence.
- Run through RNN, obtain probabilities over next word.
- **Sample** from this distribution.
- **Repeat** until </s>.

Different **sampling strategies**.

- Select *most likely* word.
- Sample proportional to *p(w)*.

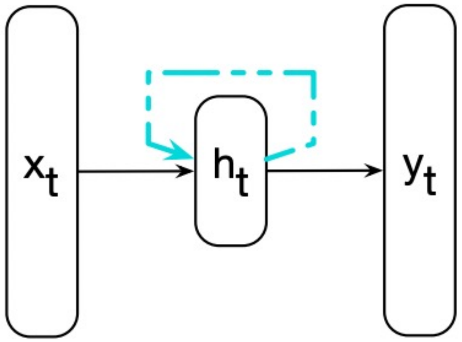


Summary

- Neural language models (e.g., using embeddings) allow for more **flexible** representations of input, facilitating **generalization**.
  - Learned **weights** map between context and predictions.
- Feed-forward LMs require a **fixed context window**.
- **Recurrent neural networks (RNNs)** model context using recurrent connection.
  - Recurrent connection acts as "**memory**" of previous hidden state h_{t-1}.
- In RNNs, "**context**" is folded into representation of previous hidden state.
- Various innovations to RNNs, including LSTMs, which better handle **long-distance dependencies**.

Lecture plan

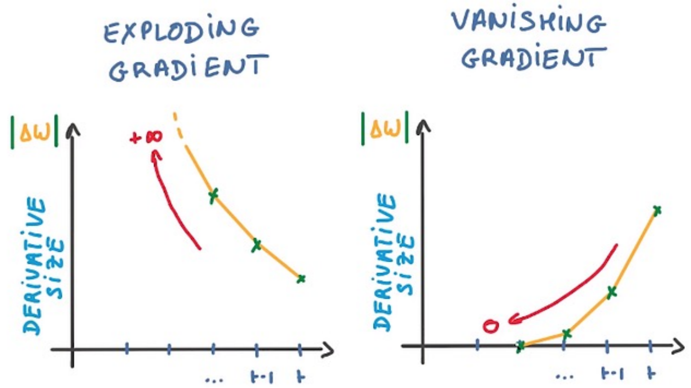
- Review: embeddings.
- Common architectures:
  - Feedforward language model.
  - Recurrent neural network.
  - Transformer architecture.
- Next time: LLMs in Python!



RNNs: recap, and limitations

- A **recurrent neural network (RNN)** has at least one *recurrent* connection, which acts as a kind of "memory" of the context.
- RNNs work pretty well, but do have limitations.

Limitation #1:
Vanishing/exploding gradient.



Limitation #2:
Training is hard to parallelize.

Recurrent structure makes it hard to process many batches in parallel—harder to take advantage of compute.

The advent of “attention”

Attention is a mechanism that—metaphorically—allows an LLM to “focus” (or “attend”) on specific elements in a sequence.

- Often, accurate predictions depend on words from a while ago.

Check the program log and find out whether it ran please.

Check the battery log and find out whether it ran down please.

The advent of “attention”

Attention is a mechanism that—metaphorically—allows an LLM to “focus” (or “attend”) on specific elements in a sequence.

- Often, <u>accurate predictions</u> depend on words from a while ago.

Check the **program** log and find out whether it **ran please**.

Check the **battery** log and find out whether it **ran down** please.

…whether it ran ____

- Knowing what comes next depends on looking <u>far back</u> in the sequence.

The advent of "attention"

Attention is a mechanism that—metaphorically—allows an LLM to "focus" (or "attend") on specific elements in a sequence.

- Often, accurate predictions depend on words from a while ago.

Check the **program** log and find out whether it **ran please**.

Check the **battery** log and find out whether it **ran down** please.

…whether it ran ____

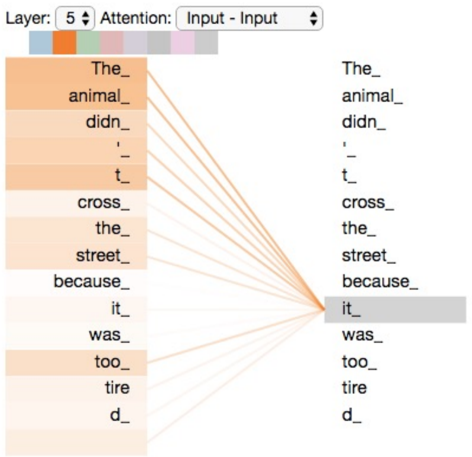
- Knowing what comes next depends on looking far back in the sequence.

The advent of “attention”

Attention is a mechanism that—metaphorically—allows an LLM to “focus” (or “attend”) on specific elements in a sequence.

- Often, <u>accurate predictions</u> depend on words from a while ago.
- This also helps identify <u>relationships</u> between elements in the sequence.

The animal didn’t cross the street because it was tired.

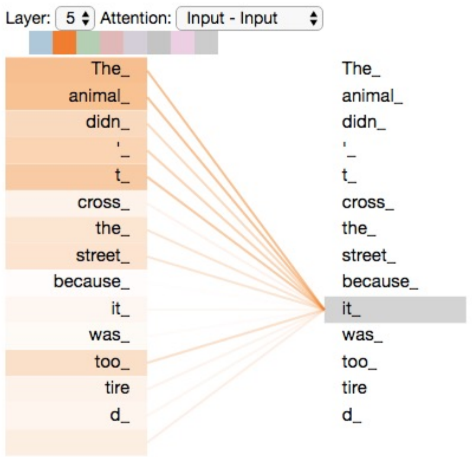


The advent of “attention”

Attention is a mechanism that—metaphorically—allows an LLM to “focus” (or “attend”) on specific elements in a sequence.

- Often, accurate predictions depend on words from a while ago.
- This also helps identify relationships between elements in the sequence.

The **animal** didn’t cross the street because **it** was tired.

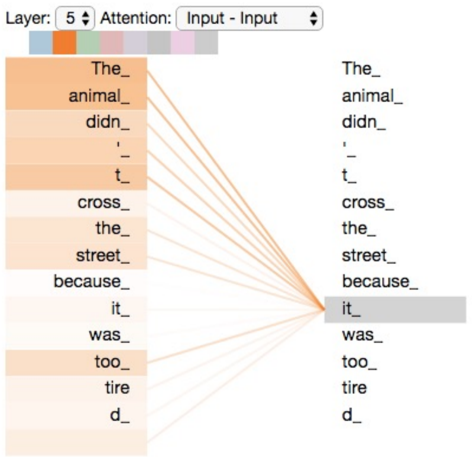


The advent of "attention"

Attention is a mechanism that—metaphorically—allows an LLM to "focus" (or "attend") on specific elements in a sequence.

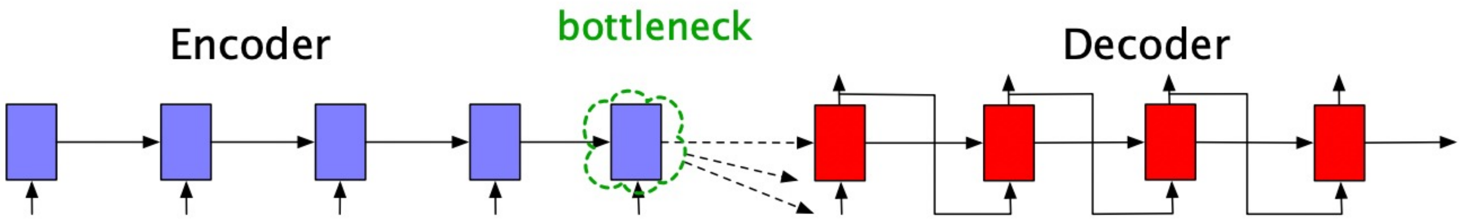
- Often, accurate predictions depend on words from a while ago.
- This also helps identify relationships between elements in the sequence.

The **animal** didn't cross the street because **it** was tired.



Attention: the origins

- Originally, attention was developed to help with **machine translation**.
- Traditional, RNN-based translation models had a "bottleneck" in their design.

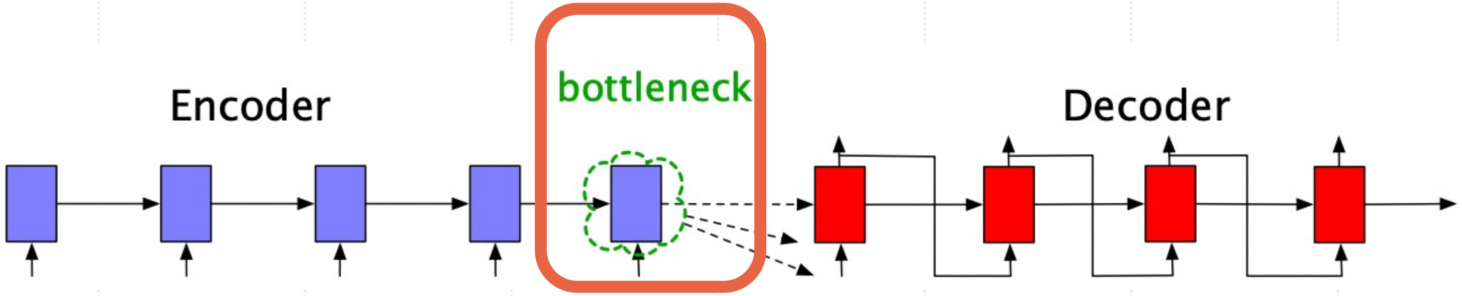


Encodes source language.

Decodes to target language.

Attention: the origins

- Originally, attention was developed to help with **machine translation**.
- Traditional, RNN-based translation models had a “bottleneck” in their design.



Encodes source language.

Decodes to target language.

Bottleneck: *all* the information required to translate a sentence must be packed into this last hidden state.

Attention: the origins

- Originally, attention was developed to help with **machine translation**.
- Traditional, RNN-based translation models had a "bottleneck" in their design.
- Attention is a mechanism for putting *all* those hidden states into a single fixed-length vector—by focusing on what's most relevant.

Dot-product attention: implements "relevance" as *embedding similarity.*

To illustrate this, let's look at an example from a domain we're already familiar with—language modeling.

Dot-product attention: illustrated

In **dot-product attention**, the dot product between every pair of words is used to build a custom, context-dependent vector.

The cat sat on ___

“on”

The

cat

sat

on

Dot-product attention: illustrated

In **dot-product attention**, the dot product between every pair of words is used to build a custom, context-dependent vector.

The cat sat on ___ "on"

V1	The
V2	cat
V3	sat
V4	on

Dot-product attention: illustrated

In **dot-product attention**, the dot product between every pair of words is used to build a custom, context-dependent vector.

The cat sat on ___

“on”

V1	The
V2	cat
V3	sat
V4	on

Each of these is represented by an **embedding**.

The dot product captures the **similarity** in embeddings.

Dot-product attention: illustrated

In **dot-product attention**, the dot product between every pair of words is used to build a custom, context-dependent vector.

The cat sat on ___

		"on" $w * c$
V1	The	.2
V2	cat	.1
V3	sat	.1
V4	on	1

Numbers made up for illustration purposes!

Dot-product attention: illustrated

In **dot-product attention**, the dot product between every pair of words is used to build a custom, context-dependent vector.

The cat sat on ___

		"on"	$w * c$	$\sigma(x)_j = \frac{e^{x_j}}{\sum_k e^{x_k}}$
V1	The		.2	.2
V2	cat		.1	.18
V3	sat		.1	.18
V4	on		1	.44

Now, we **soft-max** these values to create a probability distribution.

Dot-product attention: illustrated

In **dot-product attention**, the dot product between every pair of words is used to build a custom, context-dependent vector.

The cat sat on ___

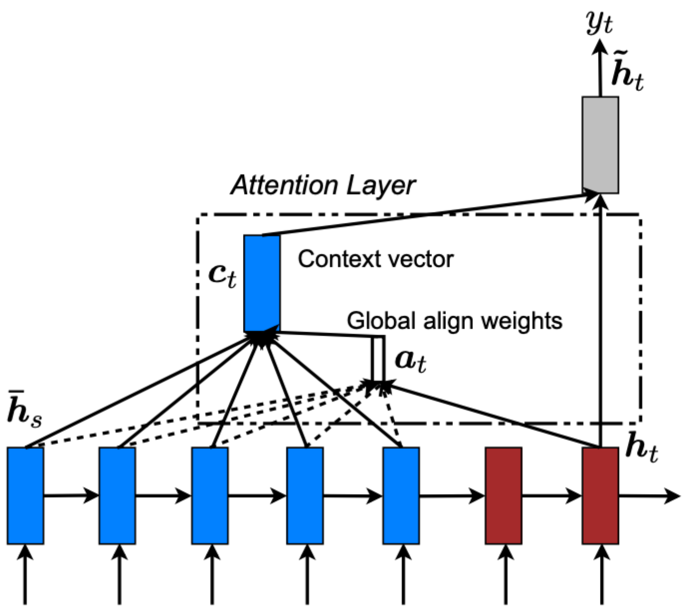
		"on"	$w * c$	$\sigma(x)_j = \frac{e^{x_j}}{\sum_k e^{x_k}}$
V1	The		.2	.2
V2	cat		.1	.18
V3	sat		.1	.18
V4	on		1	.44

These are our **attention weights**.

Each represents the "relevance" of V_n to "on".

Dot-product attention: illustrated

In **dot-product attention**, the dot product between every pair of words is used to build a custom, context-dependent vector.



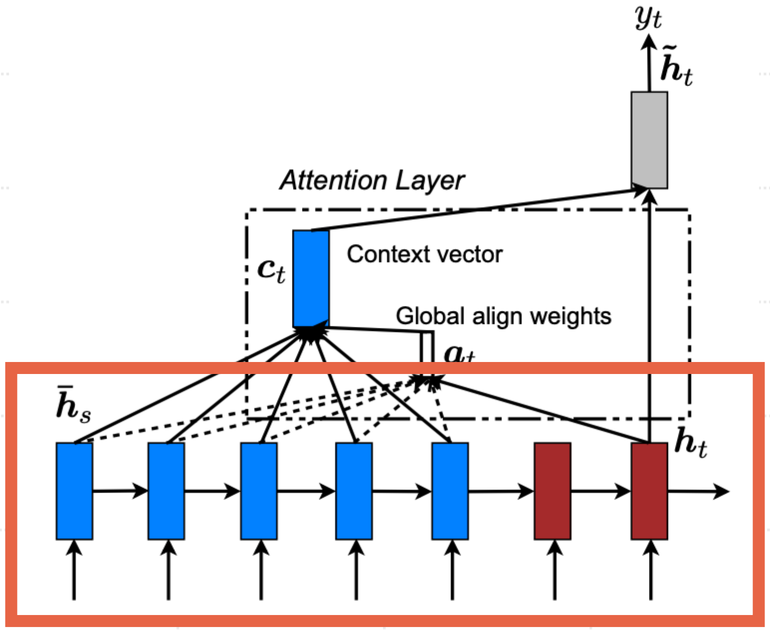
Now, compute **weighted average** over all hidden states—using these attention scores as "weights"!

$$\mathbf{c}_i = \sum_j \alpha_{ij} \, \mathbf{h}^e_j$$

Dot-product attention: illustrated

In **dot-product attention**, the dot product between every pair of words is used to build a custom, context-dependent vector.

Hidden states



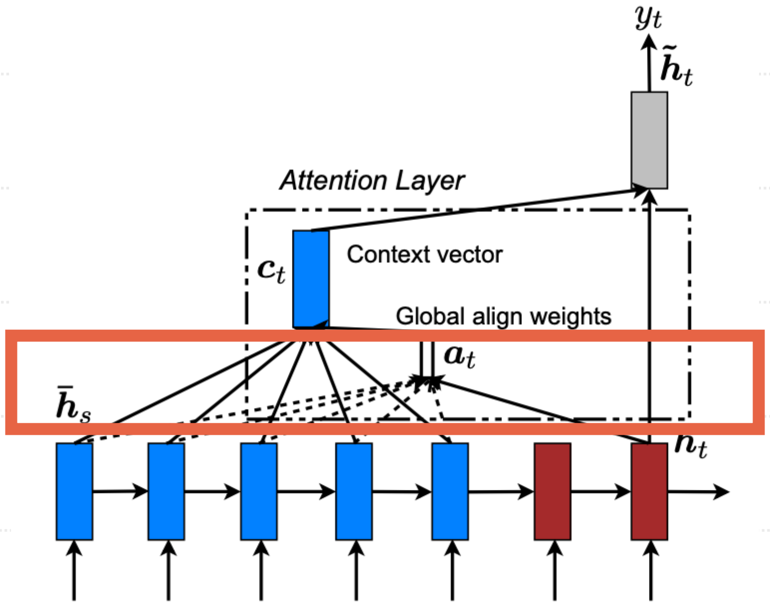
Now, compute **weighted average** over all hidden states—using these <u>attention scores</u> as "weights"!

$$\mathbf{c}_i = \sum_j \alpha_{ij} \mathbf{h}_j^e$$

Dot-product attention: illustrated

In **dot-product attention**, the dot product between every pair of words is used to build a custom, context-dependent vector.

Compute attention weights



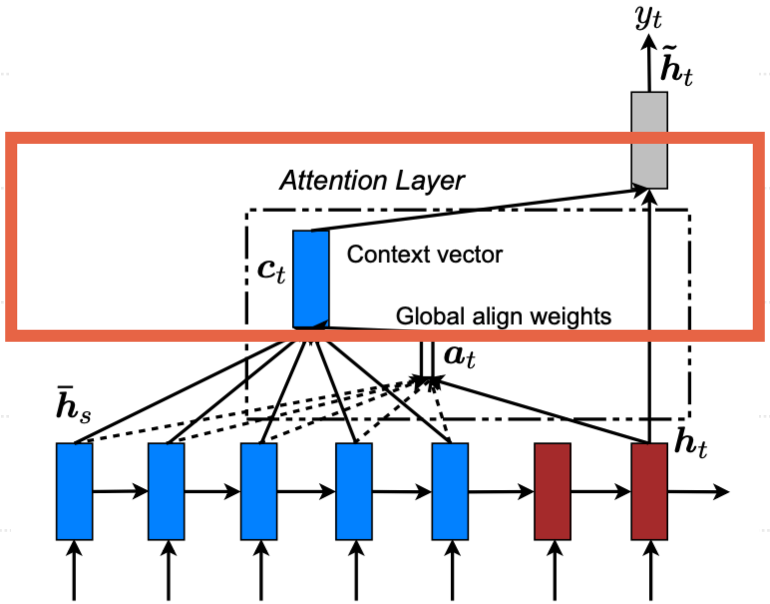
Now, compute **weighted average** over all hidden states—using these attention scores as "weights"!

$$\mathbf{c}_i = \sum_j \alpha_{ij}\, \mathbf{h}^e_j$$

Dot-product attention: illustrated

In **dot-product attention**, the dot product between every pair of words is used to build a custom, context-dependent vector.

Use attention weights to create new **context vector**.



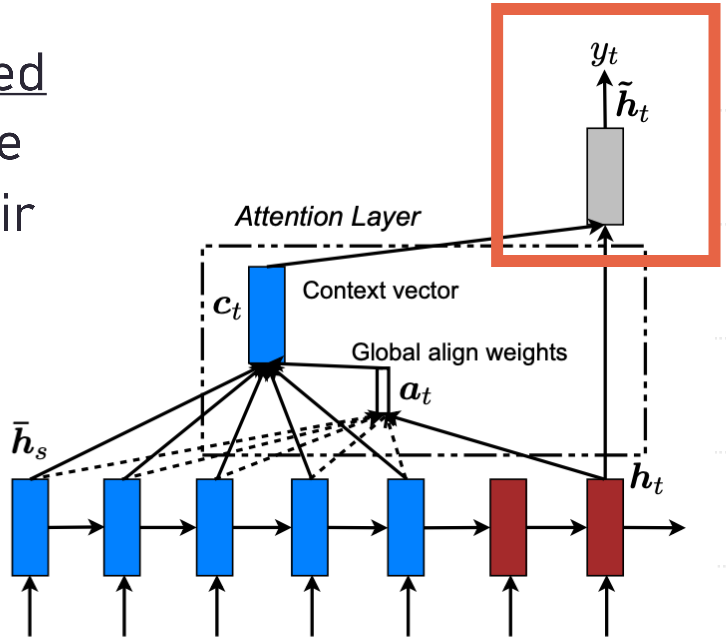
Now, compute **weighted average** over all hidden states—using these <u>attention scores</u> as "weights"!

$$\mathbf{c}_i = \sum_j \alpha_{ij} \mathbf{h}_j^e$$

Dot-product attention: illustrated

In **dot-product attention**, the dot product between every pair of words is used to build a custom, context-dependent vector.

Predictions are now <u>weighted</u> by different elements of the sequence depending on their “relevance”.



Now, compute **weighted average** over all hidden states—using these <u>attention scores</u> as “weights”!

$$\mathbf{c}_i = \sum_j \alpha_{ij} \mathbf{h}^e_j$$

Dot-product attention: illustrated

In **dot-product attention**, the dot product between every pair of words is used to build a custom, context-dependent vector.

In theory, we can do this at each layer of a neural network.

But the dot product is still a pretty coarse measure of attention.

Can we do better?

“RNN + Attention—but throw out the RNN!”

Introducing transformers

The **Transformer** is a neural network architecture that uses multi-head self-attention, with no recurrent units.

- Use a **fixed context window**.
- No **recurrent connections**.
- Use **self-attention**.
- Have multiple attention “heads” (**multi-head self-attention**).
- Use **positional embeddings**.

"RNN + Attention—but throw out the RNN!"

Introducing transformers

The **Transformer** is a neural network architecture that uses multi-head self-attention, with no recurrent units.

- Use a **fixed context window**.
- No **recurrent connections**.
- Use **self-attention**.
- Have multiple attention "heads" (**multi-head self-attention**).
- Use **positional embeddings**.

What do these aspects of a transformer remind you of?

A traditional **feed-forward neural language model**!

(Note: this is why you often hear about the "context window size" of models like ChatGPT, Claude, etc.)

“RNN + Attention—but throw out the RNN!”

Introducing transformers

The **Transformer** is a neural network architecture that uses multi-head self-attention, with no recurrent units.

- Use a **fixed context window**.
- No **recurrent connections**.
- Use **self-attention**.
- Have multiple attention “heads” (**multi-head self-attention**).
- Use **positional embeddings**.

These are new concepts—let’s focus on **self-attention** first.

Self-attention: match-making for words

In **self-attention**, the relevance of each word to each other is calculated in context and shared, informing the model's predictions.

Query (Q): representation of current word, used to score against all other words in sequence.

Key (K): labels for other words in sequence, which we "match" against in our search.

Value (V): represent the "content" of each word, which are weighed by attention scores.

A robot must obey the orders given it…

https://jalammar.github.io/illustrated-gpt2/#part-1-got-and-language-modeling

Self-attention: match-making for words

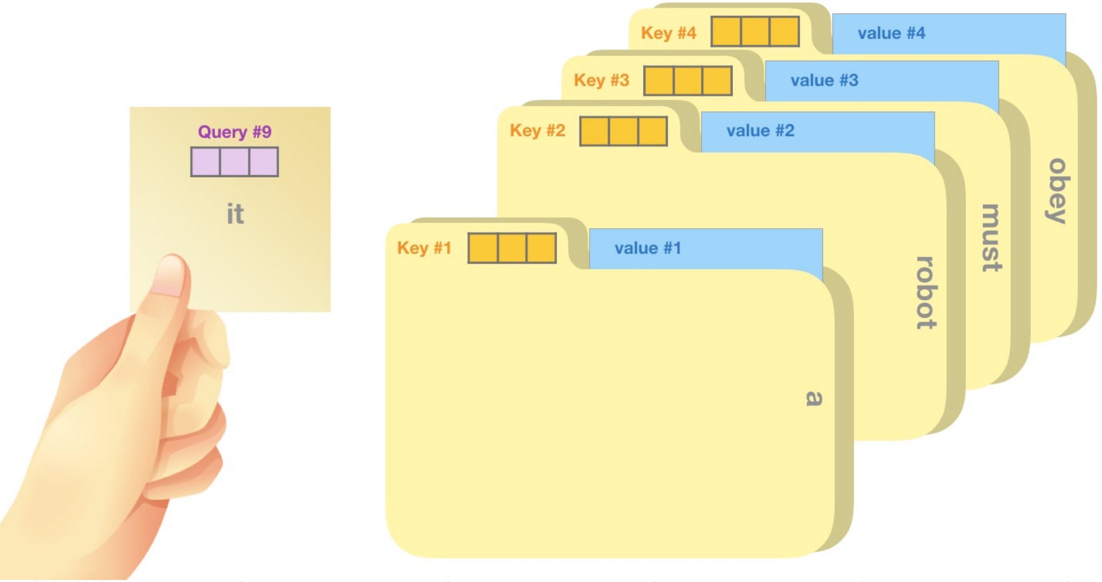
In **self-attention**, the relevance of each word to each other is calculated in context and shared, informing the model's predictions.

Query (Q): representation of current word, used to score against all other words in sequence.

Key (K): labels for other words in sequence, which we "match" against in our search.

Value (V): represent the "content" of each word, which are weighed by attention scores.

A robot must obey the orders given **it**…



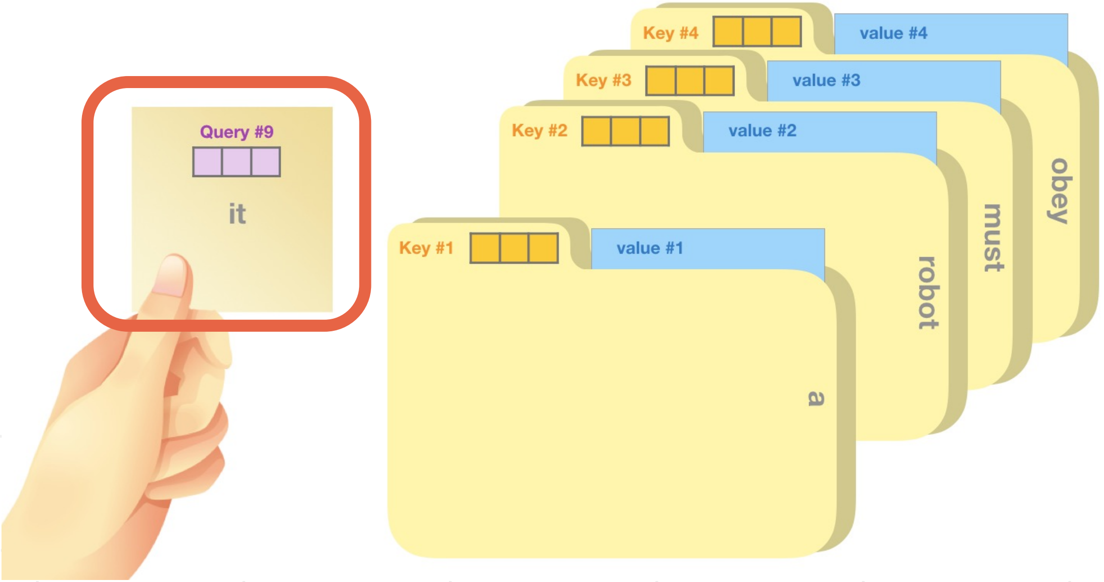
https://jalammar.github.io/illustrated-gpt2/#part-1-got-and-language-modeling

Self-attention: match-making for words

In **self-attention**, the relevance of each word to each other is calculated in context and shared, informing the model's predictions.

Here, we're looking for words that are relevant to "it".

A robot must obey the orders given **it**…



https://jalammar.github.io/illustrated-gpt2/#part-1-got-and-language-modeling

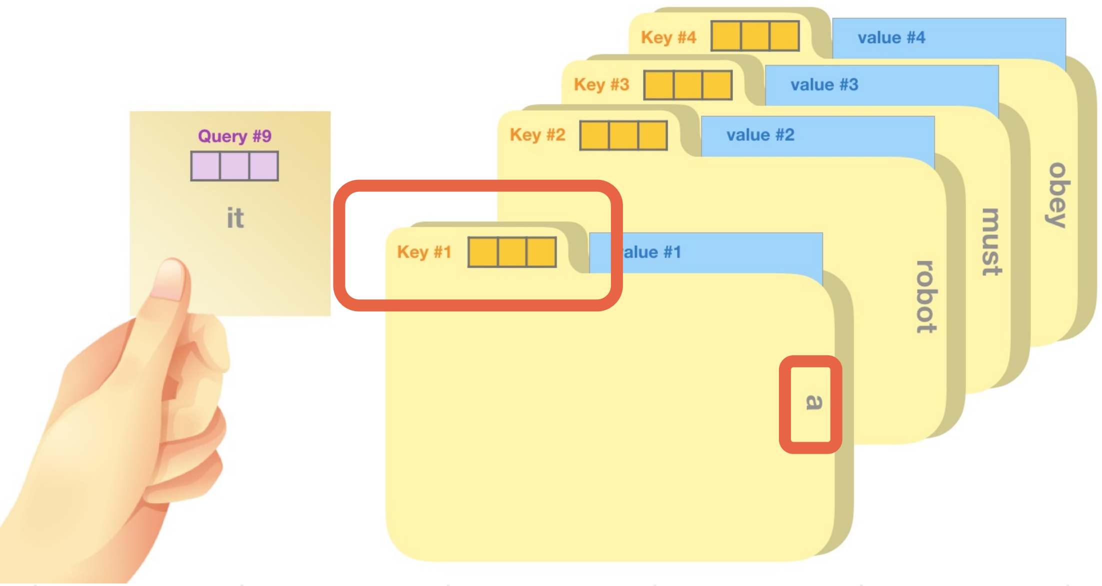
Self-attention: match-making for words

In **self-attention**, the <u>relevance</u> of each word to each other is calculated <u>in context</u> and <u>shared</u>, informing the model's predictions.

Here, we're looking for words that are relevant to "it".

Key for each word is like a <u>label</u> for "folders" in a filing cabinet.

A robot must obey the orders given **it**…



https://jalammar.github.io/illustrated-gpt2/#part-1-got-and-language-modeling

Self-attention: match-making for words

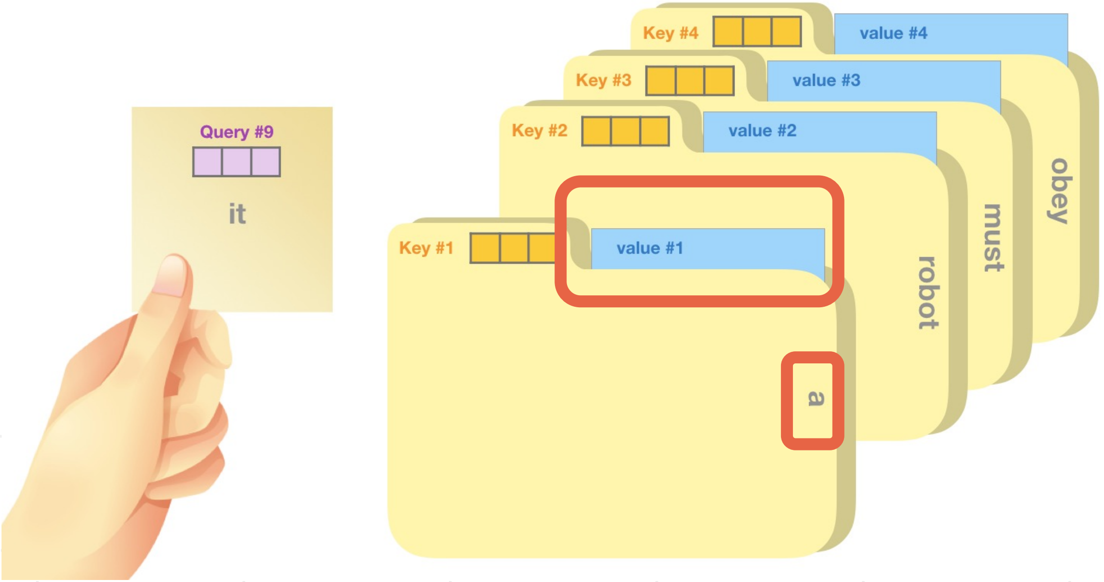
In **self-attention**, the relevance of each word to each other is calculated in context and shared, informing the model's predictions.

Here, we're looking for words that are relevant to "it".

Key for each word is like a label for "folders" in a filing cabinet.

Values are the contents of those filing cabinets.

A robot must obey the orders given **it**…



https://jalammar.github.io/illustrated-gpt2/#part-1-got-and-language-modeling

Self-attention: match-making for words

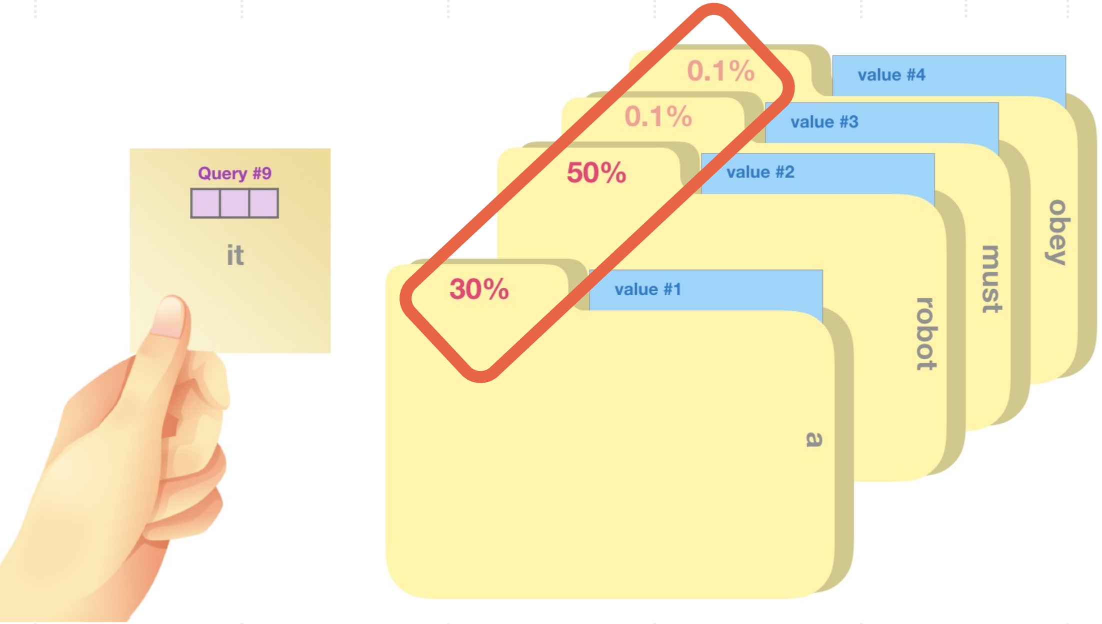
In **self-attention**, the relevance of each word to each other is calculated in context and shared, informing the model's predictions.

To compute **attention score**, multiply query by key vectors for each pair.

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{q}_i \cdot \mathbf{k}_j$$

(We then **normalize** and **soft-max** these scores to get a probability distribution.)

A robot must obey the orders given **it**...



https://jalammar.github.io/illustrated-gpt2/#part-1-got-and-language-modeling

Self-attention: match-making for words

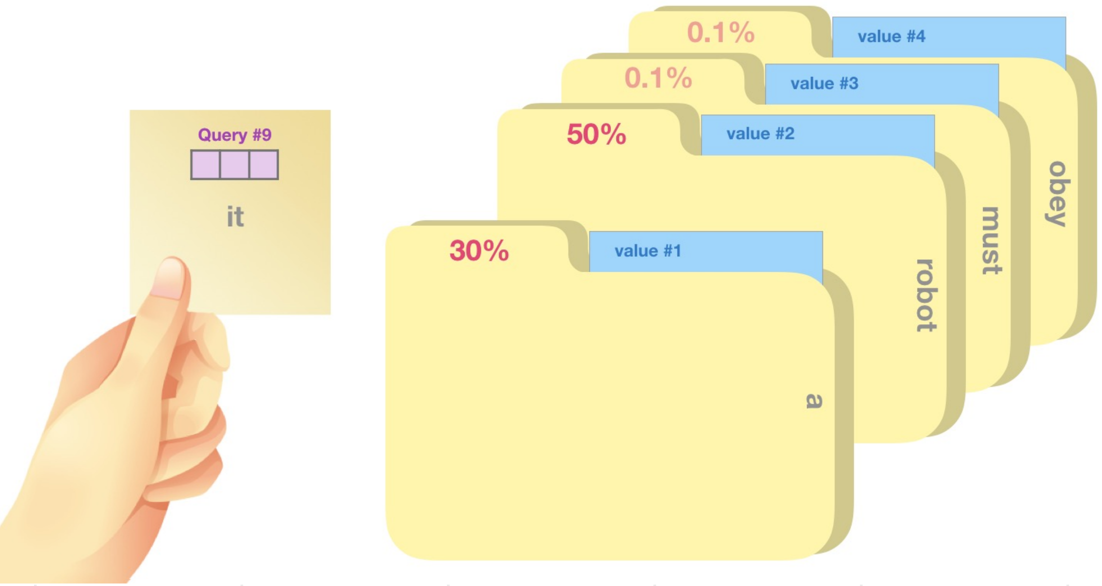
In **self-attention**, the relevance of each word to each other is calculated in context and shared, informing the model's predictions.

To compute **attention score**, multiply query by key vectors for each pair.

Now, multiply (and sum) attention scores by value vectors.

$$\mathbf{y}_i = \sum_{j \leq i} \alpha_{ij} \mathbf{v}_j$$

A robot must obey the orders given **it**...



https://jalammar.github.io/illustrated-gpt2/#part-1-got-and-language-modeling

Self-attention: match-making for words

In **self-attention**, the relevance of each word to each other is calculated in context and shared, informing the model's predictions.

To compute **attention score**, multiply query by key vectors for each pair.

Now, multiply (and sum) attention scores by value vectors.

$$\mathbf{y}_i = \sum_{j \leq i} \alpha_{ij} \mathbf{v}_j$$

A robot must obey the orders given **it**...

Word	Value vector	Score	Value X Score
<s>		0.001	
a		0.3	
robot		0.5	
must		0.002	
obey		0.001	
the		0.0003	
orders		0.005	
given		0.002	
		0.19	
		Sum:	

This is our new **contextualized embedding** for "it".

https://jalammar.github.io/illustrated-gpt2/#part-1-got-and-language-modeling

Self-attention: match-making for words

In **self-attention**, the <u>relevance</u> of each word to each other is calculated <u>in context</u> and <u>shared</u>, informing the model's predictions.

We compute **attention scores** between each word w_t and every word that comes before it.

In an **auto-regressive model**, we prevent attention from "looking ahead" at future words.

N

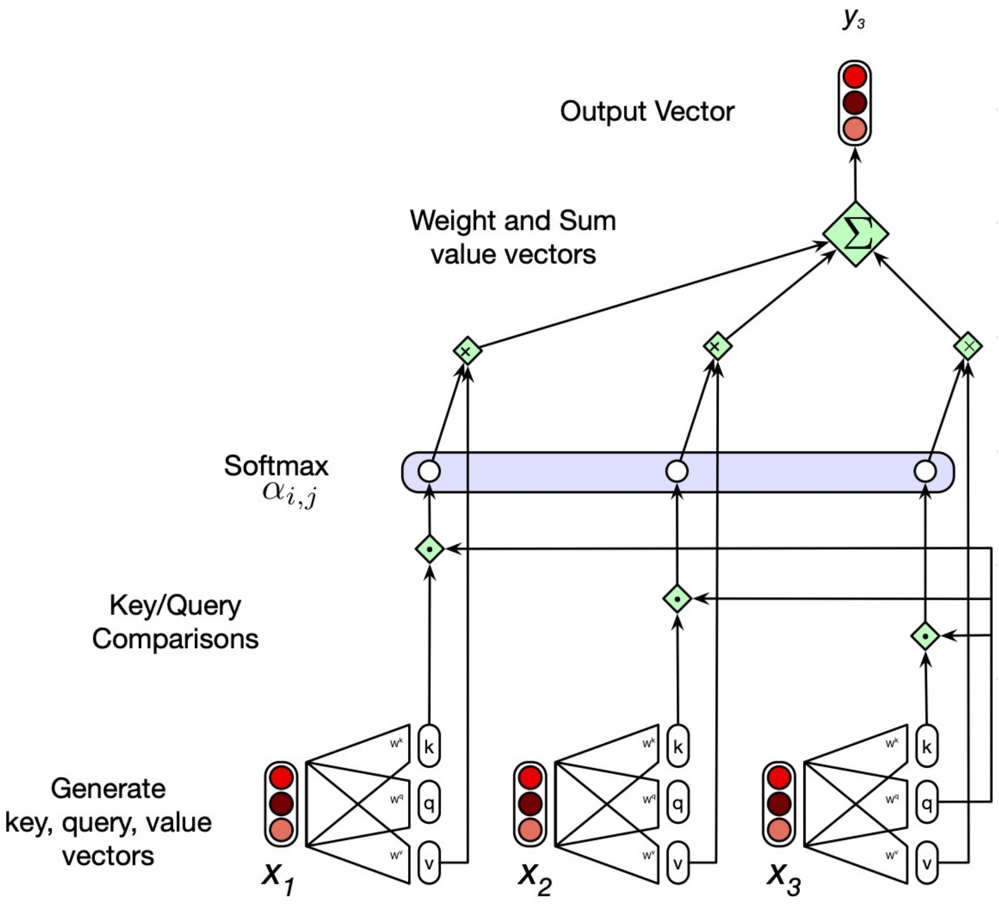
q1•k1	$-\infty$	$-\infty$	$-\infty$	$-\infty$
q2•k1	q2•k2	$-\infty$	$-\infty$	$-\infty$
q3•k1	q3•k2	q3•k3	$-\infty$	$-\infty$
q4•k1	q4•k2	q4•k3	q4•k4	$-\infty$
q5•k1	q5•k2	q5•k3	q5•k4	q5•k5

N

In terms of compute time, how "efficient" is this process?

It's **quadratic**—we must compute dot product between every pair of tokens in the input.

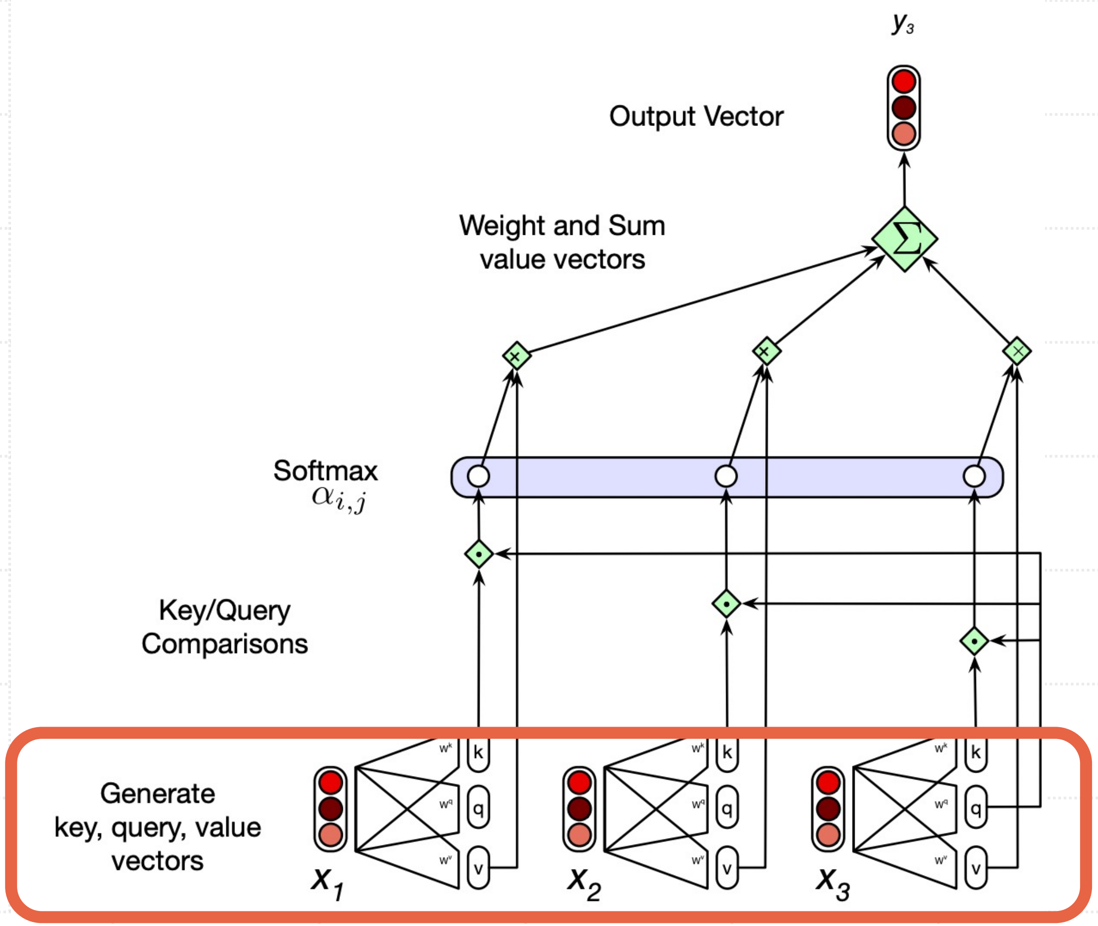
Self-attention: a closer look



Suppose we are computing **self-attention** for X_3.

https://jalammar.github.io/illustrated-gpt2/#part-1-got-and-language-modeling

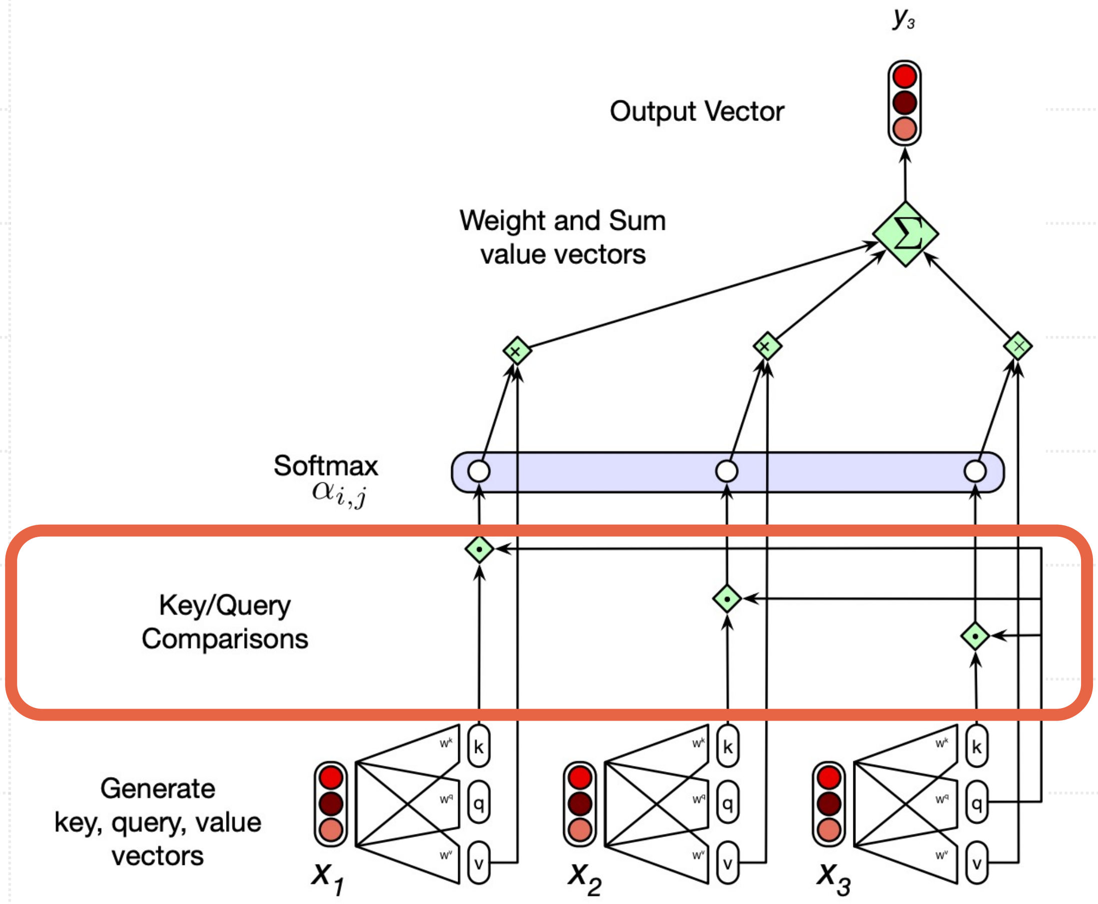
Self-attention: a closer look



Suppose we are computing **self-attention** for X_3.

For each word in sequence, compute **key**, **query**, and **value** vectors.

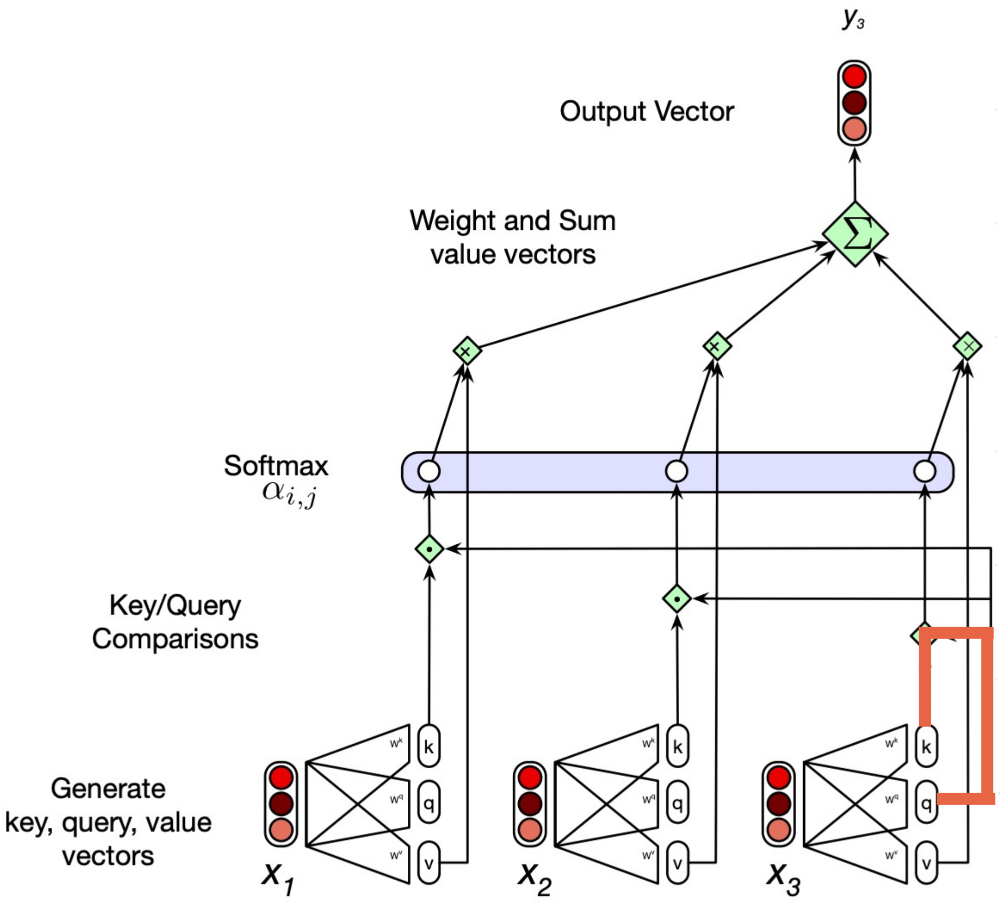
Self-attention: a closer look



Suppose we are computing **self-attention** for X_3.

Compute relevance of X_1 and X_2 to X_3.

Self-attention: a closer look

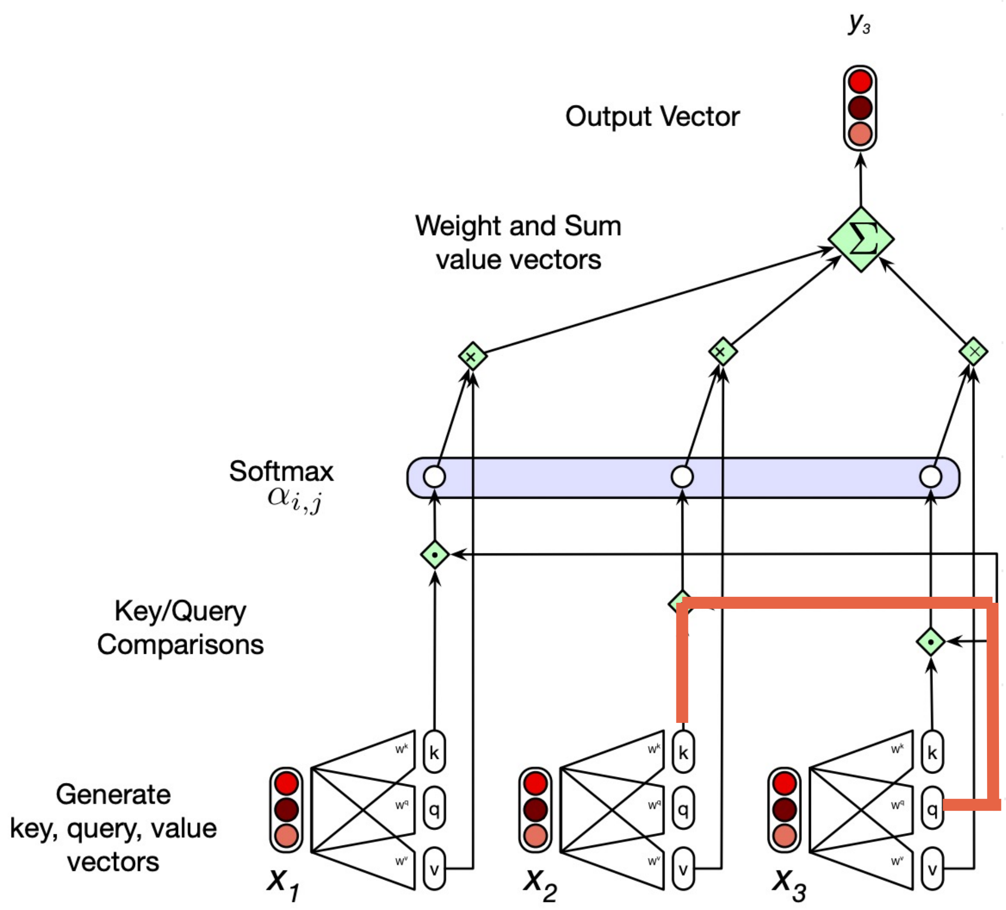


Suppose we are computing **self-attention** for X_3.

Compute relevance of X_1 and X_2 to X_3.

https://jalammar.github.io/illustrated-gpt2/#part-1-got-and-language-modeling

Self-attention: a closer look

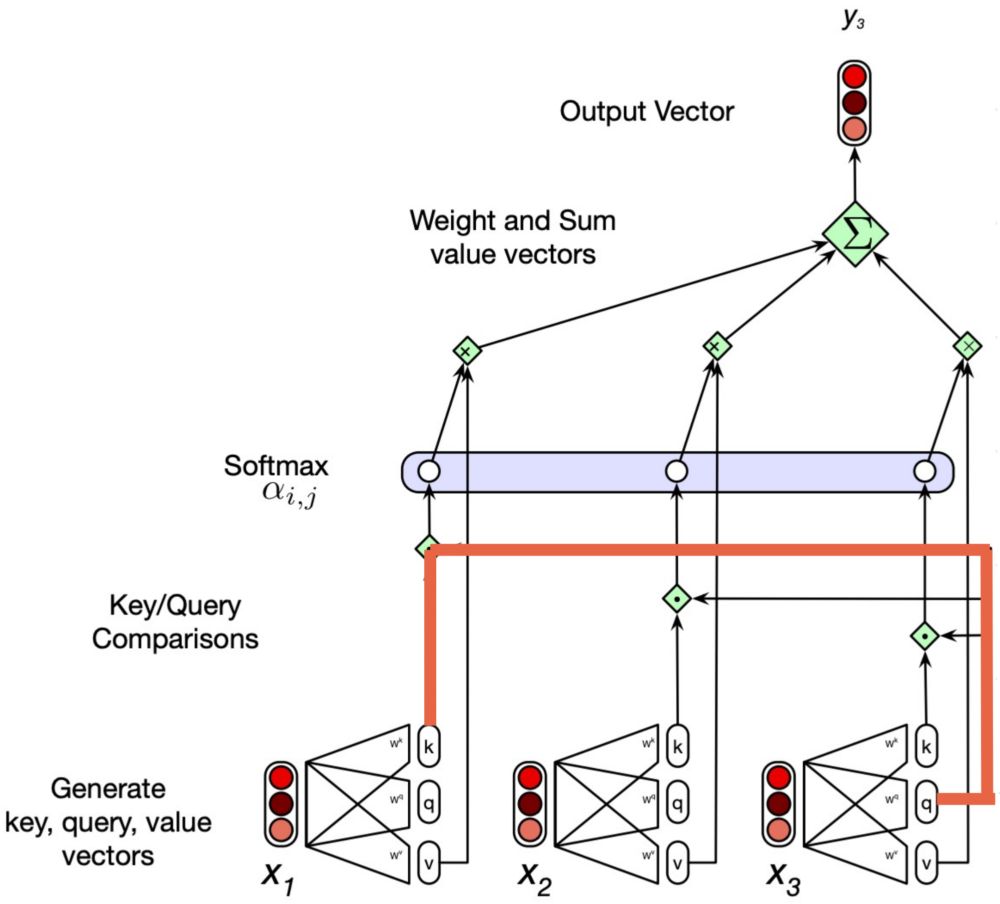


Suppose we are computing **self-attention** for X_3.

Compute relevance of X_1 and X_2 to X_3.

https://jalammar.github.io/illustrated-gpt2/#part-1-got-and-language-modeling

Self-attention: a closer look

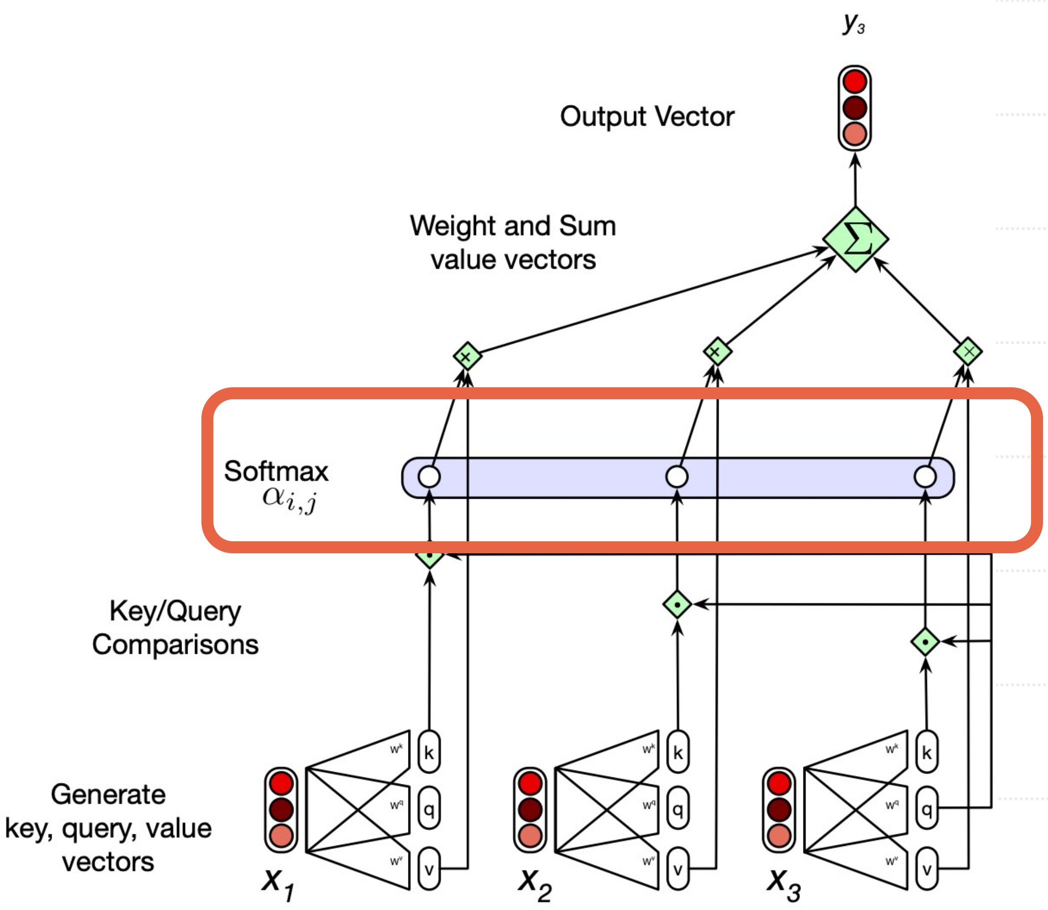


Suppose we are computing **self-attention** for X_3.

Compute relevance of X_1 and X_2 to X_3.

https://jalammar.github.io/illustrated-gpt2/#part-1-got-and-language-modeling

Self-attention: a closer look

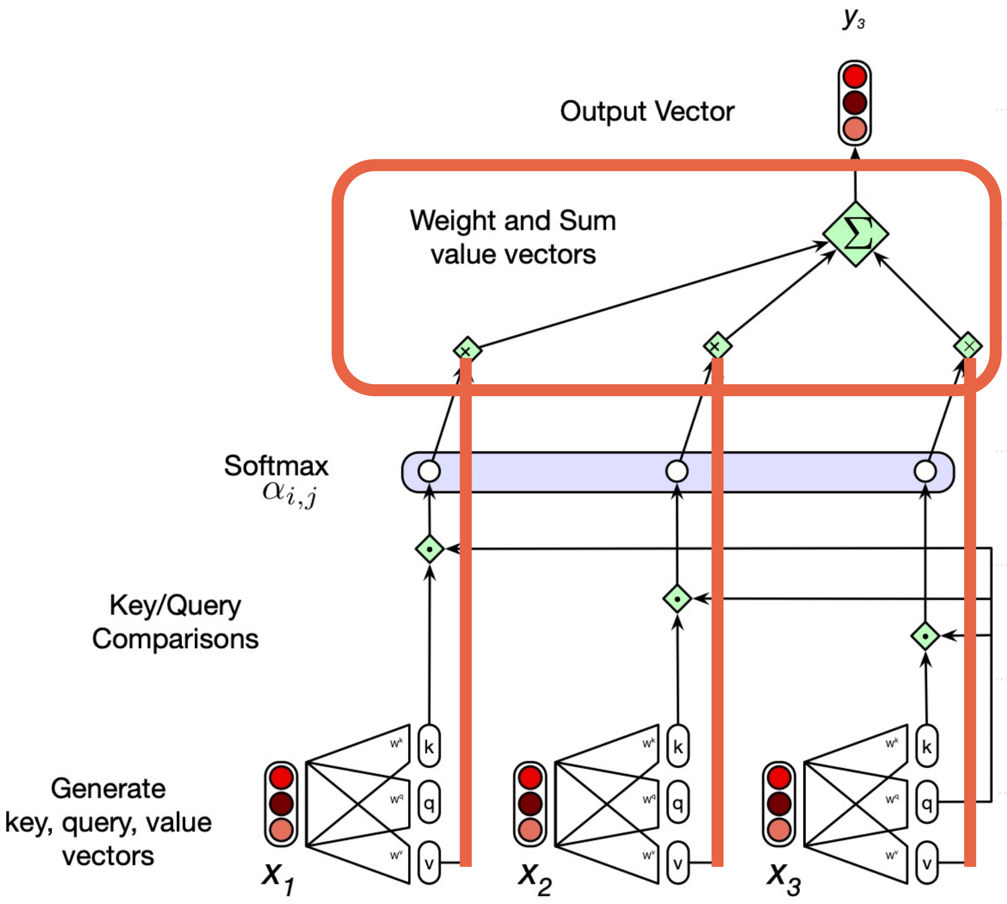


Suppose we are computing **self-attention** for X_3.

Soft-max these to get **attention scores**.

https://jalammar.github.io/illustrated-gpt2/#part-1-got-and-language-modeling

Self-attention: a closer look

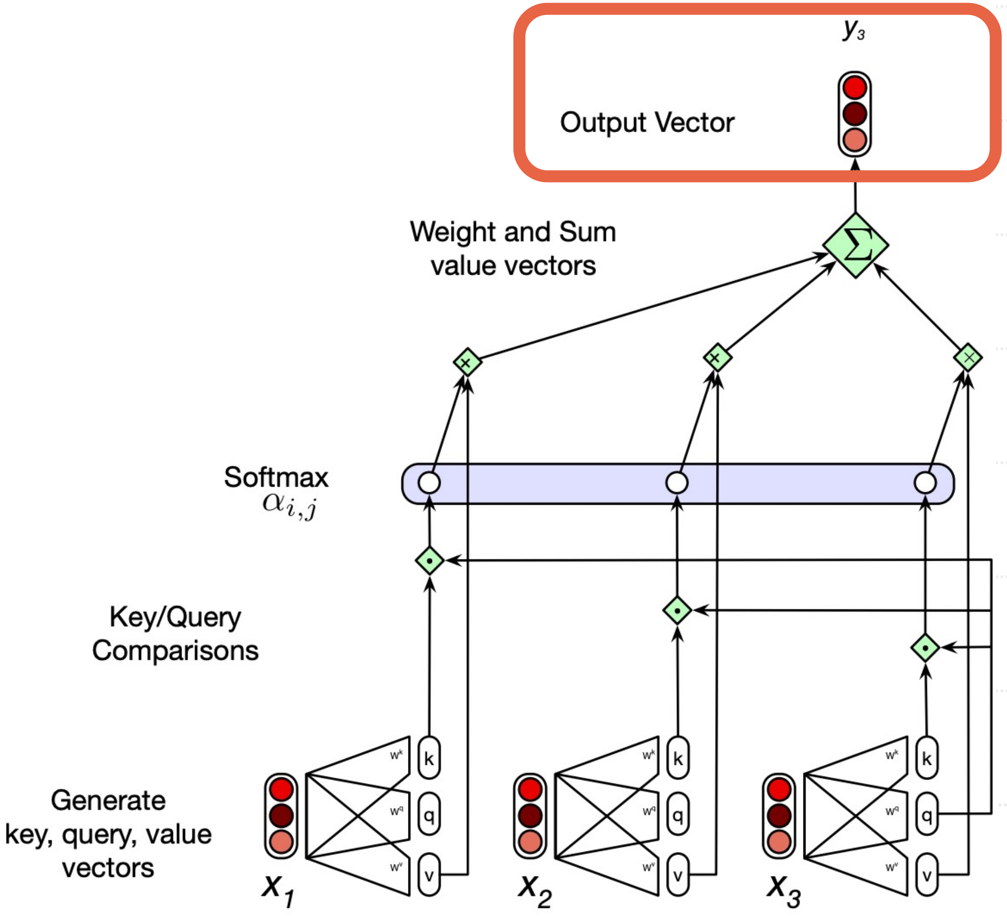


Suppose we are computing **self-attention** for X_3.

Use attention scores to weigh the **value vectors**.

https://jalammar.github.io/illustrated-gpt2/#part-1-got-and-language-modeling

Self-attention: a closer look

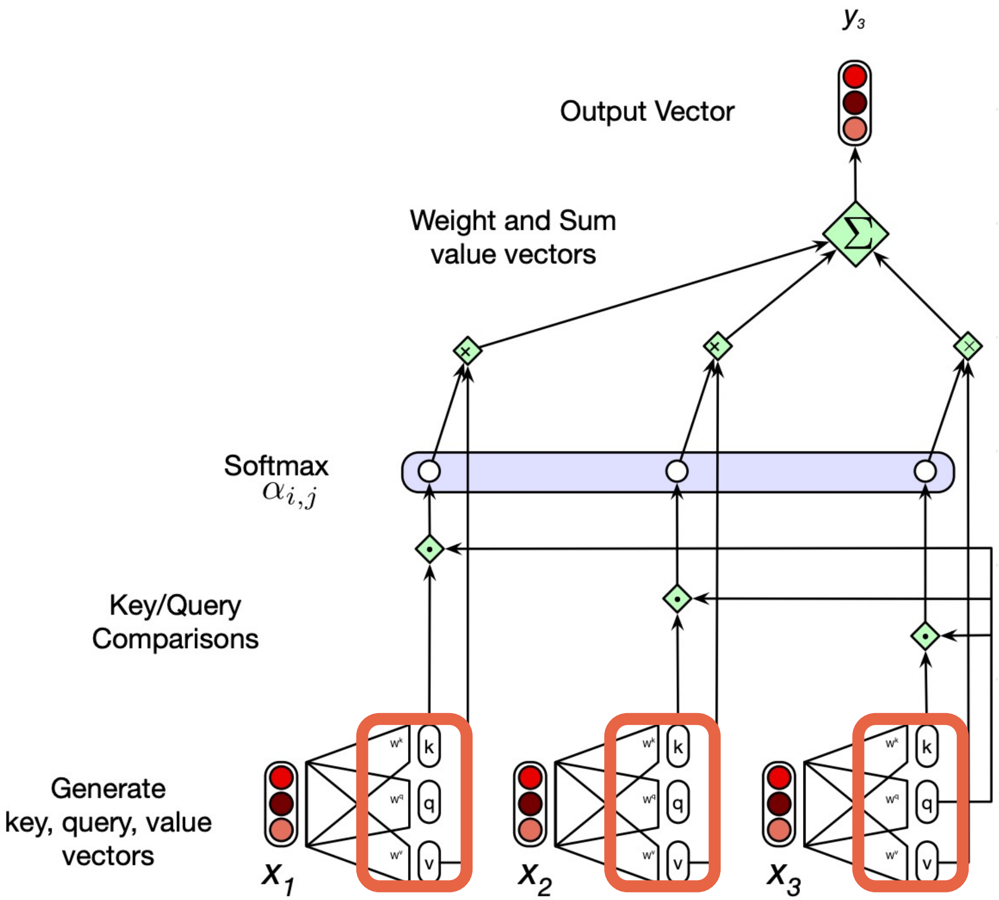


Suppose we are computing **self-attention** for X_3.

The result is a <u>new embedding Y_3</u>, which "folds in" the relevant information from X_1 and X_2 into X_3.

https://jalammar.github.io/illustrated-gpt2/#part-1-got-and-language-modeling

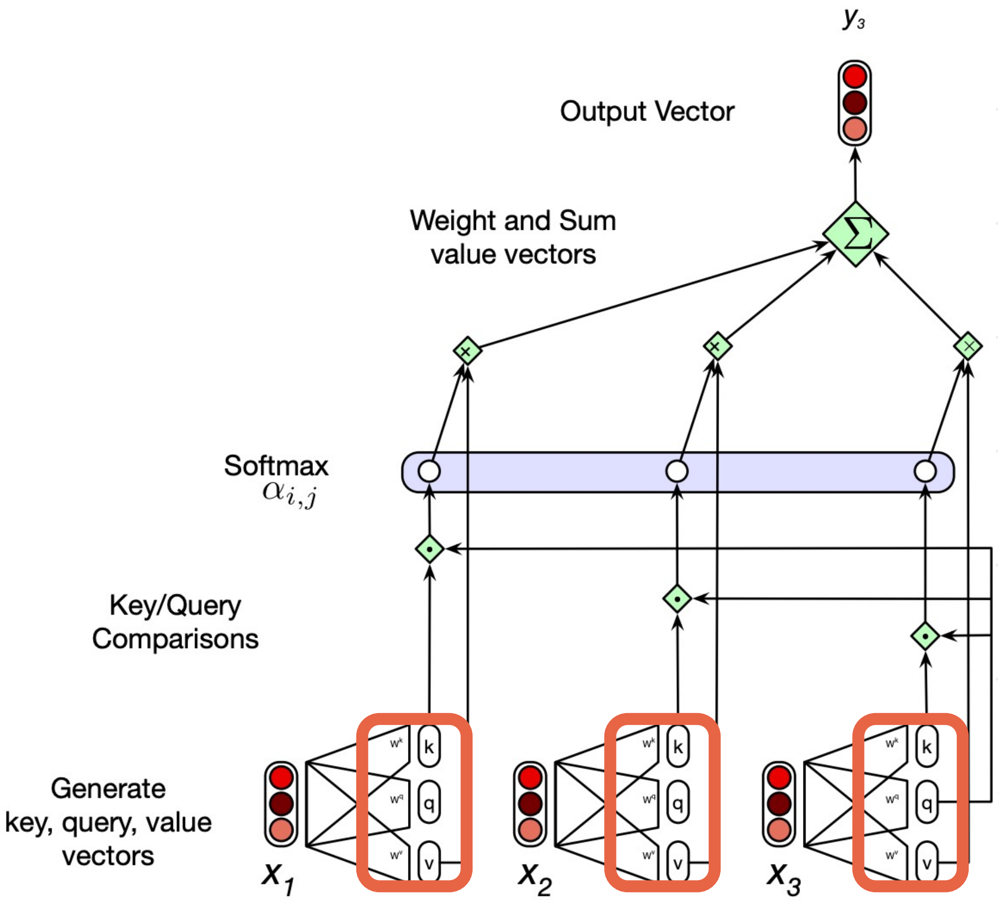
Self-attention: a closer look



Where do Q, K, V come from?

https://jalammar.github.io/illustrated-gpt2/#part-1-got-and-language-modeling

Self-attention: a closer look



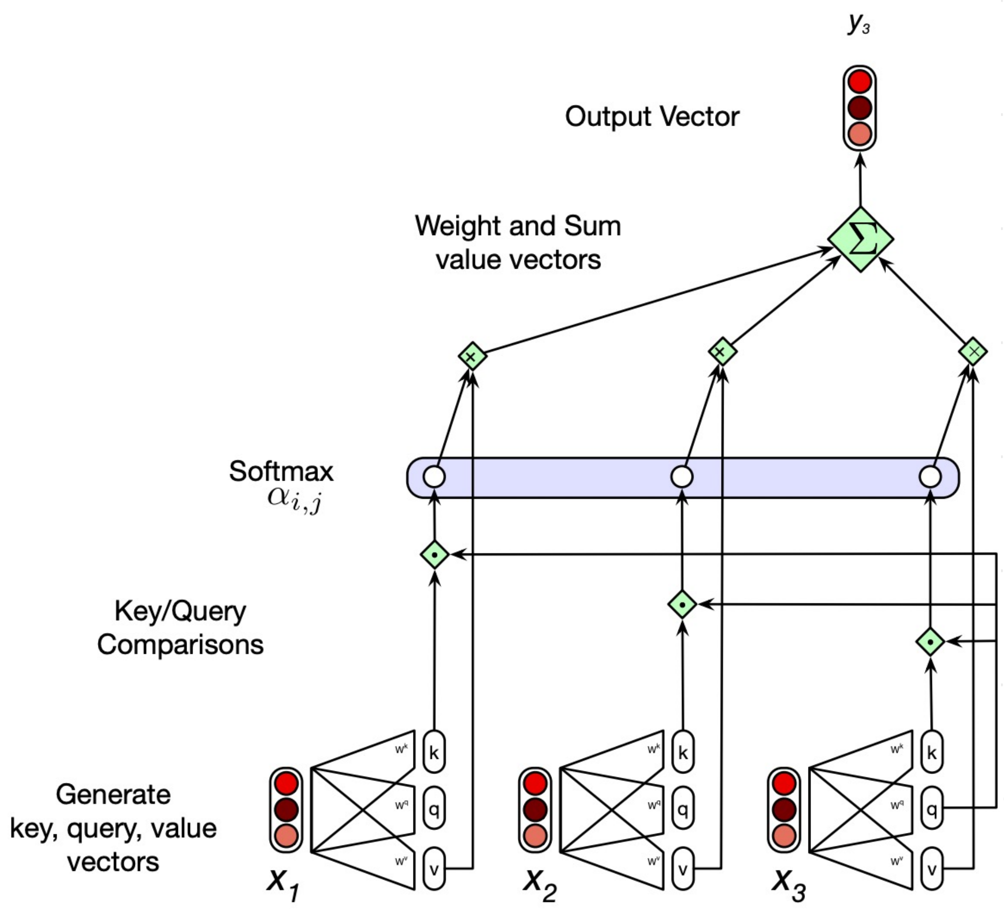
During training, we also **learn weight matrices W^Q, W^K, and W^V,** which we multiply by input **X**.

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{\mathbf{Q}}; \ \ \mathbf{K} = \mathbf{X}\mathbf{W}^{\mathbf{K}}; \ \ \mathbf{V} = \mathbf{X}\mathbf{W}^{\mathbf{V}}$$

Learned just like standard weights—by iteratively updating through **back-propagation**.

https://jalammar.github.io/illustrated-gpt2/#part-1-got-and-language-modeling

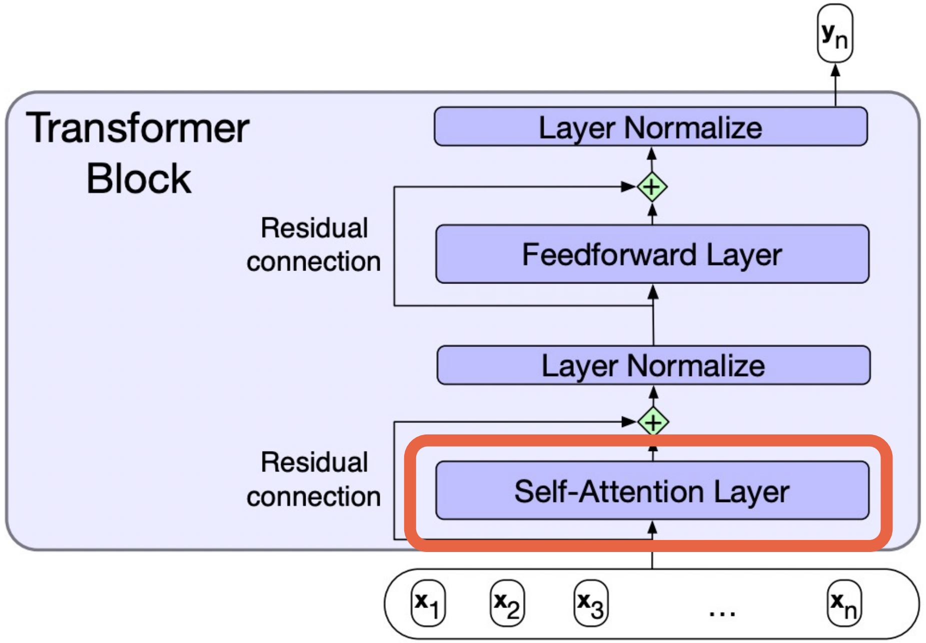
Self-attention: a closer look



But self-attention is just **one component** of the Transformer…

The Transformer "block"

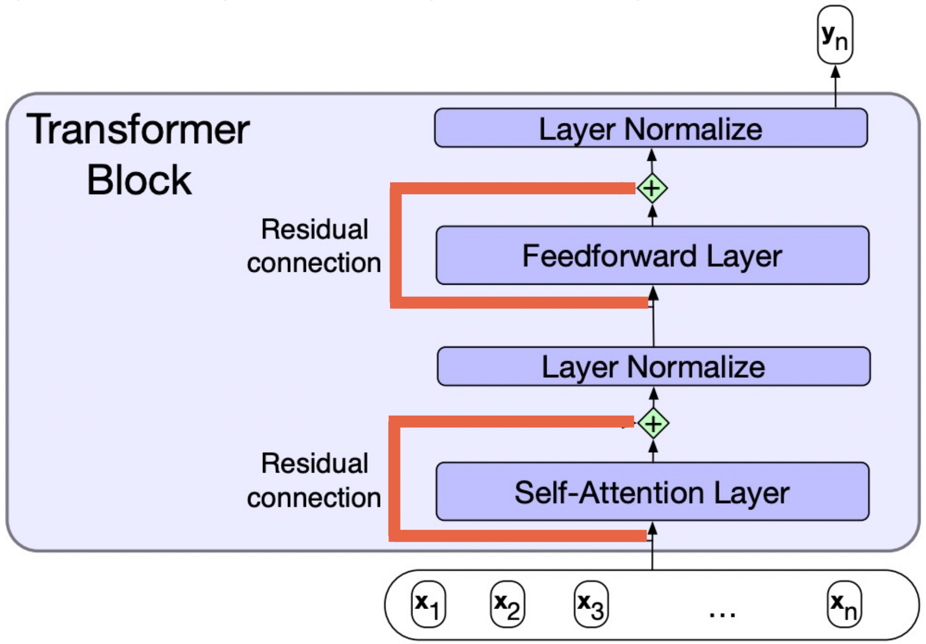
A **Transformer** "block" contains a self-attention layer, feed-forward layers, residual connections, and normalizing layers.



Self-attention: used to compute new, context-dependent representations for each token.

The Transformer “block”

A **Transformer** “block” contains a self-attention layer, feed-forward layers, residual connections, and normalizing layers.



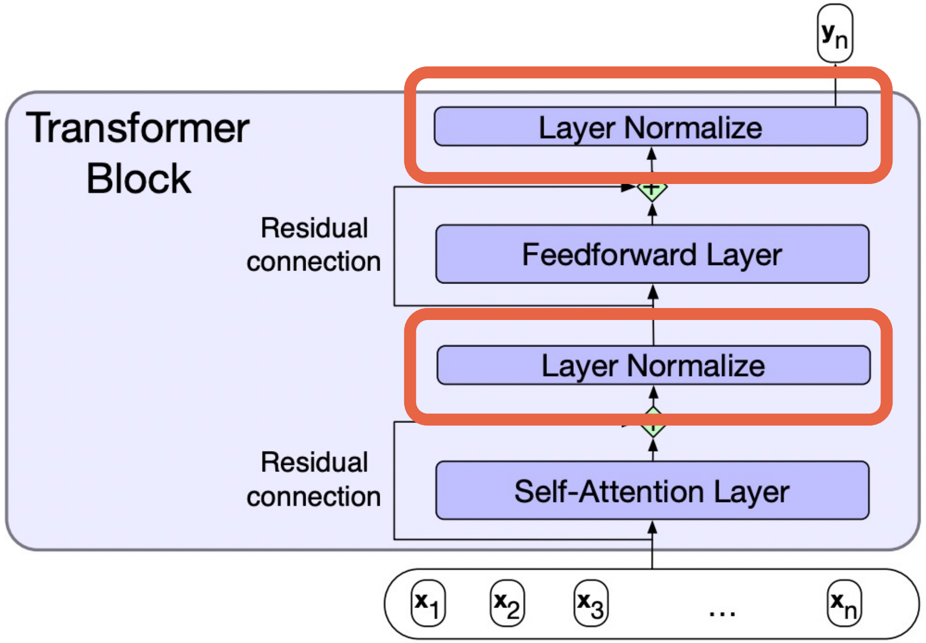
The **“residual connection”** projects directly from a lower layer to a higher layer, without passing through the intermediate layer.

To implement, add a layer’s *input* to its *output* before passing it forward.

`“dog” + Self-Attention(“dog”)`

The Transformer “block”

A **Transformer** “block” contains a self-attention layer, feed-forward layers, residual connections, and normalizing layers.

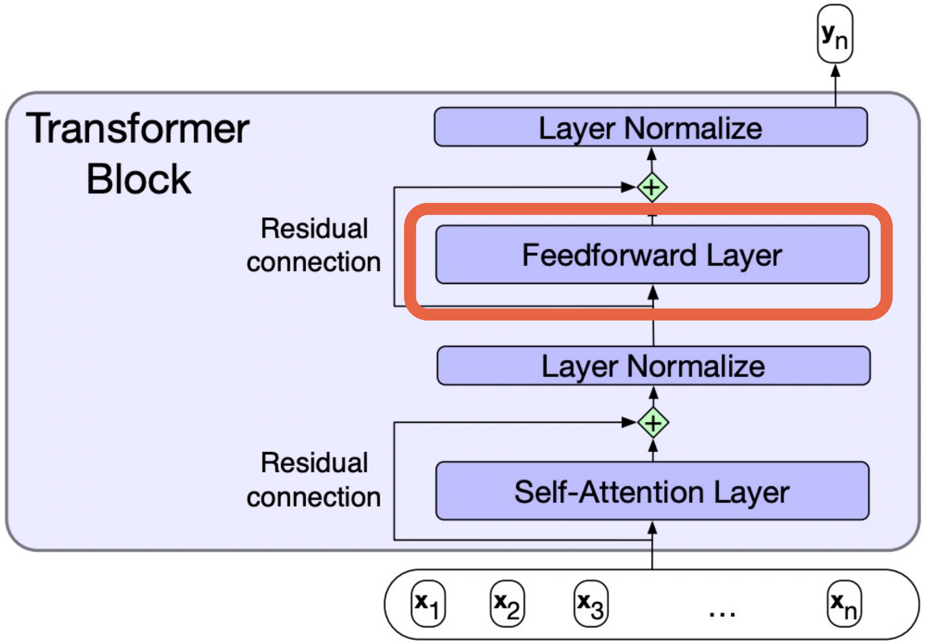


“Layer normalization” keeps the values of a hidden layer within a range that facilitates gradient-based training—similar to a *z*-score.

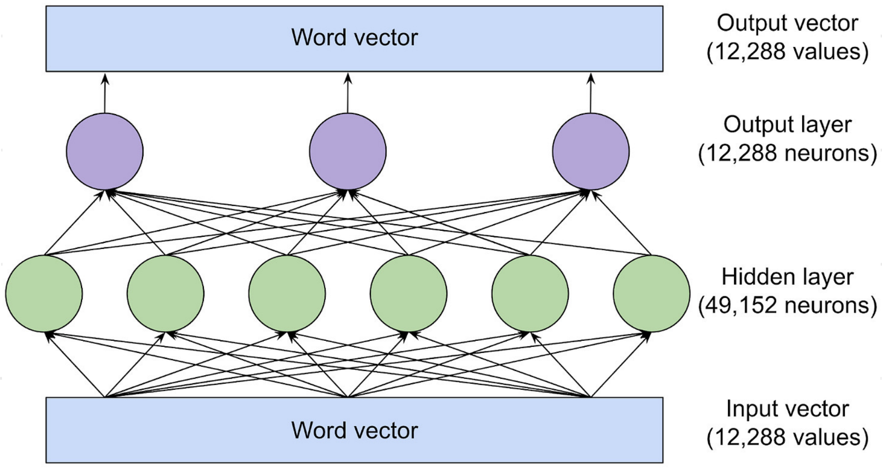
In GPT-2 and GPT-3, this FFN has **two layers**.

The Transformer "block"

A **Transformer** "block" contains a self-attention layer, feed-forward layers, residual connections, and normalizing layers.



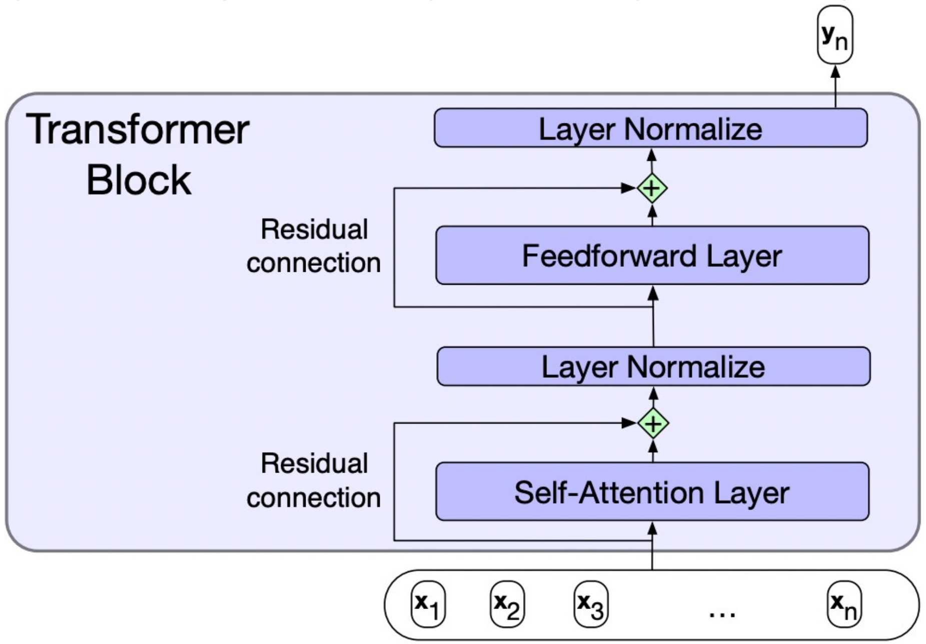
These vectors are then passed to a **feed-forward network**.



Schematic of FFN in GPT-3.

The Transformer "block"

A **Transformer** "block" contains a self-attention layer, feed-forward layers, residual connections, and normalizing layers.

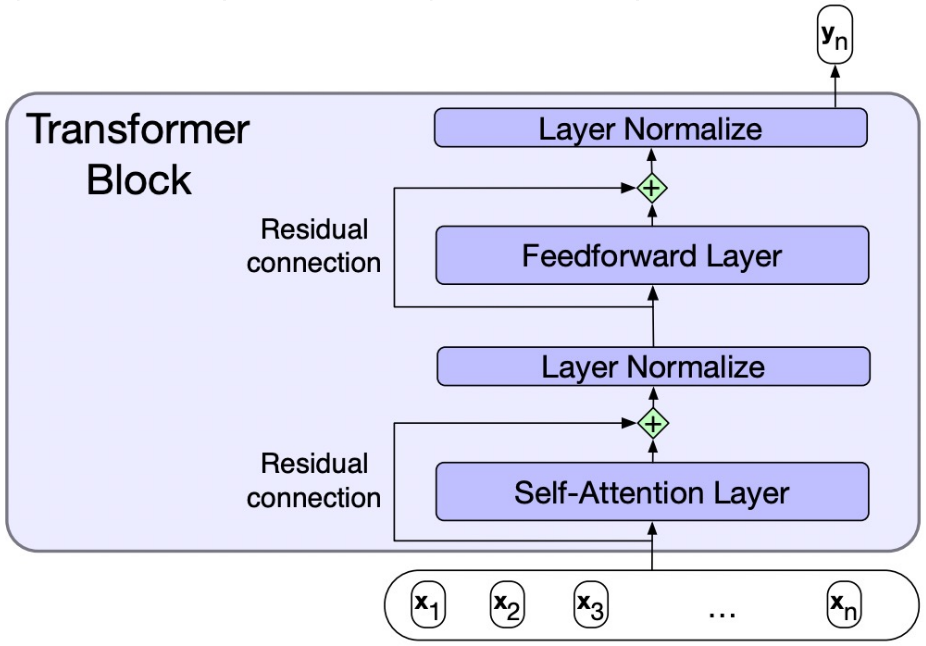


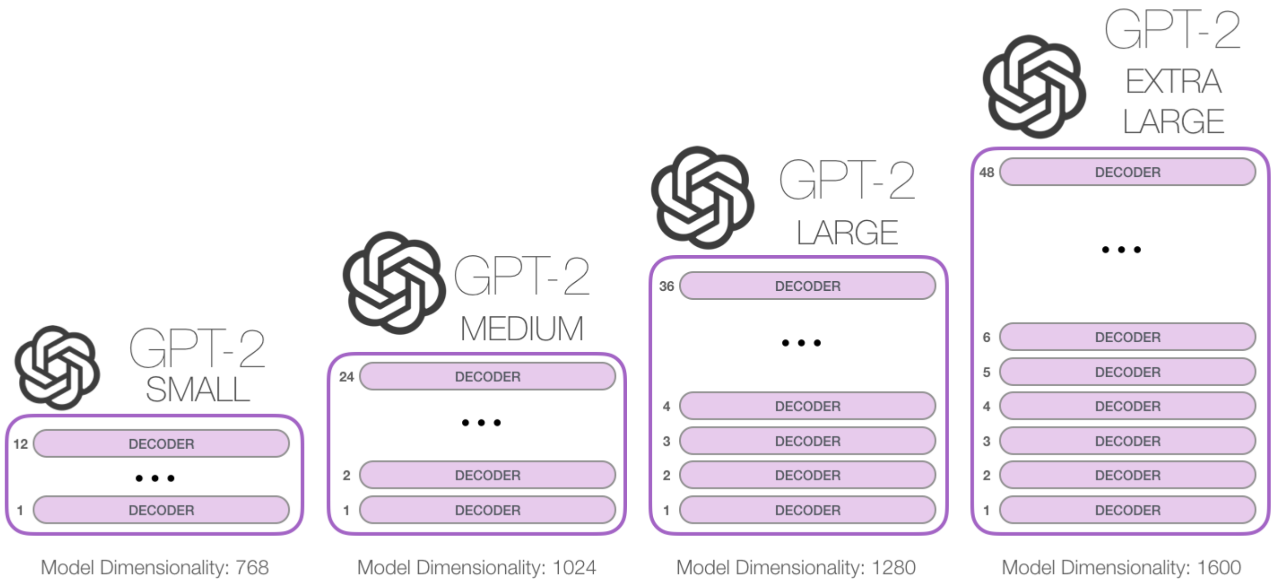
Also called *decoder blocks*.

Models like GPT-2 and GPT-3 have *many* of these!

The Transformer "block"

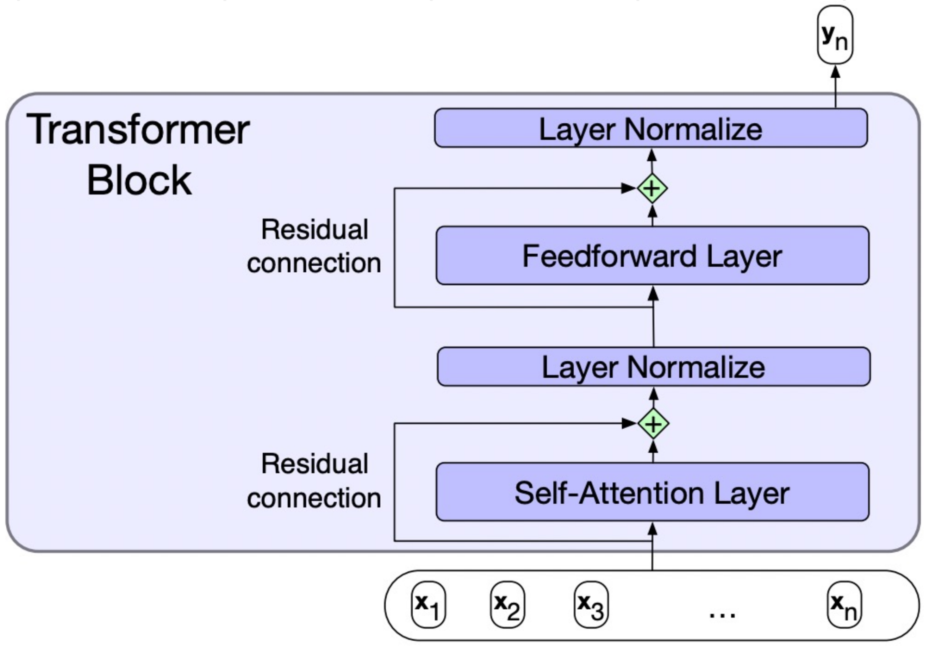
A **Transformer** "block" contains a self-attention layer, feed-forward layers, residual connections, and normalizing layers.





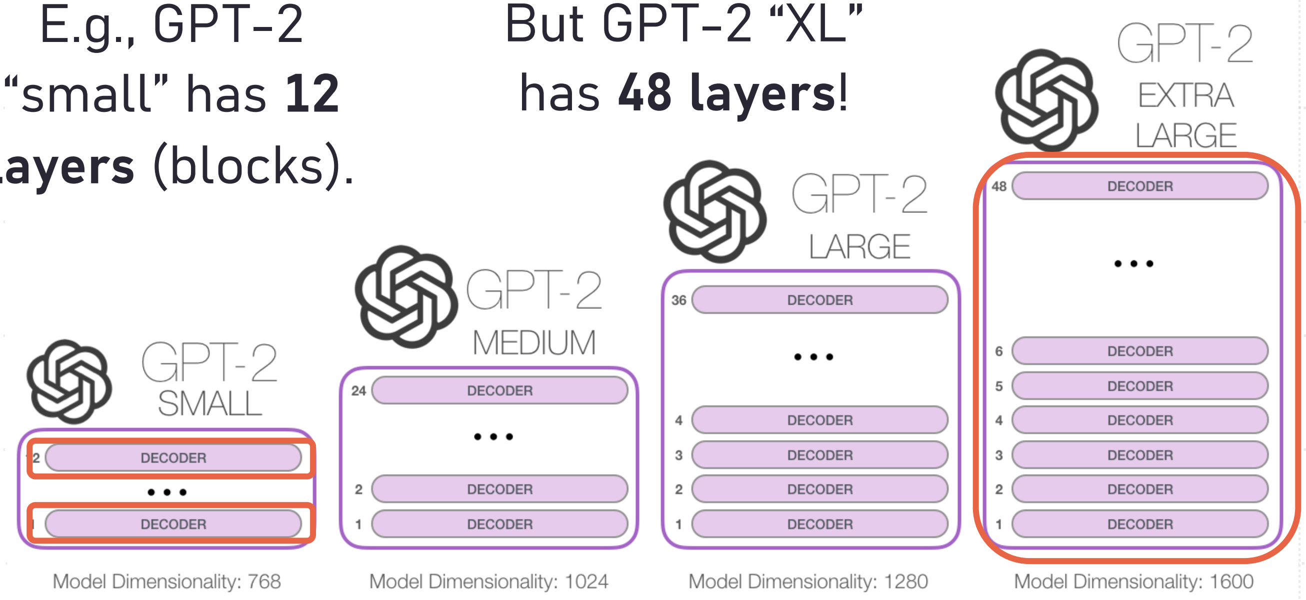
The Transformer “block”

A **Transformer** “block” contains a self-attention layer, feed-forward layers, residual connections, and normalizing layers.



E.g., GPT-2 “small” has **12 layers** (blocks).

But GPT-2 “XL” has **48 layers**!



"RNN + Attention—but throw out the RNN!"

Introducing transformers

The **Transformer** is a neural network architecture that uses multi-head self-attention, with no recurrent units.

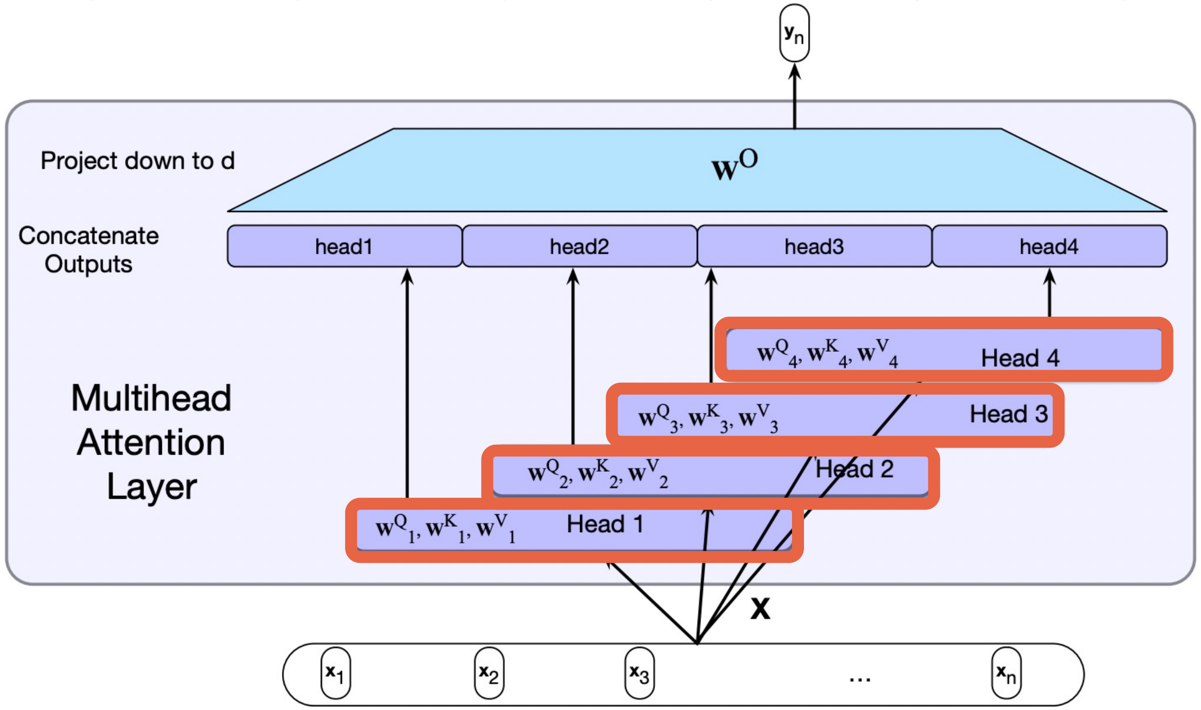
- Use a **fixed context window**.
- No **recurrent connections**.
- Use **self-attention**.
- Have multiple attention "heads" (**multi-head self-attention**).
- Use **positional embeddings**.

We've now covered self-attention—but what's "multi-head" attention?

When we discuss **probing** and **mechanistic interpretability**, we'll talk about research trying to figure out what these heads actually do!

Multi-head attention

In **multi-head attention**, each layer has multiple attention "heads", each with their own set of learnable weights for producing queries, keys, and values.



Each "head" might learn to track different kinds of relationships.

Over-simplified example:

- Maybe one head tracks syntax.
- Another head tracks proper names.
- Another head tracks events…

“RNN + Attention—but throw out the RNN!”

Introducing transformers

The **Transformer** is a neural network architecture that uses multi-head self-attention, with no recurrent units.

- Use a **fixed context window**.
- No **recurrent connections**.
- Use **self-attention**.
- Have multiple attention “heads” (**multi-head self-attention**).
- Use **positional embeddings**.

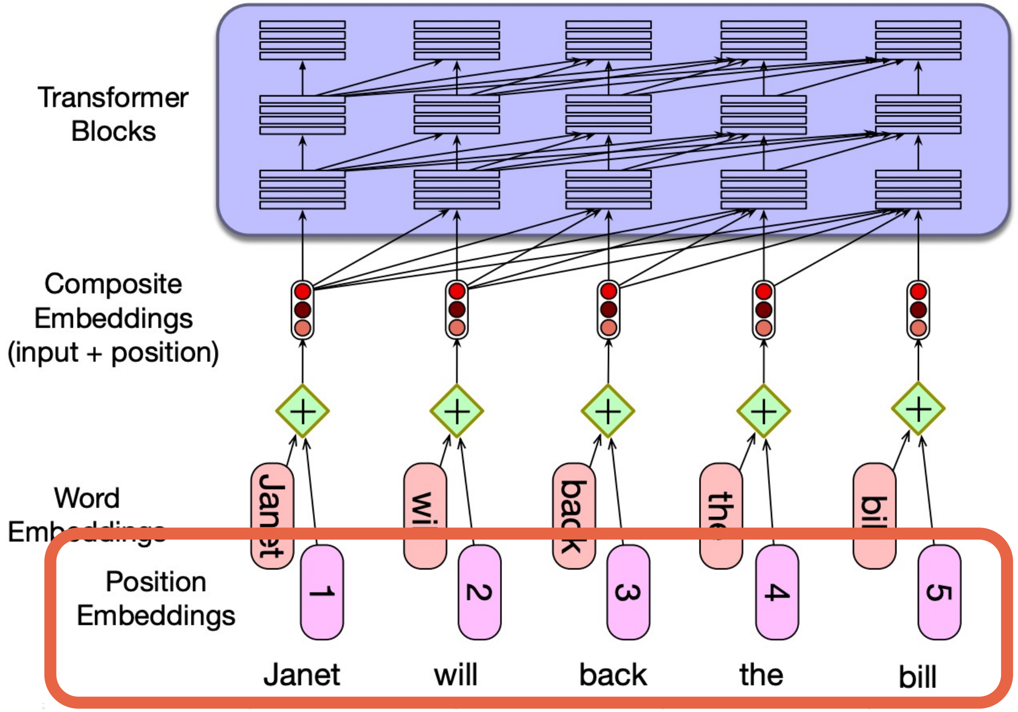
Okay, but what about the **order** of tokens?

With RNNs, order is built into the structure of the network.

Transformers use **positional embeddings** to track order.

Positional embeddings track order

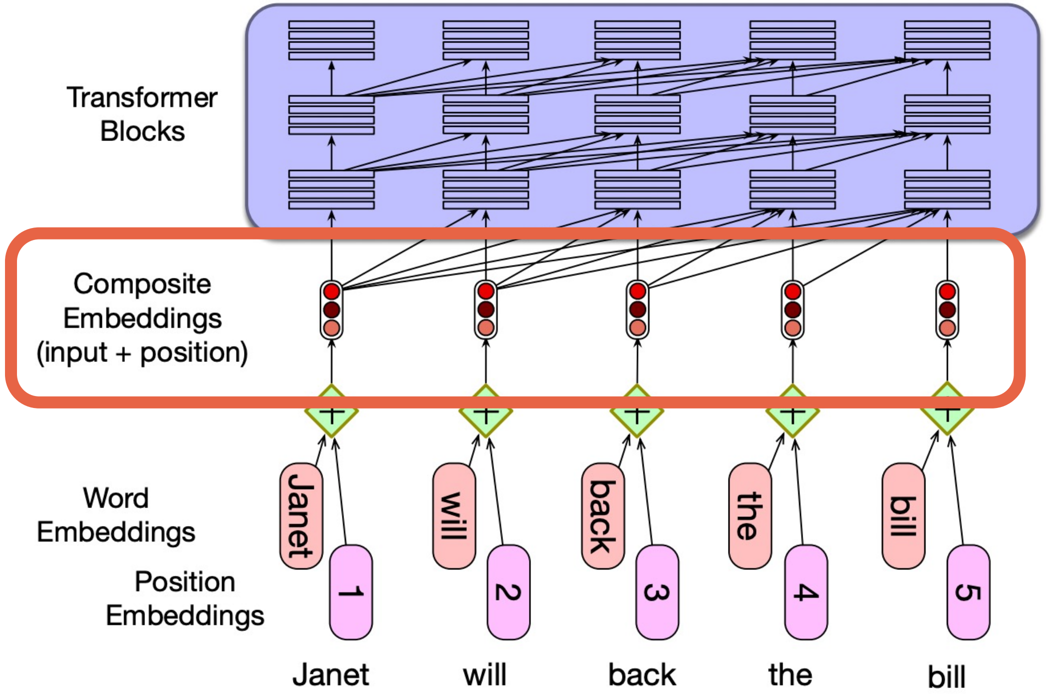
To represent order, input embeddings are combined with **positional embeddings** specific to each position in a sequence.



To learn, begin with random embeddings representing each "position" in a sequence (1, 2, 3, …)

Positional embeddings track order

To represent order, input embeddings are combined with **positional embeddings** specific to each position in a sequence.



To learn, begin with random embeddings representing each "position" in a sequence (1, 2, 3, …)

Once learned, we add positional embeddings with word embeddings.

Now, composite embeddings reflect both *word* and its *position*.

"RNN + Attention—but throw out the RNN!"

Introducing transformers

The **Transformer** is a neural network architecture that uses multi-head self-attention, with no recurrent units.

- Use a **fixed context window**.
- No **recurrent connections**.
- Use **self-attention**.
- Have multiple attention "heads" (**multi-head self-attention**).
- Use **positional embeddings**.

These are very complicated systems! Still lots to learn about why this architecture works.

One practical benefit is (so far) transformers are easier to train than RNNs.

"RNN + Attention—but throw out the RNN!"

Introducing transformers

The **Transformer** is a neural network architecture that uses multi-head self-attention, with no recurrent units.

- Use a **fixed context window**.
- No **recurrent connections**.
- Use **self-attention**.
- Have multiple attention "heads" (**multi-head self-attention**).
- Use **positional embeddings**.

Under the hood, ChatGPT uses a **transformer** model (plus some other stuff).

"RNN + Attention—but throw out the RNN!"

Introducing transformers

The **Transformer** is a neural network architecture that uses multi-head self-attention, with no recurrent units.

- Use a **fixed context window**.
- No **recurrent connections**.
- Use **self-attention**.
- Have multiple attention "heads" (**multi-head self-attention**).
- Use **positional embeddings**.

Under the hood, Chat**GPT** uses a **transformer** model (plus some other stuff).

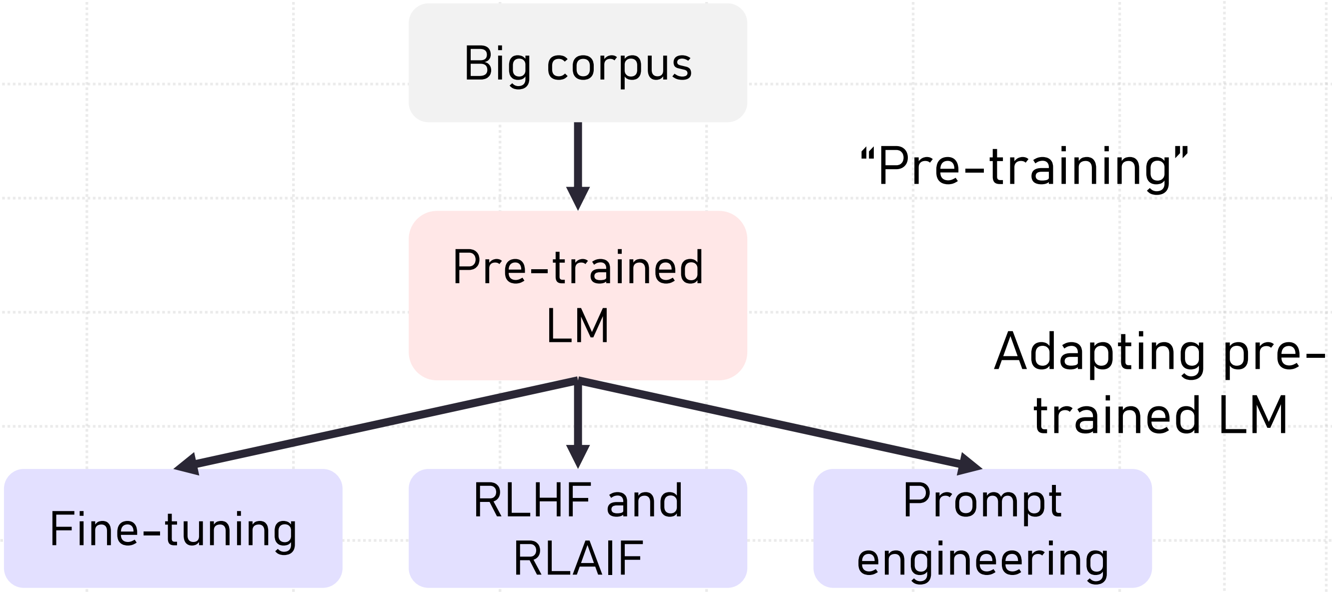
GPT = **G**enerative **P**re-trained **T**ransformer

So what's that "pre-trained" word mean…?

Pre-trained language models

A **pre-trained language model** is a (large) language model that's already been trained on a large corpus using self-supervision.

- "Pre-training" just means training **without a specific end goal in mind** (besides word prediction).
- A "pre-trained" LM can then be **adapted** for specific purposes.
- Practically, it's helpful so we **don't have to train from scratch**!



Pre-trained language models

A **pre-trained language model** is a (large) language model that's already been trained on a large corpus using self-supervision.

Next time, we'll discuss how to use pre-trained models in Python, using a library called **transformers**.

Summary

- **Self-attention** is a mechanism that allows each word to “look for” other words that are relevant in the input.
- This process creates new **context-dependent vectors** that share relevant information across the words in the input.
- Self-attention a key part part of the **“transformer block”**, which also has other features like a feed-forward network.
- So far, transformers tend to work better than other models like RNNs, and are **easier and faster to train**.
- “**Pre-training**” involves training a model (like a transformer) on a large corpus to learn the “basics” of how language works.