Machine Learning Department
School of Computer Science
Carnegie Mellon University

Transformer Language Models

Pat Virtue
Lecture 2
Jan. 16, 2025

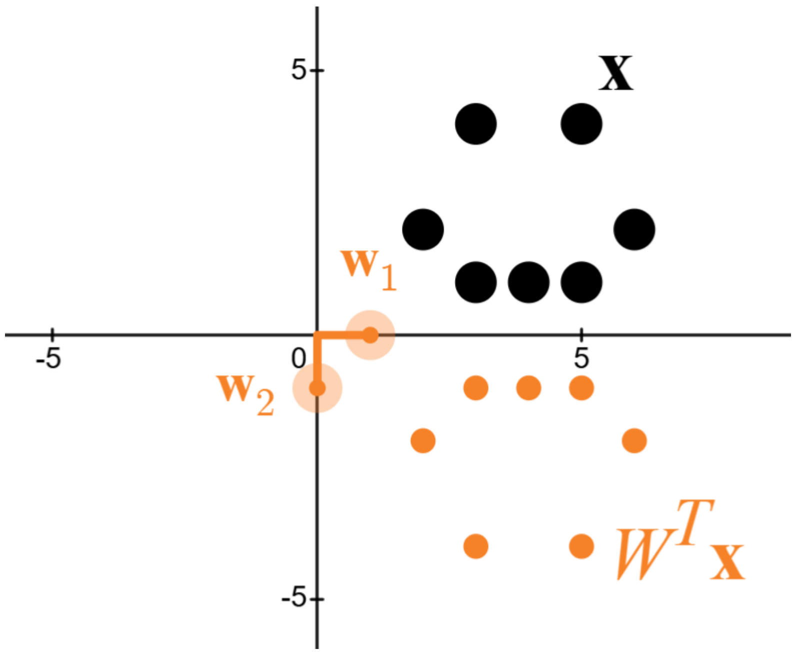
Slide credits: Matt Gormley

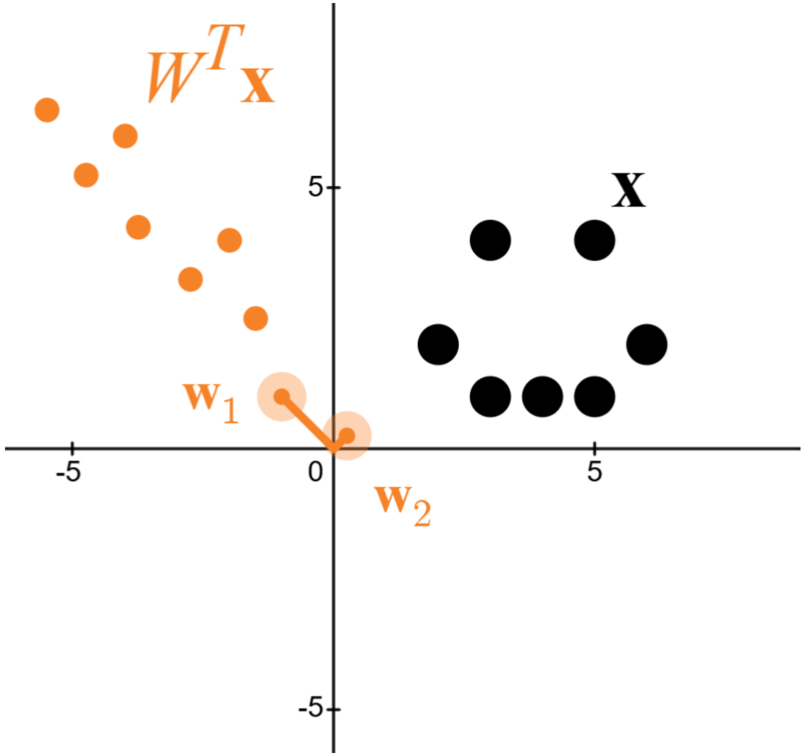
Reminders

- **Homework 0: PyTorch + Weights & Biases**
  - **Out: Wed, Jan 15**
  - **Due: Mon, Jan 27 at 11:59pm**
  - **Two parts:**
    1. **written part to Gradescope**
    2. **programming part to Gradescope**
  - unique policy for this assignment: we will grant (essentially) any and all extension requests, but you must request one

Linear Transforms: Graphical Intuition

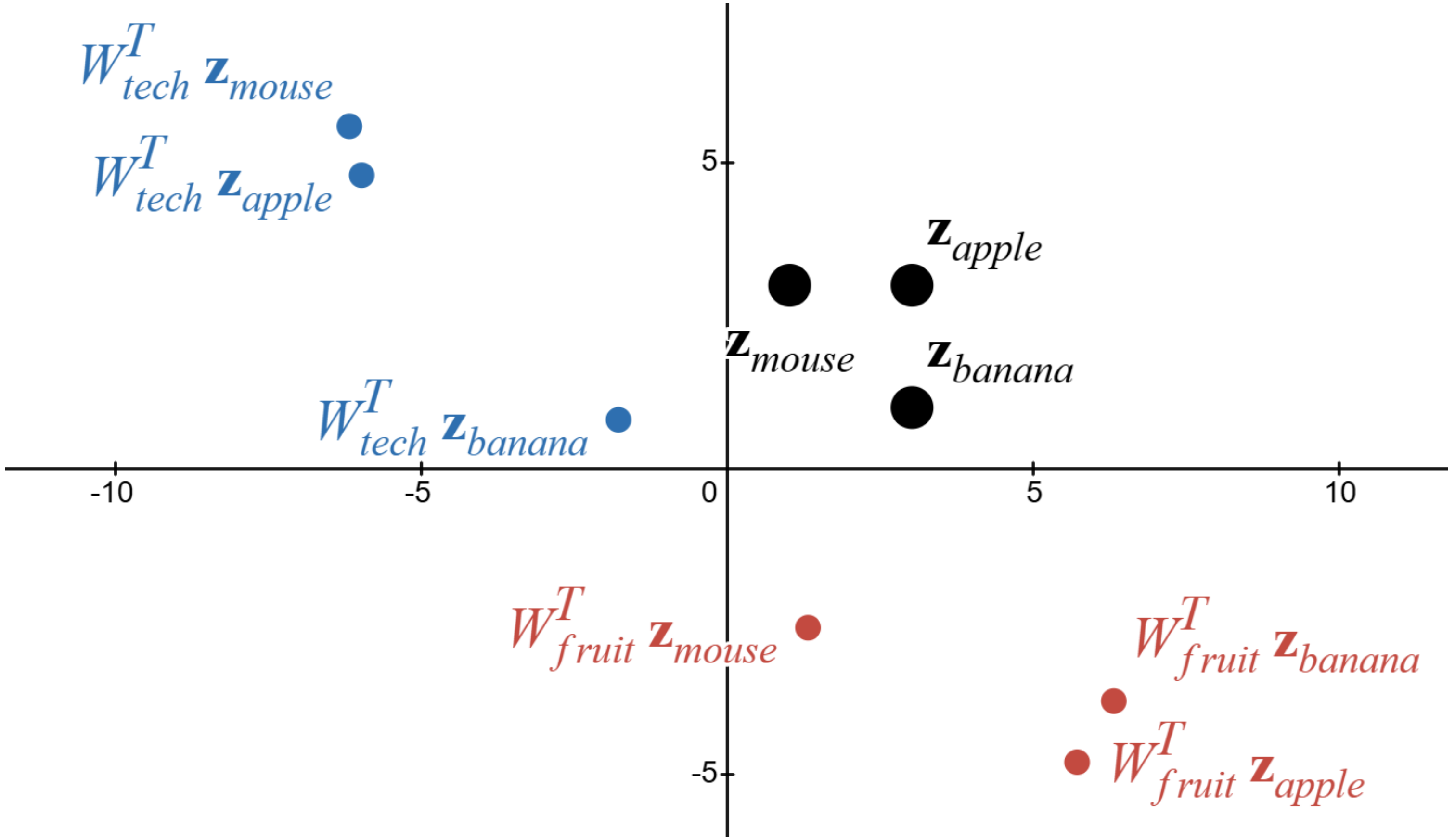
- In both RNNs and Transformer models, we see quite a few linear transforms
- A simple $\mathbf{z} = W^{\top}\mathbf{x}$ can move points quite a bit
- Desmos example for $\boldsymbol{x}$ and $\mathbf{z}$ in $\mathbb{R}^2$ https://www.desmos.com/calculator/gl5ljvorcy





Linear Transforms: Graphical Intuition

- Two different transforms $W_{tech}^{\top}\mathbf{z}$ and $W_{fruit}^{\top}\mathbf{z}$ can create two different meaningful embeddings for the input vectors $\mathbf{z}$
- Desmos example for W in $\mathbb{R}^{2\times 2}$ https://www.desmos.com/calculator/tbeclbo83h



Some History of…

LARGE LANGUAGE MODELS

Noisy Channel Models

- Prior to 2017, two tasks relied heavily on language models:
  - speech recognition
  - machine translation
- Definition: a **noisy channel model** combines a *transduction model* (probability of converting **y** to **x**) with a *language model* (probability of **y**)

transduction model — language model

$$\hat{\mathbf{y}} = \underset{\mathbf{y}}{\operatorname{argmax}}\, p(\mathbf{y} \mid \mathbf{x}) = \underset{\mathbf{y}}{\operatorname{argmax}}\, \underbrace{p(\mathbf{x} \mid \mathbf{y})}_{\text{transduction model}}\underbrace{p(\mathbf{y})}_{\text{language model}}$$

- **Goal:** to recover **y** from **x**
  - *For speech:* **x** is acoustic signal, **y** is transcription
  - *For machine translation:* **x** is sentence in source language, **y** is sentence in target language

Large (n-Gram) Language Models

- The earliest (truly) large language models were n-gram models
- Google n-Grams:
  - 2006: first release, English n-grams
    - trained on **1 trillion tokens** of web text (95 billion sentences)
    - included 1-grams, 2-grams, 3-grams, 4-grams, and 5-grams
  - 2009 – 2010: n-grams in Japanese, Chinese, Swedish, Spanish, Romanian, Portuguese, Polish, Dutch, Italian, French, German, Czech

English n-gram model is ~**3 billion parameters**

Number of unigrams:	13,588,391
Number of bigrams:	314,843,401
Number of trigrams:	977,069,902
Number of fourgrams:	1,313,818,354
Number of fivegrams:	1,176,470,663

serve as the incoming 92
serve as the incubator 99
serve as the independent 794
serve as the index 223
serve as the indication 72
serve as the indicator 120
serve as the indicators 45
serve as the indispensable 111
serve as the indispensible 40
serve as the individual 234
serve as the industrial 52
serve as the industry 607

accessoire Accessoires </S> 515
accessoire Accord i-CTDi 65
accessoire Accra accu 312
accessoire Acheter cet 1402
accessoire Ajouter au 160
accessoire Amour Beauté 112
accessoire Annuaire LOEIL 49
accessoire Architecture artiste 531
accessoire Attention : 44

惯例 为 电影 创作 52
惯例 为 的 是 95
惯例 为 目标 职位 49
惯例 为 确保 合作 69
惯例 为 确保 重组 213
惯例 为 科研 和 55
惯例 为 统称 </S> 183
惯例 为 维 和 50
惯例 为 自己 的 43
惯例 为 艺术类 学院 44
惯例 为 避免 侵权 148

Large (n-Gram) Language Models

- The earliest (truly) large language models were n-gram models
- Google n-Grams:
  - 2006: first release, English n-grams
    - trained on **1 trillion tokens** of web text (95 billion sentences)
    - included 1-grams, 2-grams, 3-grams, 4-grams, and 5-grams
  - 2009 – 2010: n-grams in Japanese, Chinese, Swedish, Spanish, Romanian, Portuguese, Polish, Dutch, Italian, French, German, Czech

English n-gram model is ~**3 billion parameters**

Number of unigrams:	13,588,391
Number of bigrams:	314,843,401
Number of trigrams:	977,069,902
Number of fourgrams:	1,313,818,354
Number of fivegrams:	1,176,470,663

Q: Is this a large training set?

A: Yes!

e Accessoires </S> 5
e Accord i-CTDi 65
e Accra accu 312
e Acheter cet 1402
e Ajouter au 160
e Amour Beauté 112
e Annuaire LOEIL 49
e Architecture artist
e Attention : 44

Q: Is this a large model?

A: Yes!

serve as the individual 234
serve as the industrial 52
serve as the industry 607

惯例 为 艺术类 学院 44
惯例 为 避免 侵权 148

How large are LLMs?

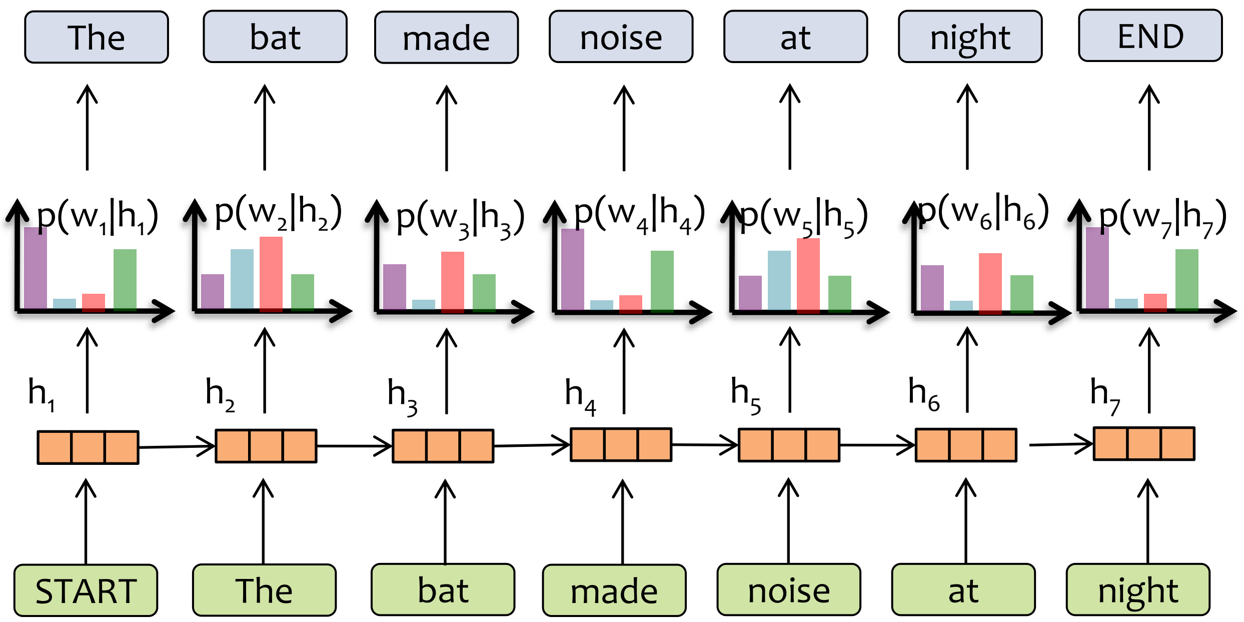
Comparison of some recent **large language models** (LLMs)

Model	Creators	Year of release	Training Data (# tokens)	Model Size (# parameters)
GPT-2	OpenAI	2019	~10 billion (40Gb)	1.5 billion
GPT-3	OpenAI	2020	300 billion	175 billion
PaLM	Google	2022	780 billion	540 billion
Chinchilla	DeepMind	2022	1.4 trillion	70 billion
LaMDA (cf. Bard)	Google	2022	1.56 trillion	137 billion
LLaMA	Meta	2023	1.4 trillion	65 billion
LLaMA-2	Meta	2023	2 trillion	70 billion
GPT-4	OpenAI	2023	?	? (1.76 trillion)
Gemini (Ultra)	Google	2023	?	? (1.5 trillion)
LLaMA-3	Meta	2024	15 trillion	405 billion

FORGETFUL RNNS

RNN Language Model

Recall...



Key Idea:

(1) convert all previous words to a **fixed length vector**

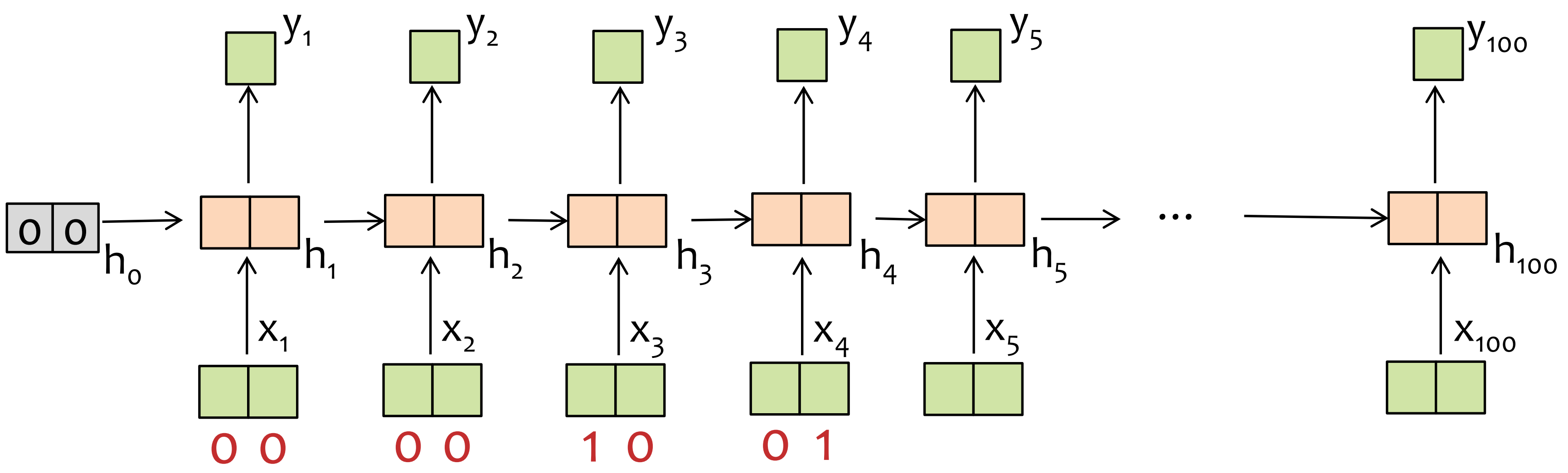
(2) define distribution $p(w_t \mid f_\theta(w_{t-1}, \dots, w_1))$ that conditions on the vector $\mathbf{h}_t = f_\theta(w_{t-1}, \dots, w_1)$

RNNs and Forgetting

Suppose we want an RNN over binary vectors of length 2 that can remember whether or not it has seen a value of 1 in both input positions.

$$\mathbf{h}_t = \sigma(W_{hh}\mathbf{h}_{t-1} + W_{hx}\mathbf{x_t} + \mathbf{b}_h) \qquad W_{hh} = \begin{bmatrix} 1/2 & 0 \\ 0 & 1/2 \end{bmatrix} \quad W_{hx} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \mathbf{b}_h = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

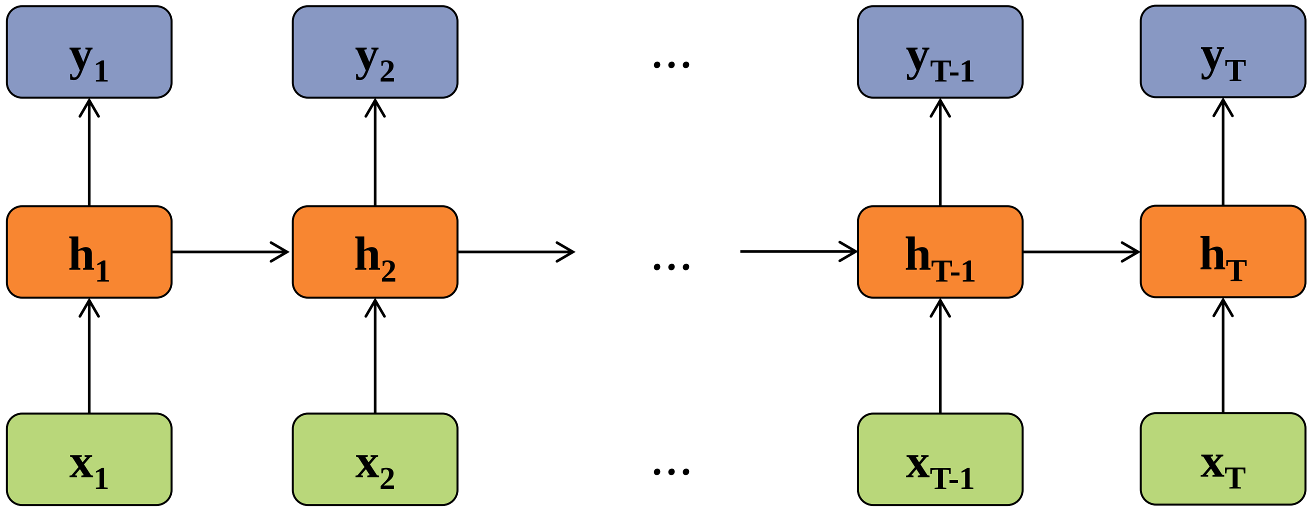
$$y_t = sign\left(W_{yh}\mathbf{h}_t + \mathbf{b}_y\right) \qquad W_{yh} = \begin{bmatrix} \quad \end{bmatrix} \quad \mathbf{b}_y = \begin{bmatrix} \quad \end{bmatrix}$$



Long Short-Term Memory (LSTM)

Motivation:

- Standard RNNs have trouble learning long distance dependencies
- LSTMs combat this issue



Long Short-Term Memory (LSTM)

Motivation:

- Vanishing gradient problem for Standard RNNs
- Figure shows sensitivity (darker = more sensitive) to the input at time t=1

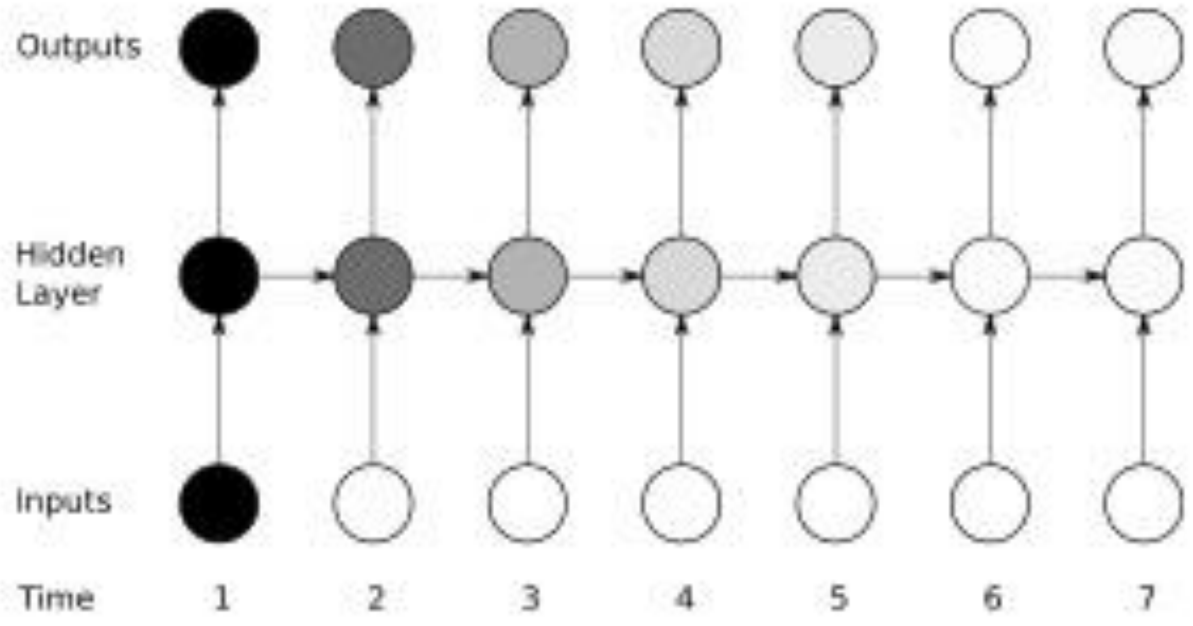


Figure from (Graves, 2012)

Long Short-Term Memory (LSTM)

Motivation:

- LSTM units have a rich internal structure
- The various "gates" determine the propagation of information and can choose to "remember" or "forget" information

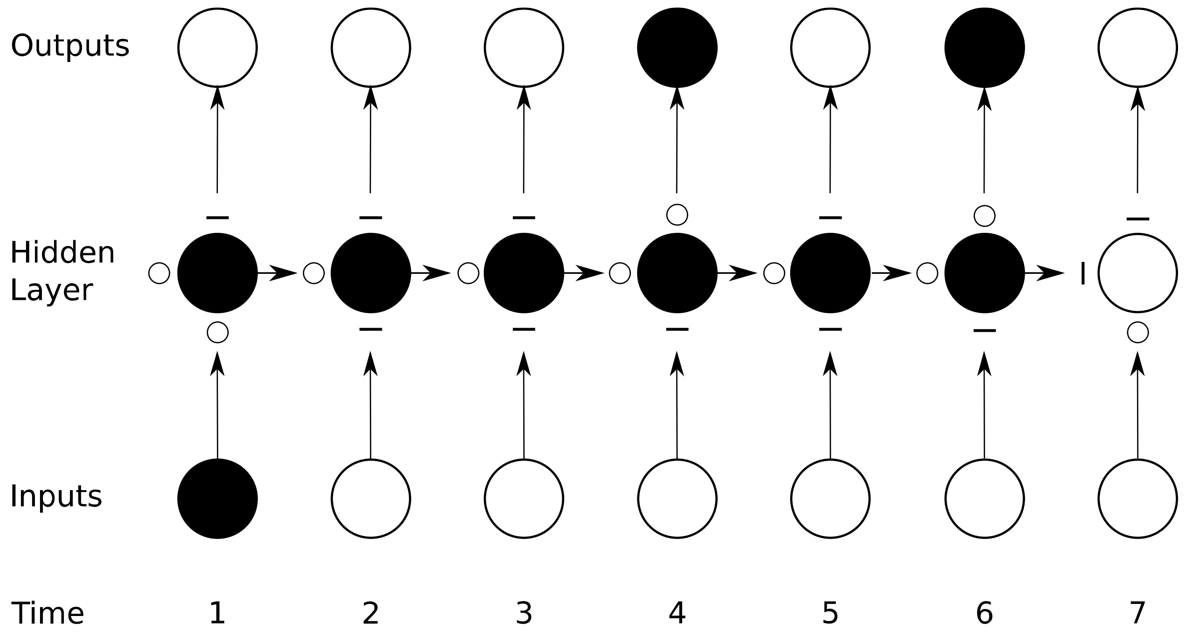
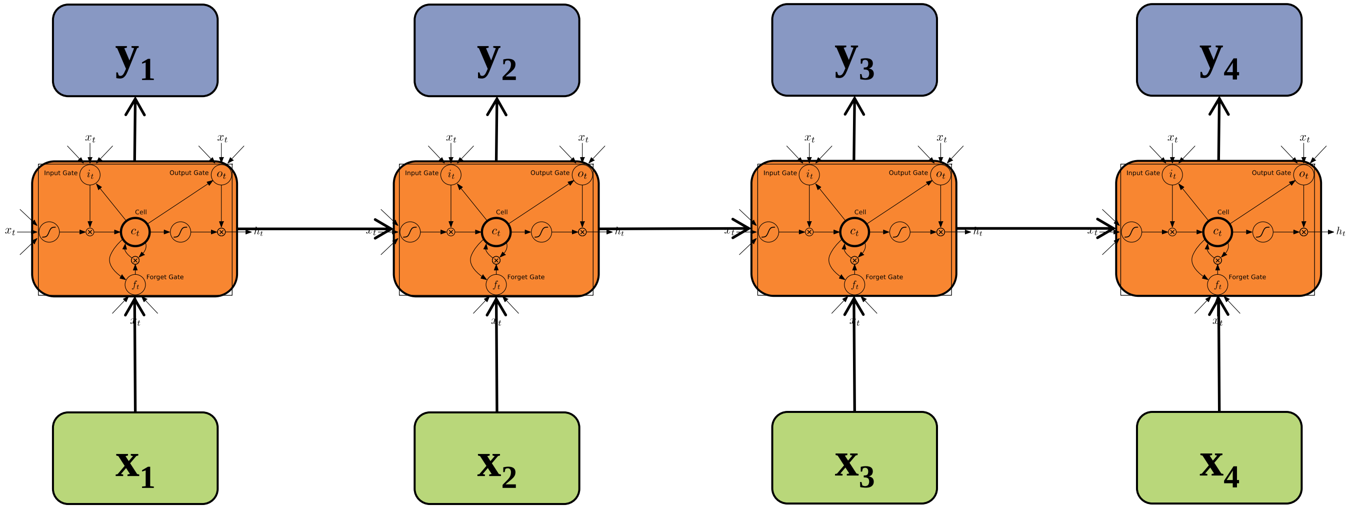


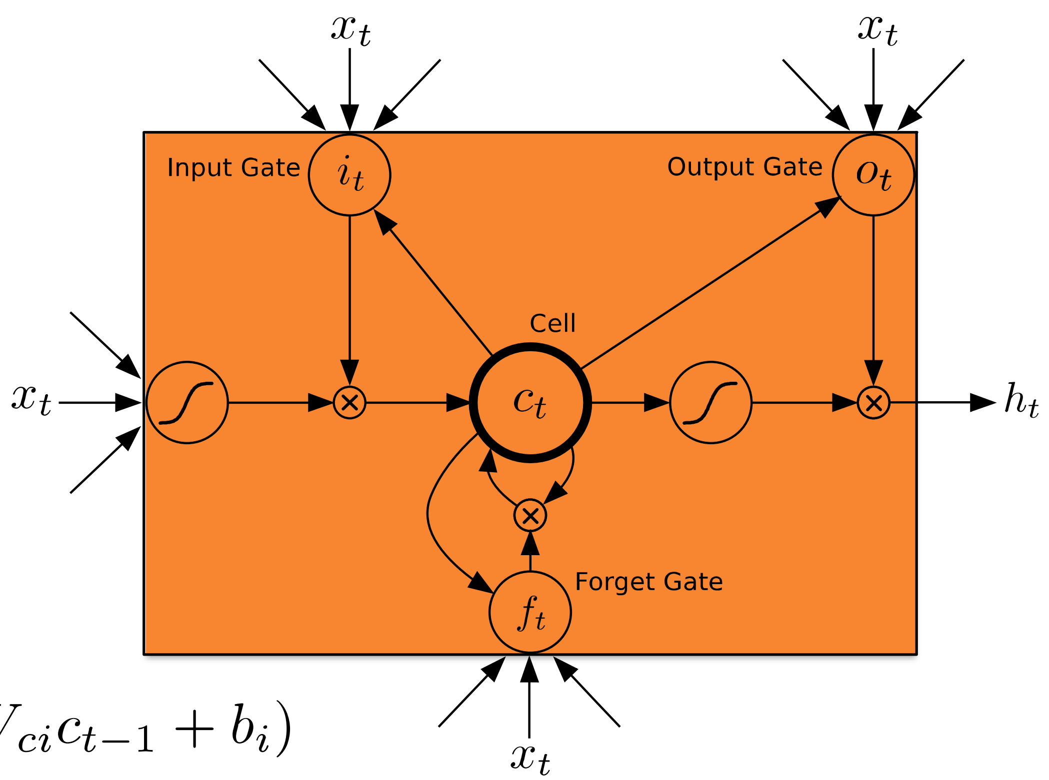
Figure from (Graves, 2012)

Long Short-Term Memory (LSTM)



Long Short-Term Memory (LSTM)

- **Input gate:** masks out the standard RNN inputs
- **Forget gate:** masks out the previous cell
- **Cell:** stores the input/forget mixture
- **Output gate:** masks out the values of the next hidden



$$i_t = \sigma\left(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i\right)$$

$$f_t = \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f\right)$$

$$c_t = f_t c_{t-1} + i_t \tanh\left(W_{xc}x_t + W_{hc}h_{t-1} + b_c\right)$$

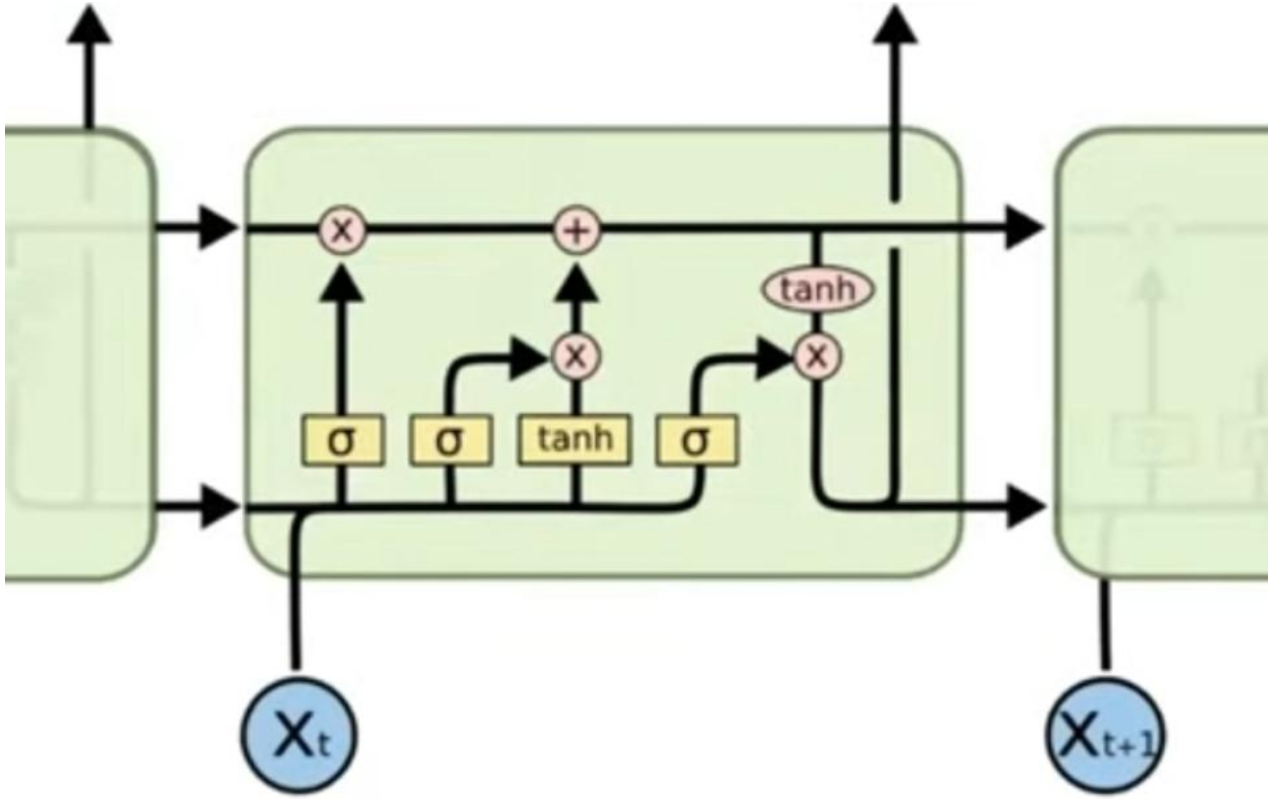
$$o_t = \sigma\left(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o\right)$$

$$h_t = o_t \tanh(c_t)$$

Figure from (Graves et al., 2013)

Long Short-Term Memory (LSTM)

- **Input gate:** masks out the standard RNN inputs
- **Forget gate:** masks out the previous cell
- **Cell:** stores the input/forget mixture
- **Output gate:** masks out the values of the next hidden



i_t :

f_t :

c_t :

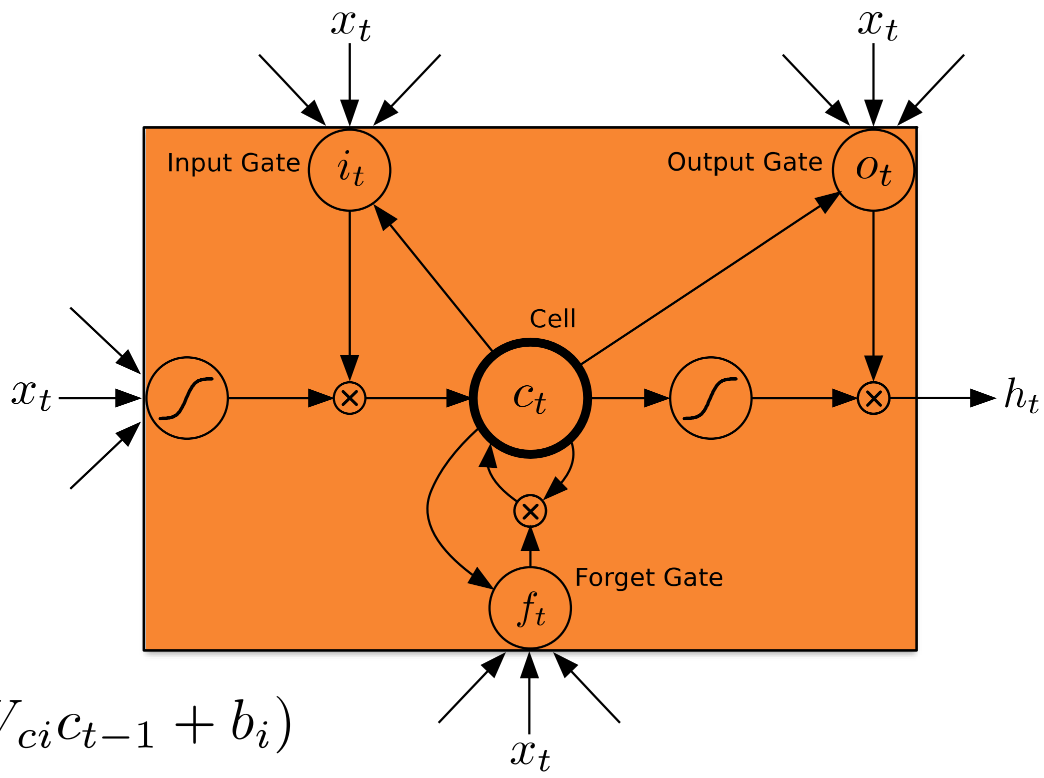
o_t :

h_t :

Figure from Bhiksha Raj CMU 11-785

Long Short-Term Memory (LSTM)

- **Input gate:** masks out the standard RNN inputs
- **Forget gate:** masks out the previous cell
- **Cell:** stores the input/forget mixture
- **Output gate:** masks out the values of the next hidden



The cell is the LSTM's long term memory, and helps control information flow over time steps

The hidden state is the output of the LSTM cell

$$i_t = \sigma\left(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i\right)$$

$$f_t = \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f\right)$$

$$c_t = f_t c_{t-1} + i_t \tanh\left(W_{xc}x_t + W_{hc}h_{t-1} + b_c\right)$$

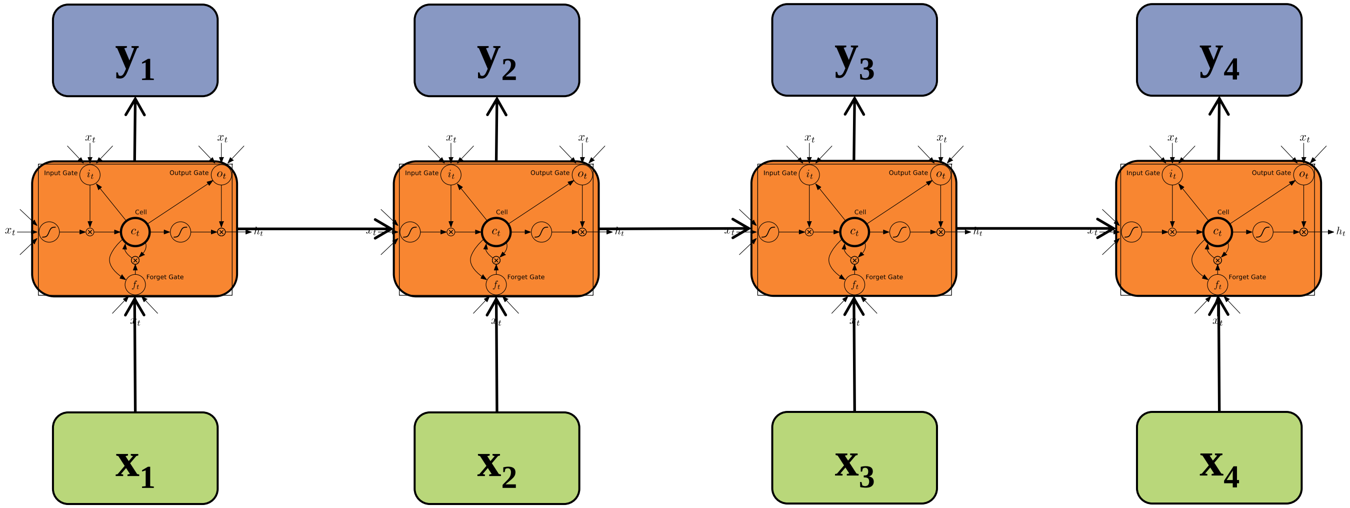
$$o_t = \sigma\left(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o\right)$$

$$h_t = o_t \tanh(c_t)$$

Identical to the Elman's networks hidden state

Figure from (Graves et al., 2013)

Long Short-Term Memory (LSTM)



Bidirectional RNN

inputs: $\mathbf{x} = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I$

hidden units: $\overrightarrow{\mathbf{h}}$ and $\overleftarrow{\mathbf{h}}$

outputs: $\mathbf{y} = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K$

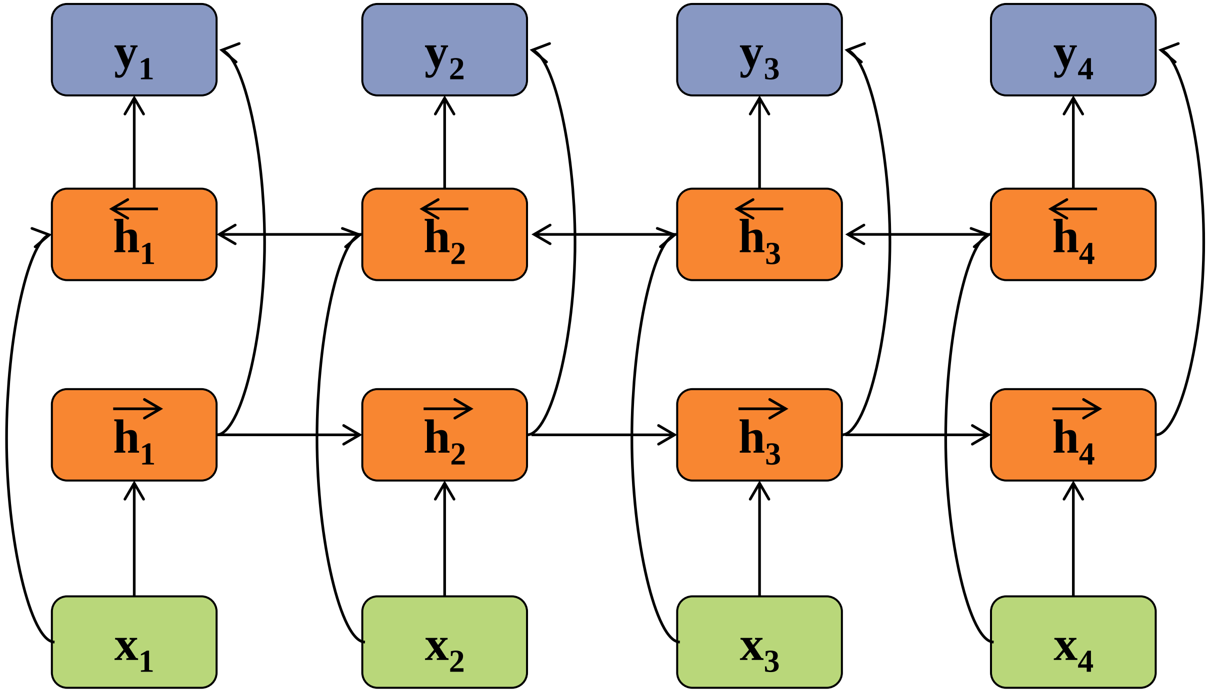
nonlinearity: $\mathcal{H}$

Recursive Definition:

$$\overrightarrow{h}_t = \mathcal{H}\left(W_{x\overrightarrow{h}} x_t + W_{\overrightarrow{h}\overrightarrow{h}} \overrightarrow{h}_{t-1} + b_{\overrightarrow{h}}\right)$$

$$\overleftarrow{h}_t = \mathcal{H}\left(W_{x\overleftarrow{h}} x_t + W_{\overleftarrow{h}\overleftarrow{h}} \overleftarrow{h}_{t+1} + b_{\overleftarrow{h}}\right)$$

$$y_t = W_{\overrightarrow{h}y} \overrightarrow{h}_t + W_{\overleftarrow{h}y} \overleftarrow{h}_t + b_y$$



Deep RNNs

inputs: $\mathbf{x} = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I$

outputs: $\mathbf{y} = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K$

nonlinearity: $\mathcal{H}$

Recursive Definition:

$$h_t^n = \mathcal{H}\left(W_{h^{n-1}h^n} h_t^{n-1} + W_{h^n h^n} h_{t-1}^n + b_h^n\right)$$

$$y_t = W_{h^N y} h_t^N + b_y$$

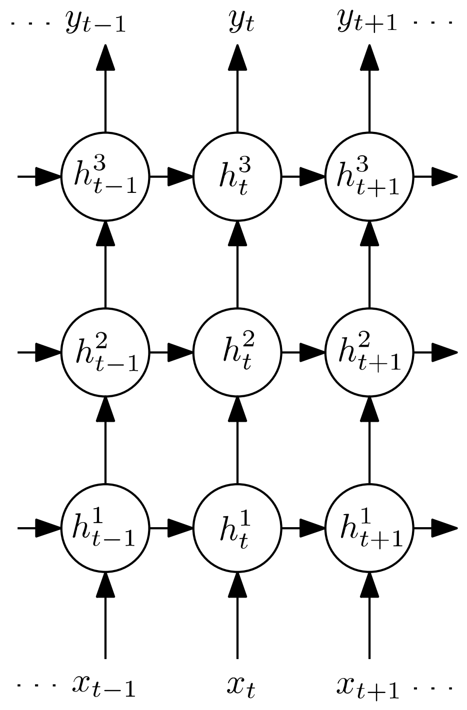
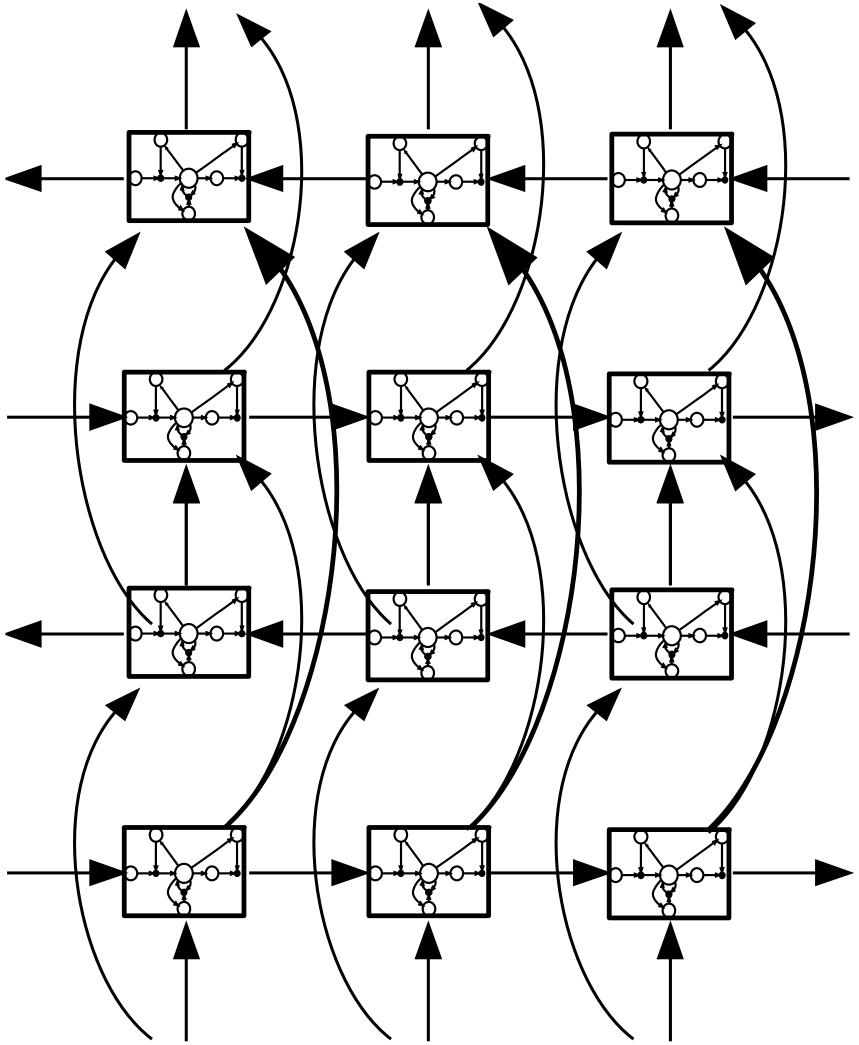


Figure from (Graves et al., 2013)

Deep Bidirectional LSTM (DBLSTM)



- Figure: input/output layers not shown
- **Same general topology** as a Deep Bidirectional RNN, but with **LSTM units** in the hidden layers
- No additional **representational power** over DBRNN, but **easier to learn** in practice

Figure from (Graves et al., 2013)

Deep Bidirectional LSTM (DBLSTM)

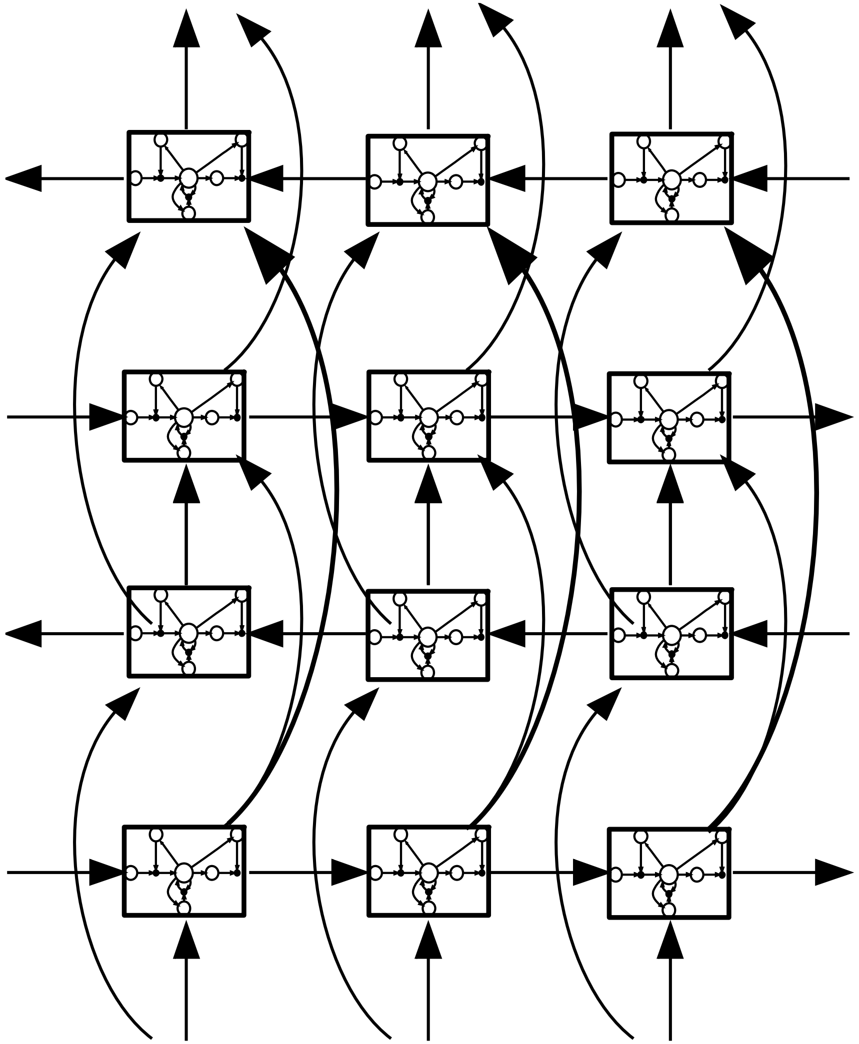


Figure from (Graves et al., 2013)

How important is this particular architecture?

Jozefowicz et al. (2015) **evaluated 10,000 different LSTM-like architectures** and found several variants that worked just as well on several tasks.

Why not just use LSTMs for everything?

Everyone did, for a time.

But…

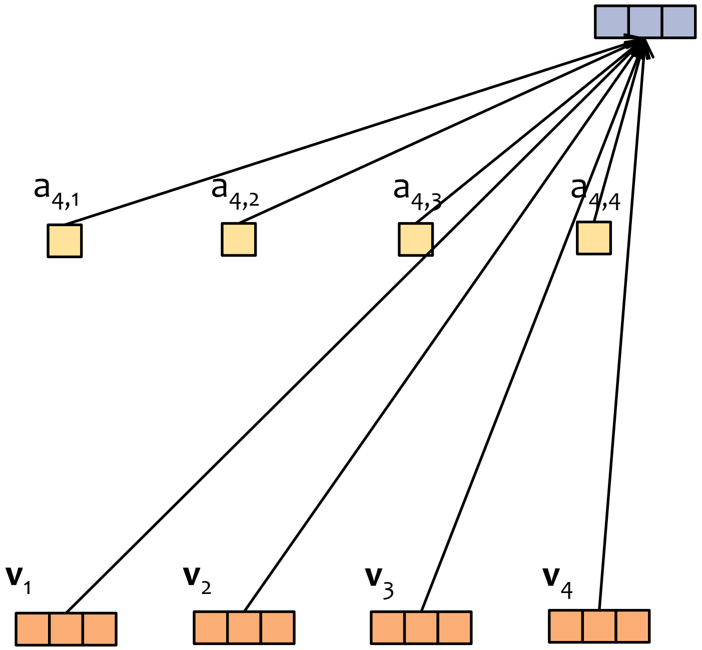
1. They still have **difficulty** with **long-range dependencies**
2. Their computation is **inherently serial,** so can't be easily parallelized on a GPU
3. Even though they (mostly) solve the vanishing gradient problem, they can still suffer from **exploding gradients**

Transformer Language Models

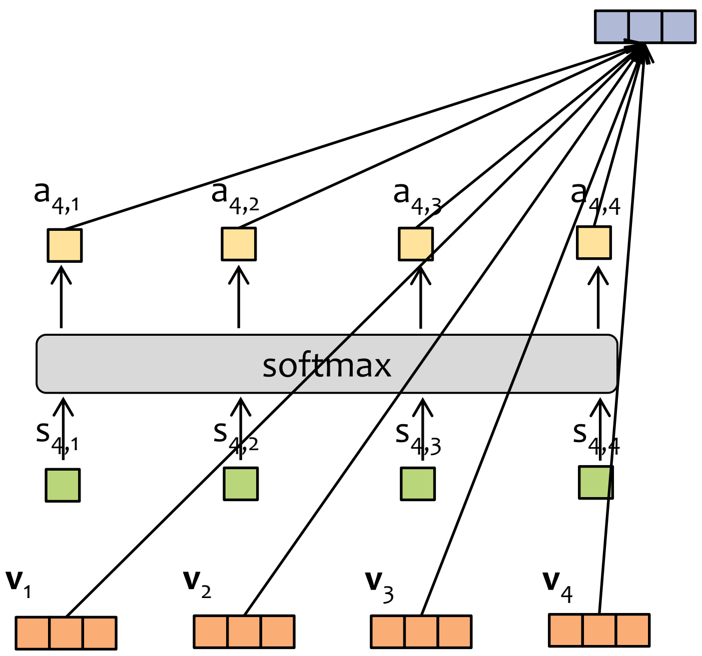
MODEL: GPT

Attention

$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$$



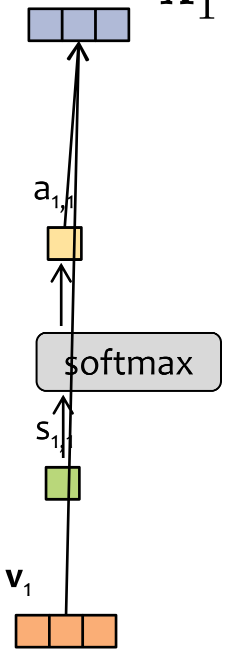
Attention



$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$$

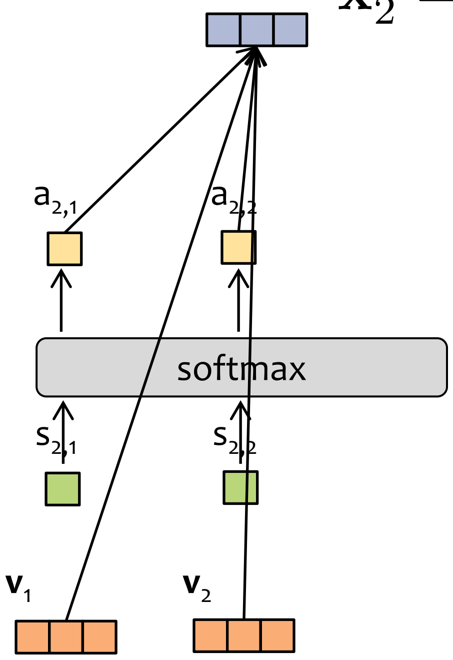
Attention

$$\mathbf{x}'_1 = \sum_{j=1}^{1} a_{1,j}\mathbf{v}_j$$

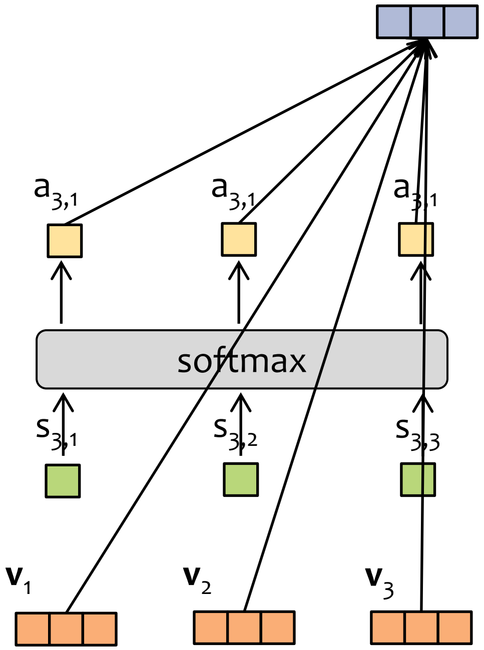


Attention

$$\mathbf{x}'_2 = \sum_{j=1}^{2} a_{2,j}\mathbf{v}_j$$



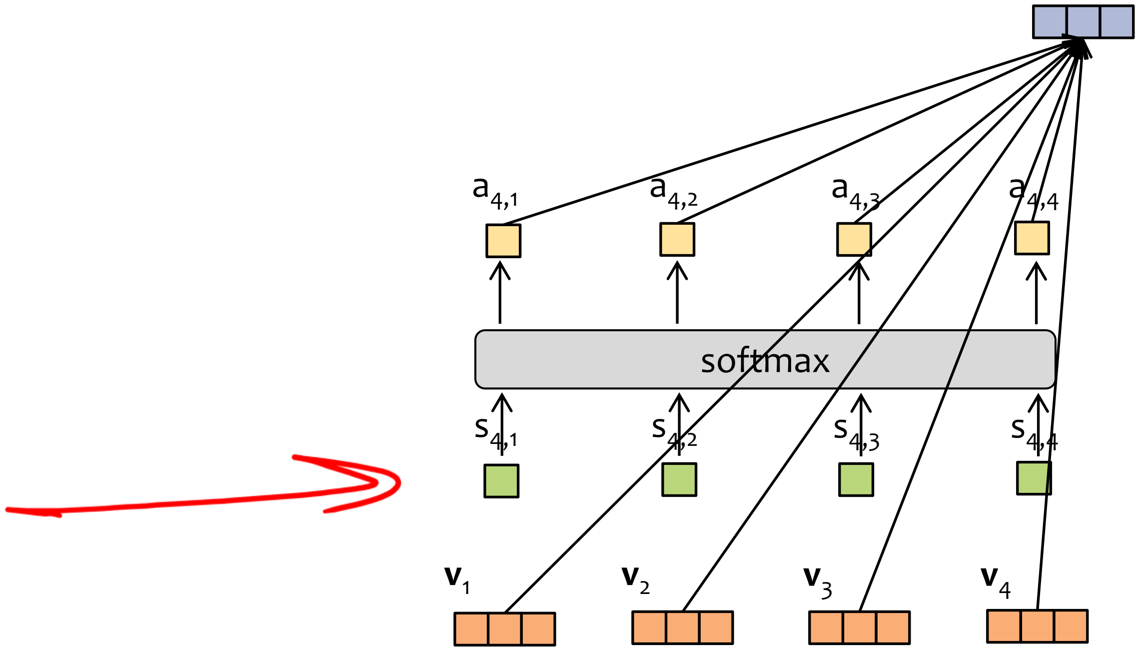
Attention



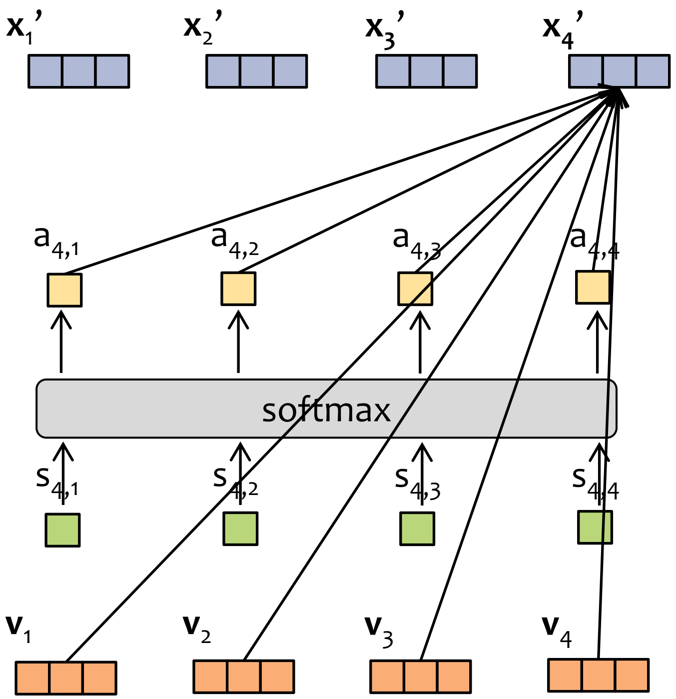
$$\mathbf{x}'_3 = \sum_{j=1}^{3} a_{3,j}\mathbf{v}_j$$

Attention

$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j} \mathbf{v}_j$$



Attention



$$\mathbf{x}'_t = \sum_{j=1}^{t} a_{t,j} \mathbf{v}_j$$

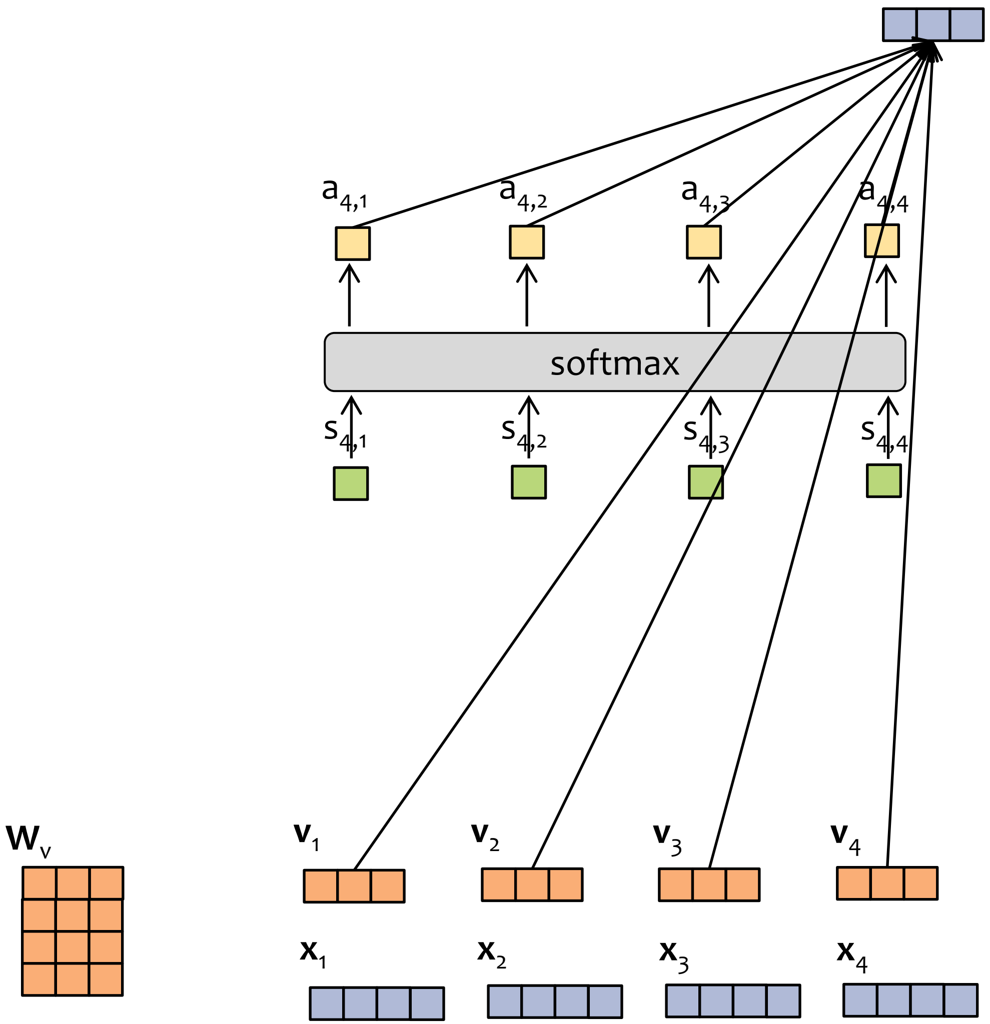
attention weights

scores

values

Scaled Dot-Product Attention

$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$$

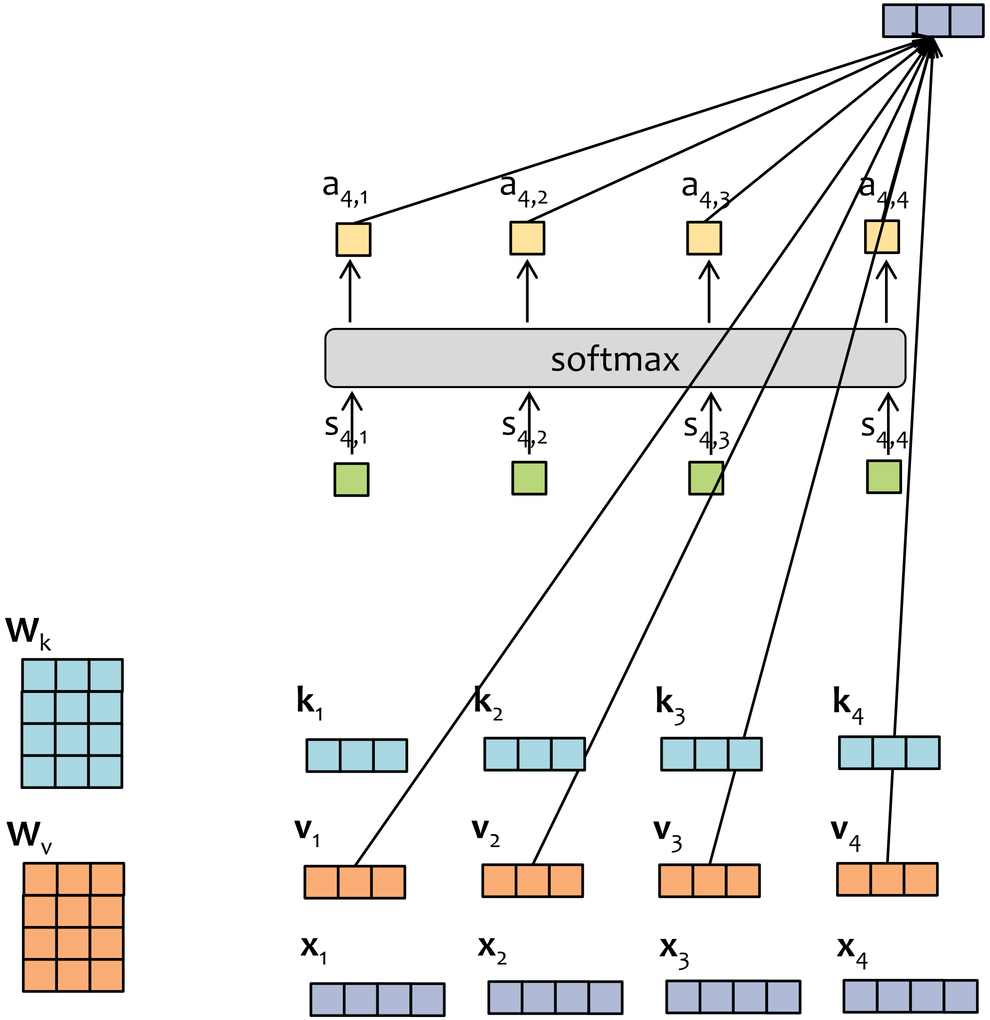


$$\mathbf{v}_j = \mathbf{W}_v^T \mathbf{x}_j$$

values

Scaled Dot-Product Attention

$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$$

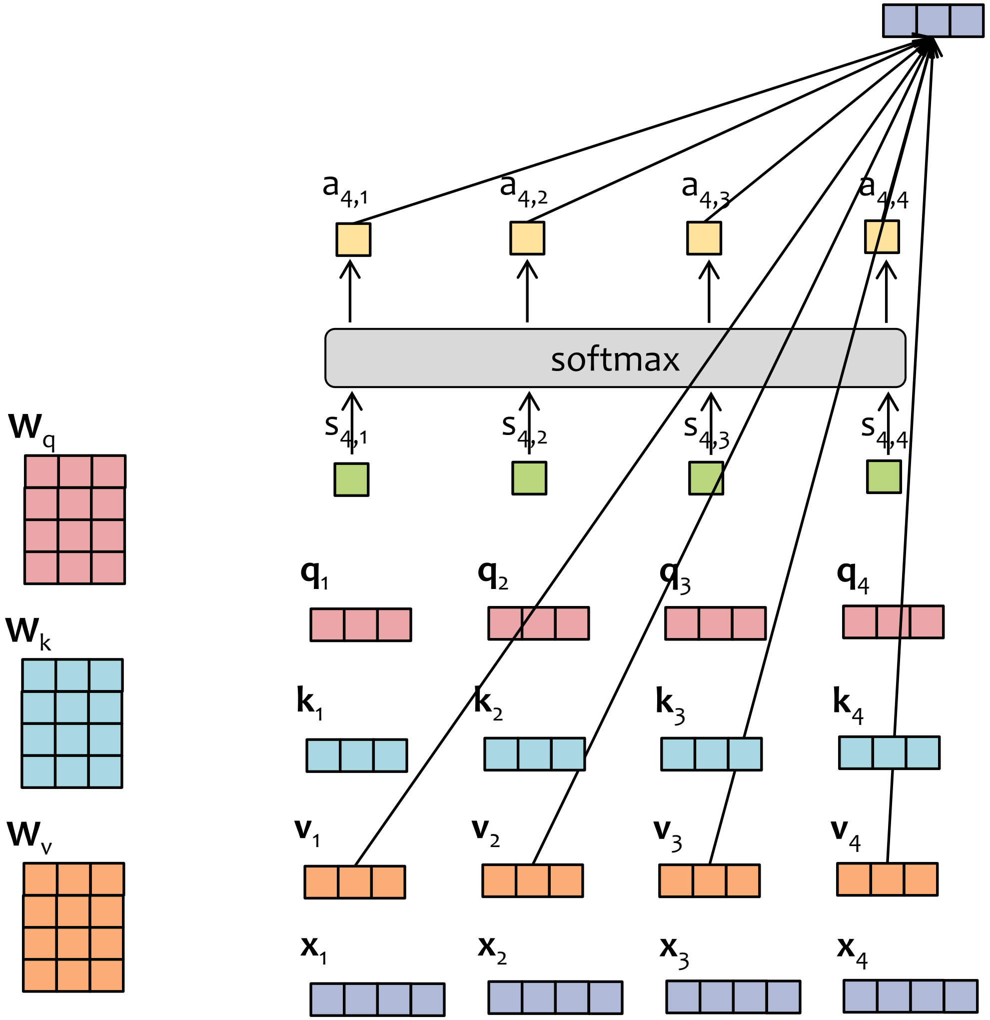


$$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j$$ keys

$$\mathbf{v}_j = \mathbf{W}_v^T \mathbf{x}_j$$ values

Scaled Dot-Product Attention

$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$$

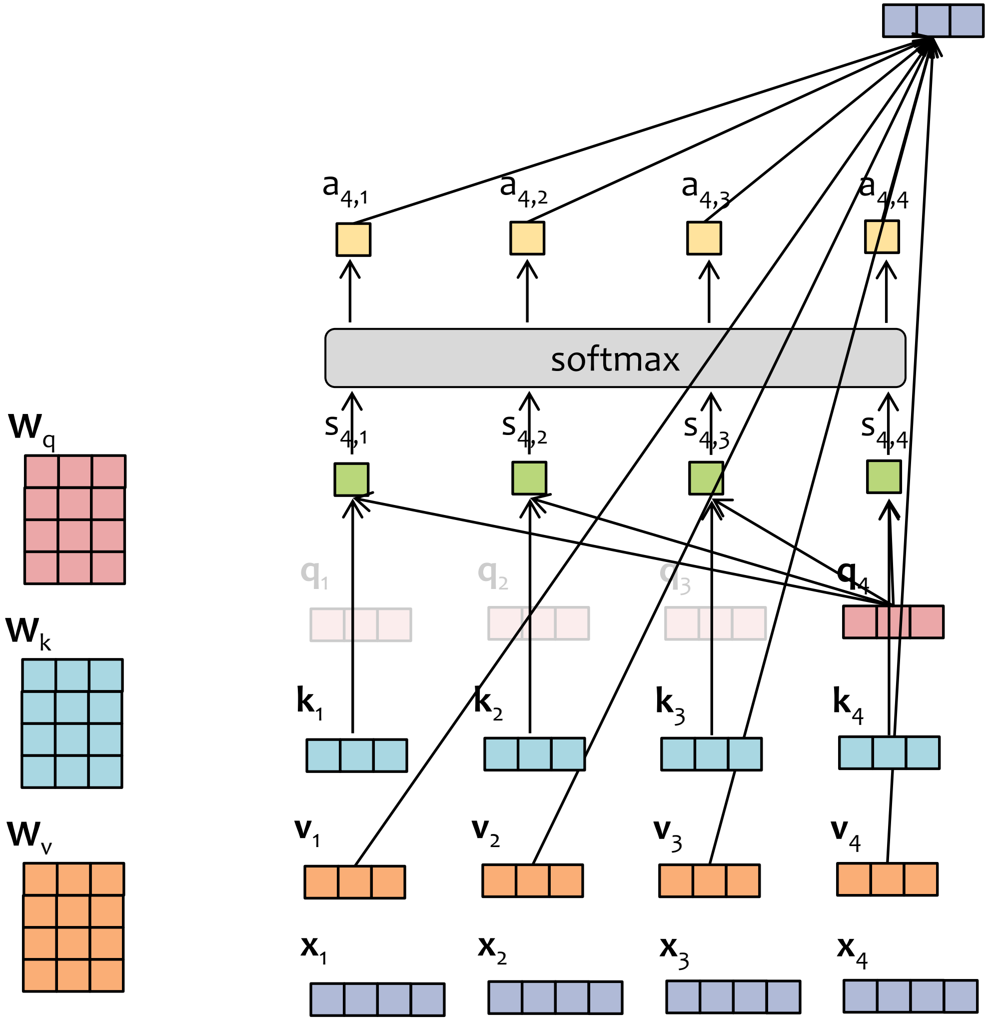


$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j$ queries

$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j$ keys

$\mathbf{v}_j = \mathbf{W}_v^T \mathbf{x}_j$ values

Scaled Dot-Product Attention



$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j} \mathbf{v}_j$$

$s_{4,j} = \mathbf{k}_j^T \mathbf{q}_4 / \sqrt{d_k}$ scores

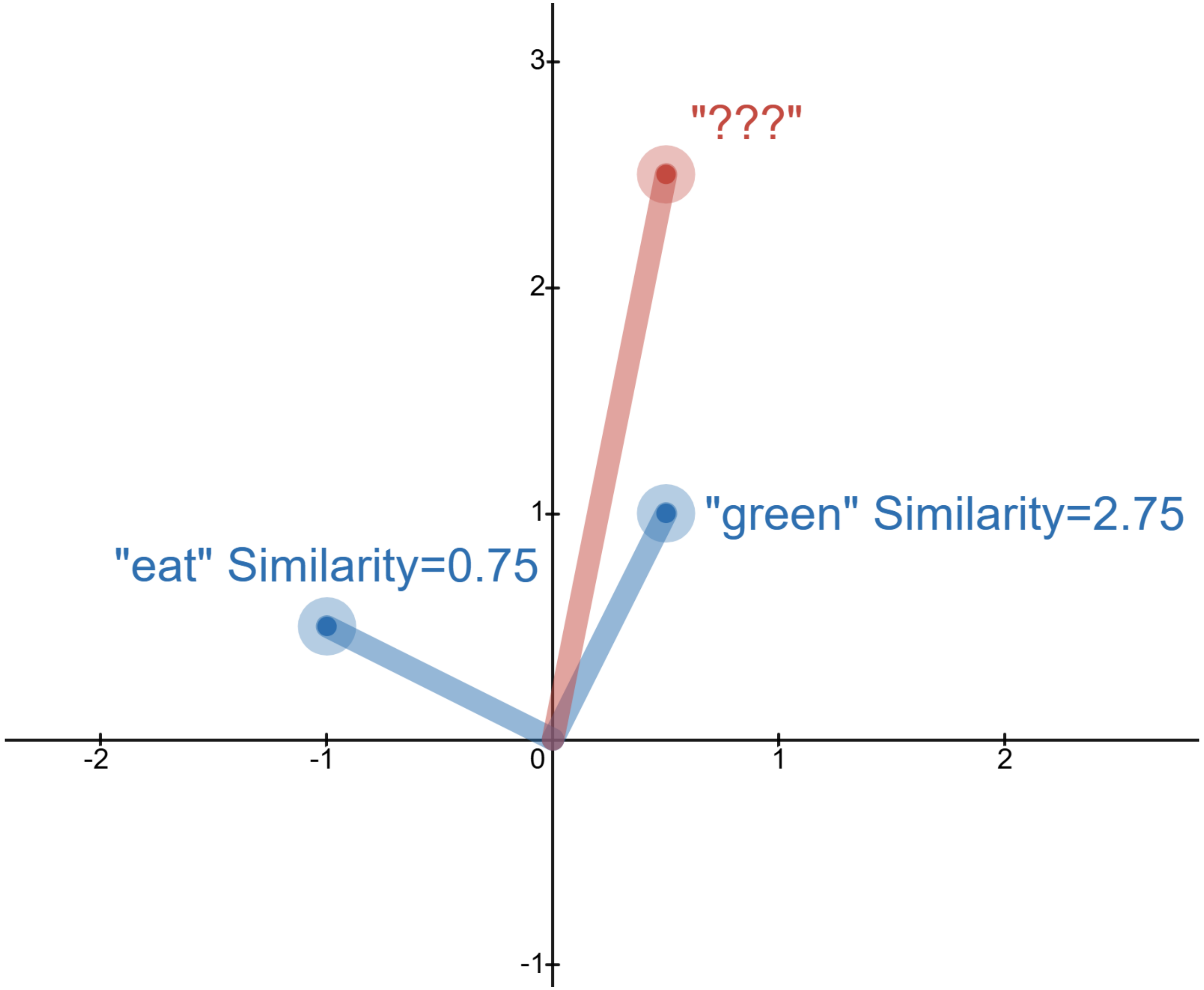
$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j$ queries

$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j$ keys

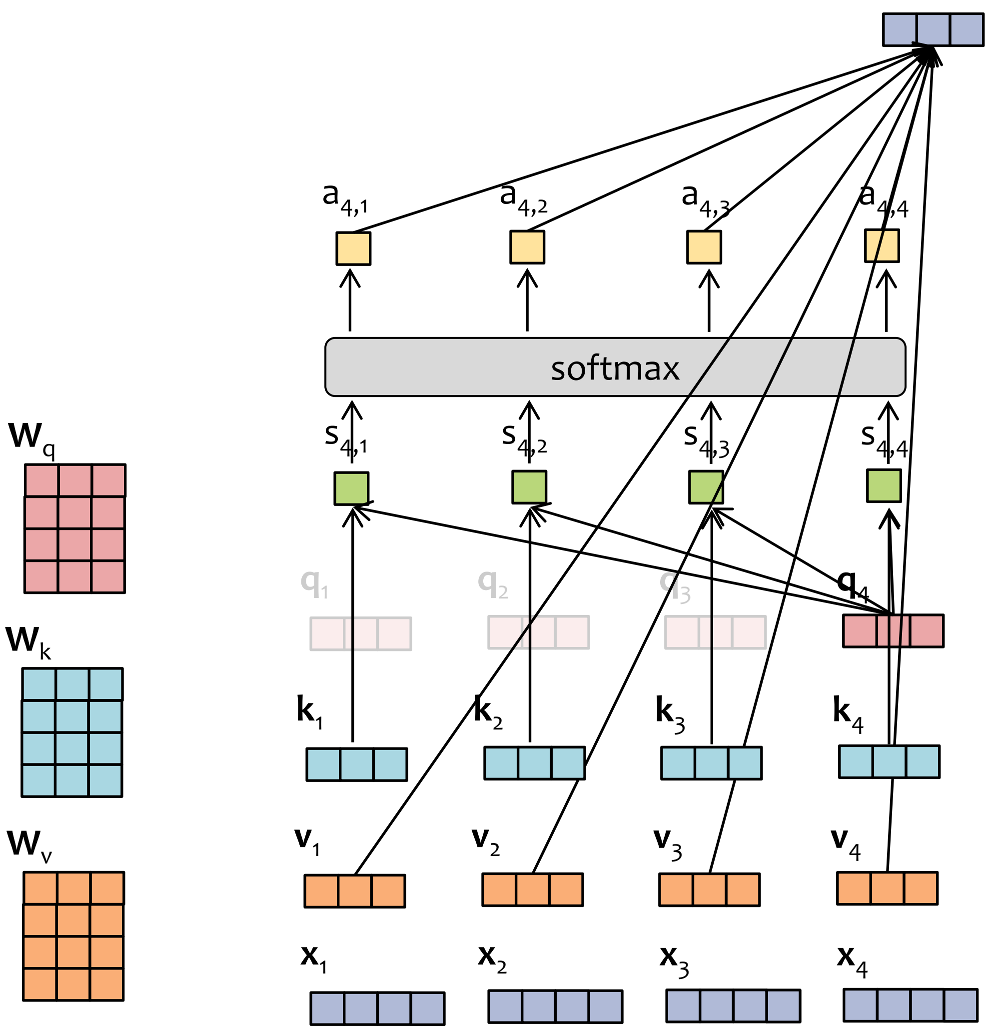
$\mathbf{v}_j = \mathbf{W}_v^T \mathbf{x}_j$ values

Cosine (dot product) Similarity: Graphical Intuition

- The dot product can function as a similarity measure of two vectors
- $similarity(\mathbf{u}, \mathbf{v}) = \mathbf{u}^{T}\mathbf{v}$, where higher values mean more similar
- Desmos example $\mathbb{R}^2$ https://www.desmos.com/calculator/3jxjjafyl1



Scaled Dot-Product Attention



$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$$

$\mathbf{a}_4 = \text{softmax}(\mathbf{s}_4)$ attention weights

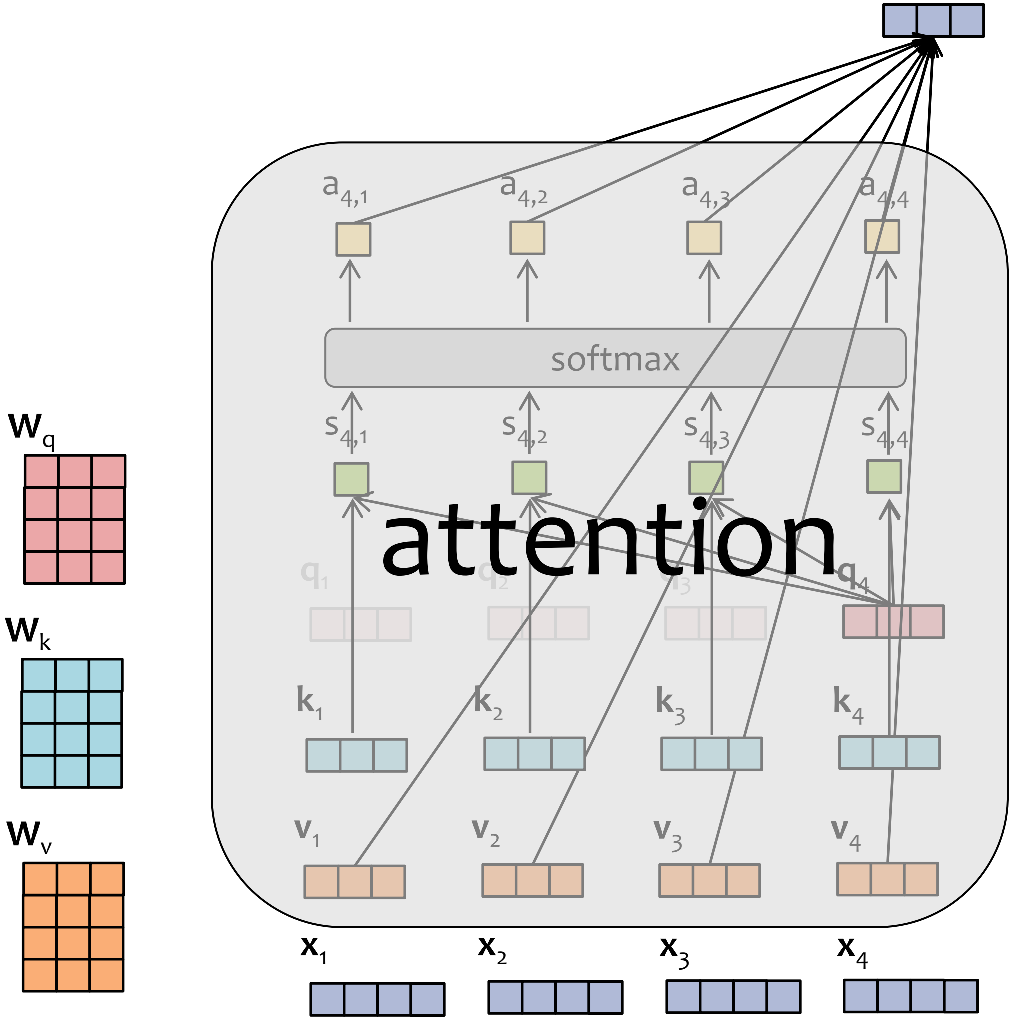
$s_{4,j} = \mathbf{k}_j^T \mathbf{q}_4 / \sqrt{d_k}$ scores

$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j$ queries

$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j$ keys

$\mathbf{v}_j = \mathbf{W}_v^T \mathbf{x}_j$ values

Scaled Dot-Product Attention



$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$$

$\mathbf{a}_4 = \text{softmax}(\mathbf{s}_4)$ attention weights

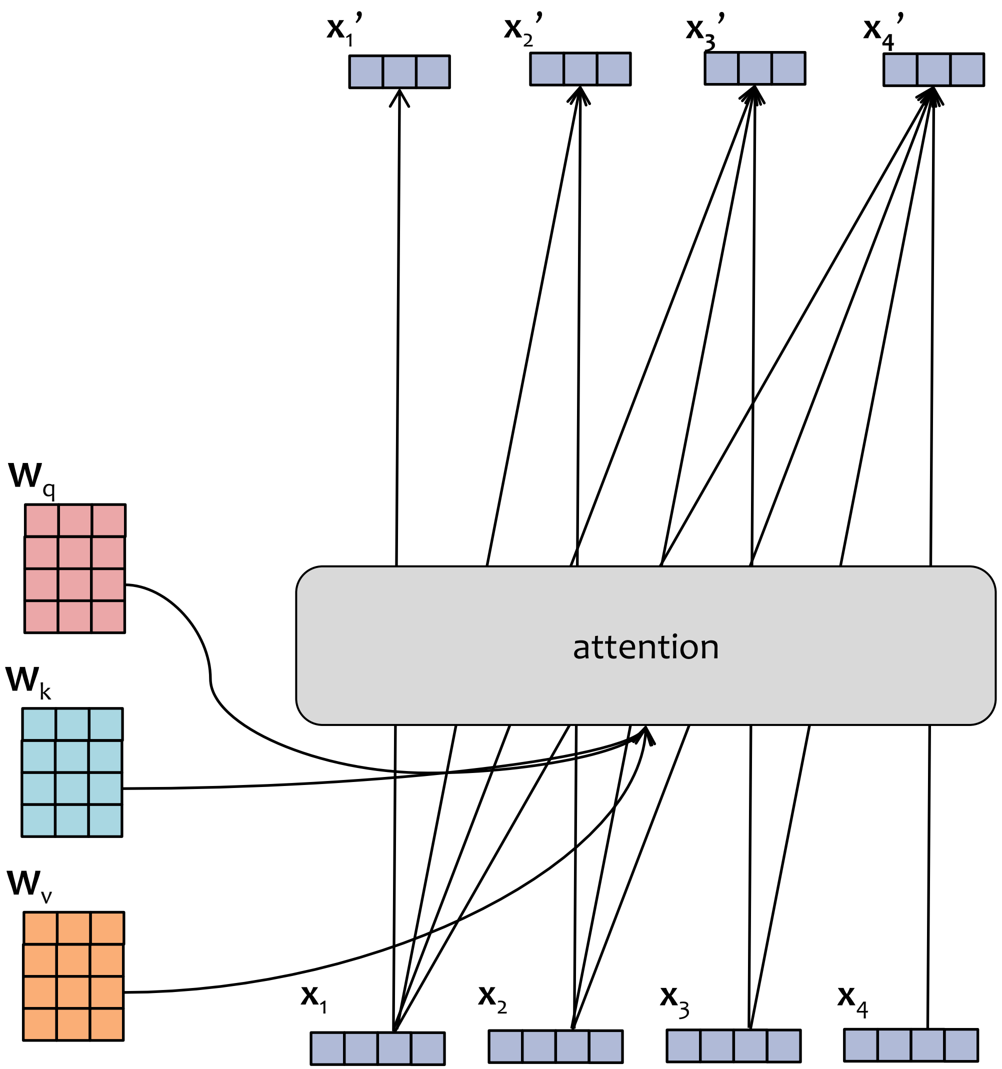
$s_{4,j} = \mathbf{k}_j^T \mathbf{q}_4 / \sqrt{d_k}$ scores

$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j$ queries

$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j$ keys

$\mathbf{v}_j = \mathbf{W}_v^T \mathbf{x}_j$ values

Scaled Dot-Product Attention



$$\mathbf{x}'_t = \sum_{j=1}^{t} a_{t,j}\mathbf{v}_j$$

$\mathbf{a}_t = \text{softmax}(\mathbf{s}_t)$ attention weights

$s_{t,j} = \mathbf{k}_j^T \mathbf{q}_t / \sqrt{d_k}$ scores

$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j$ queries

$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j$ keys

$\mathbf{v}_j = \mathbf{W}_v^T \mathbf{x}_j$ values

Animation of 3D Convolution

Recall…

http://cs231n.github.io/convolutional-networks/

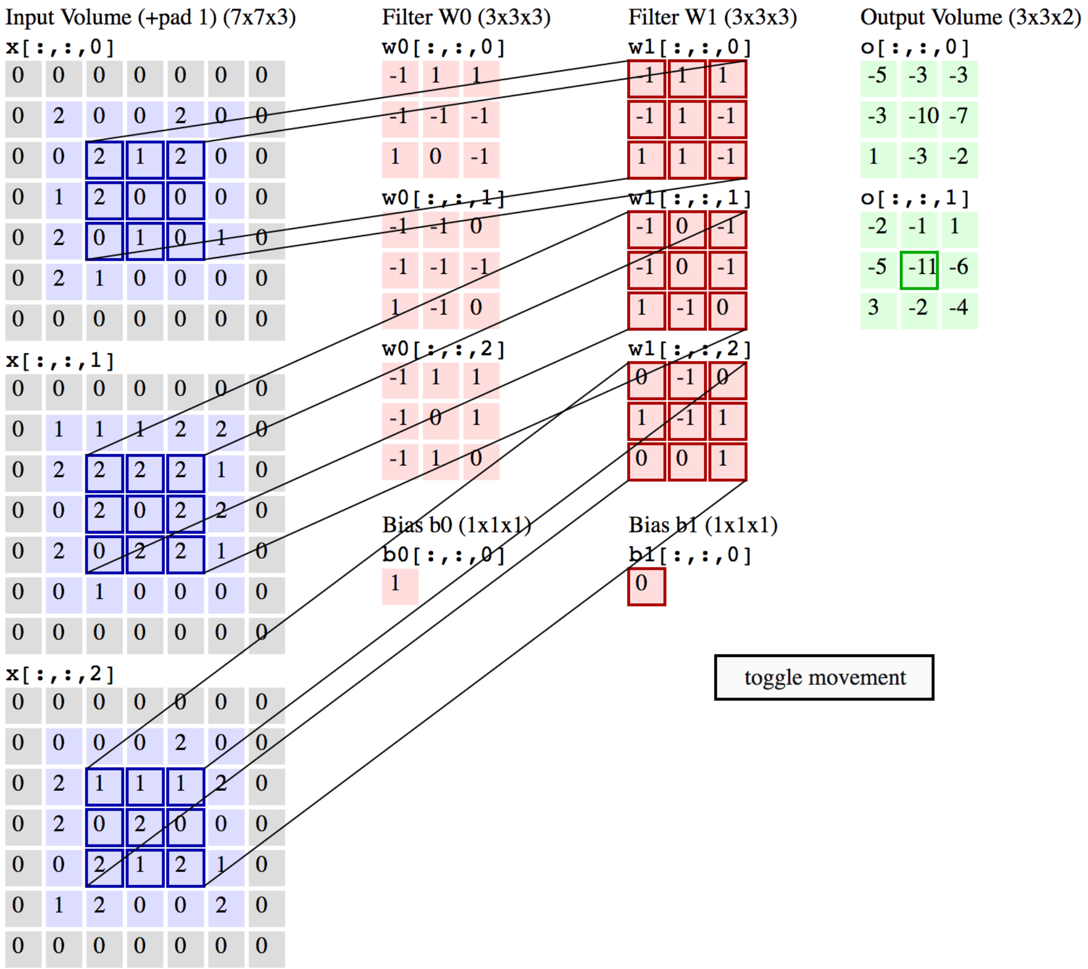
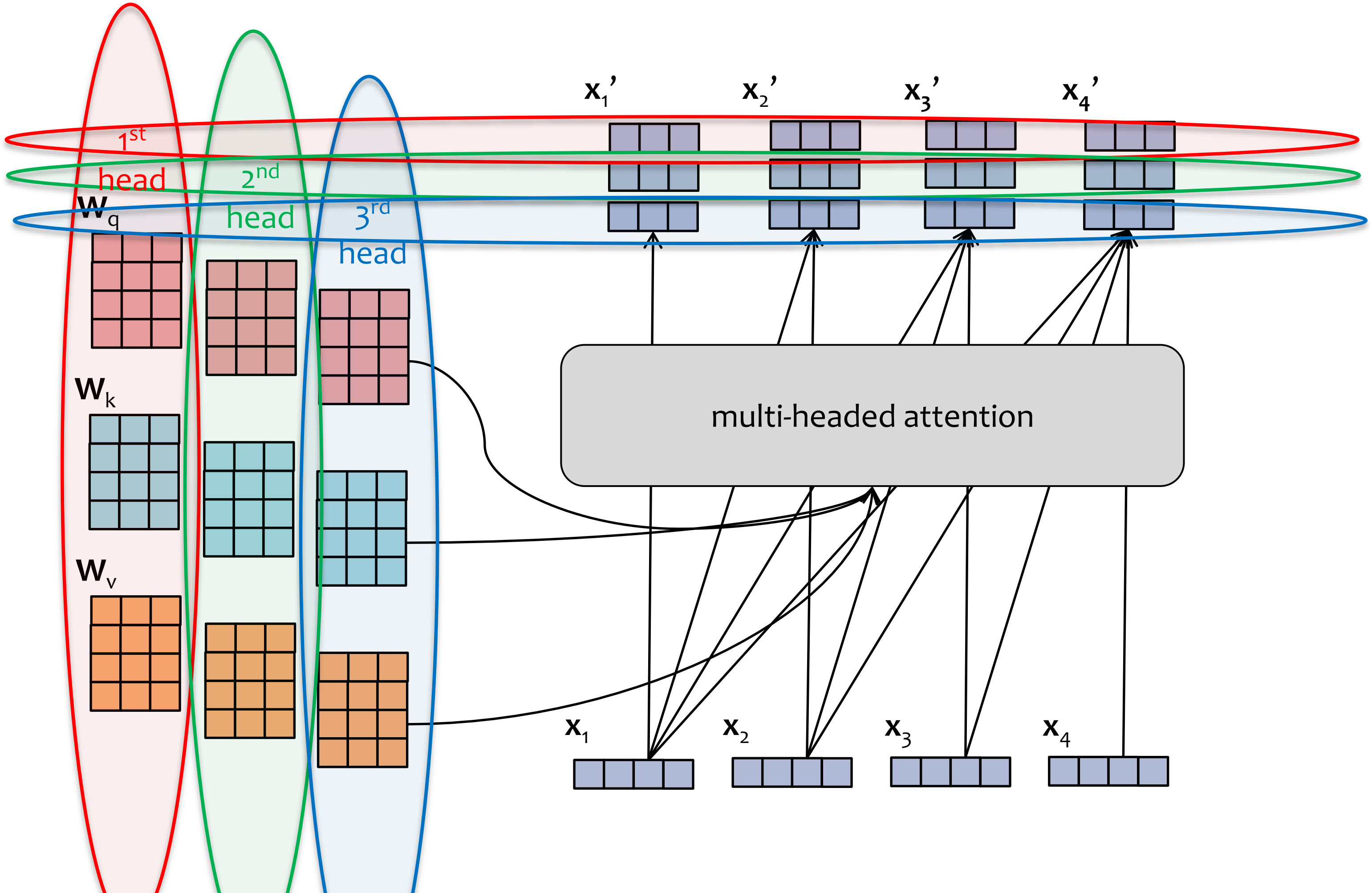


Figure from Fei-Fei Li & Andrej Karpathy & Justin Johnson (CS231N)

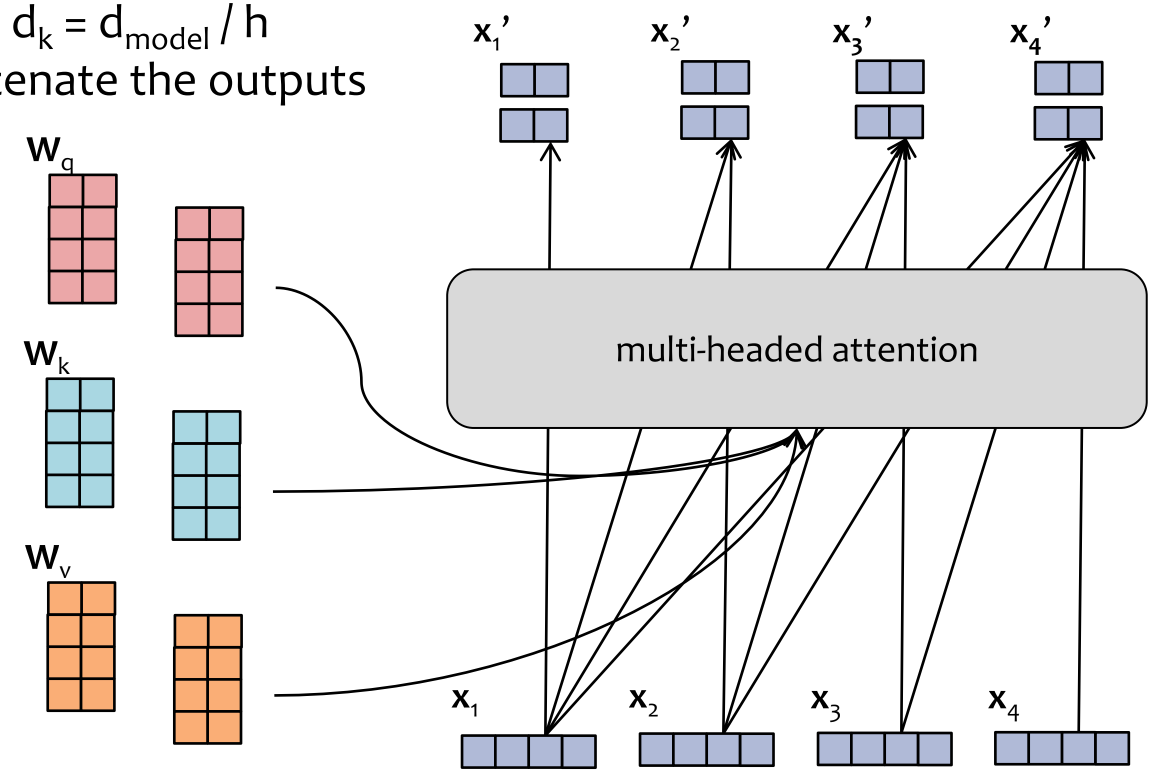
Multi-headed Attention



- Just as we can have **multiple channels** in a **convolution** layer, we can use **multiple heads** in an **attention** layer
- Each head gets **its own parameters**
- We can **concatenate** all the outputs to get a single vector for each time step

Multi-headed Attention

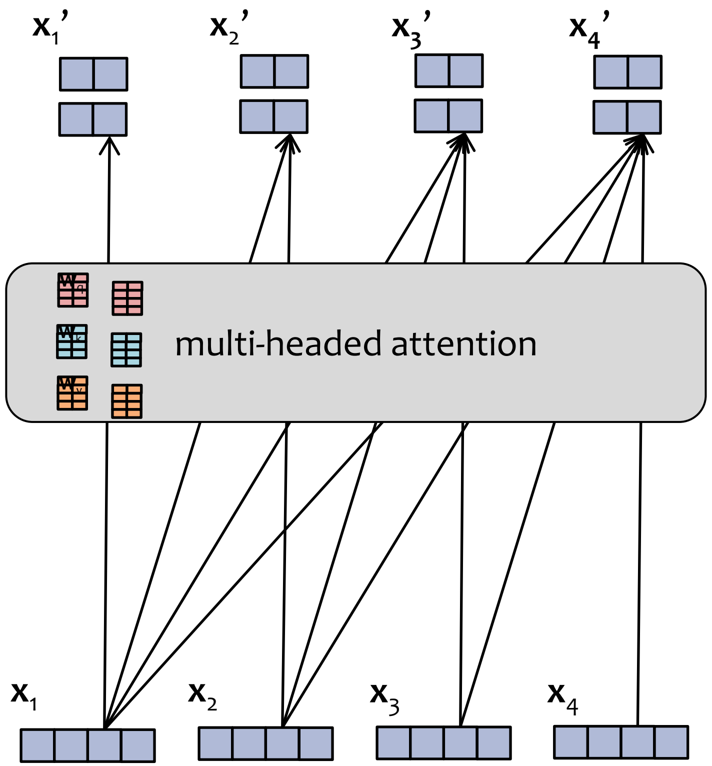
- To ensure the dimension of the **input** embedding $\mathbf{x}_t$ is the same as the **output** embedding $\mathbf{x}_t'$, Transformers usually choose the embedding sizes and number of heads appropriately:
  - d_{model} = dim. of inputs
  - d_k = dim. of each output
  - h = # of heads
  - Choose $d_k = d_{model} / h$
- Then concatenate the outputs



- Just as we can have **multiple channels** in a **convolution** layer, we can use **multiple heads** in an **attention** layer
- Each head gets **its own parameters**
- We can **concatenate** all the outputs to get a single vector for each time step

Multi-headed Attention

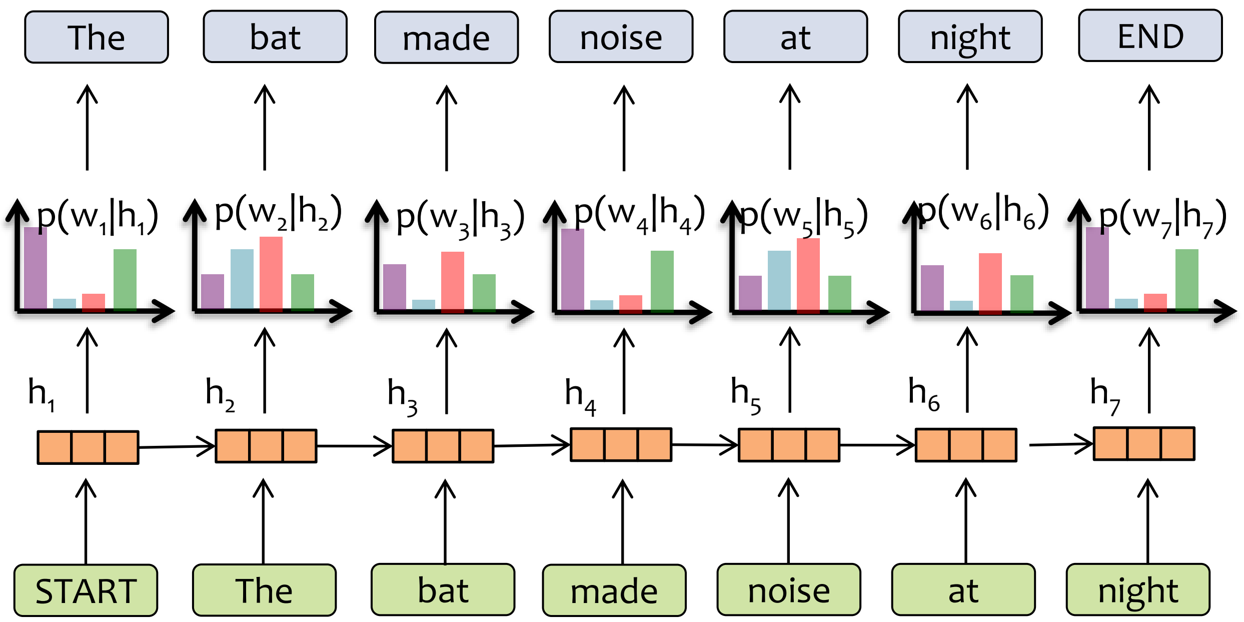
- To ensure the dimension of the **input** embedding $\mathbf{x}_t$ is the same as the **output** embedding $\mathbf{x}_t'$, Transformers usually choose the embedding sizes and number of heads appropriately:
  - d_{model} = dim. of inputs
  - d_k = dim. of each output
  - h = # of heads
  - Choose $d_k = d_{model}$ / h
- Then concatenate the outputs



- Just as we can have **multiple channels** in a **convolution** layer, we can use **multiple heads** in an **attention** layer
- Each head gets **its own parameters**
- We can **concatenate** all the outputs to get a single vector for each time step

RNN Language Model

Recall...



<u>Key Idea</u>:

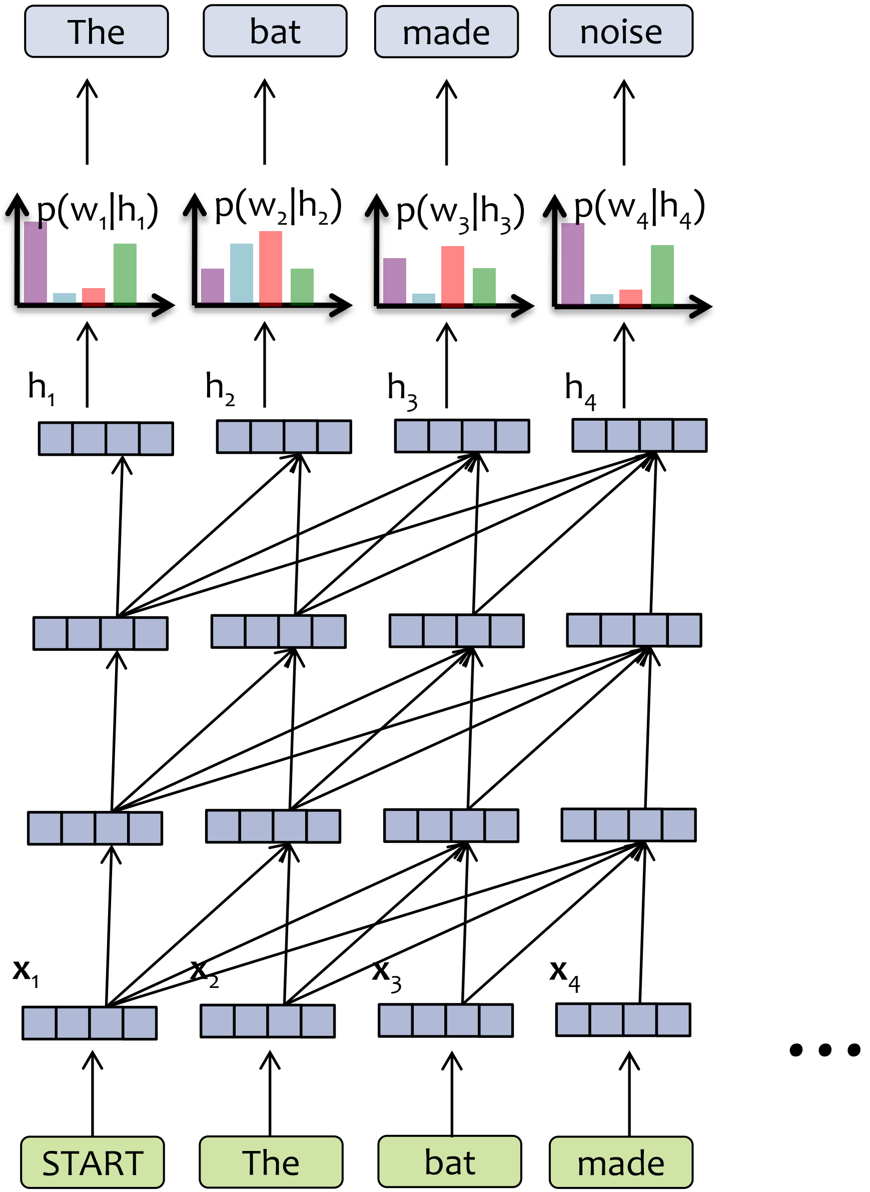
(1) convert all previous words to a **fixed length vector**

(2) define distribution $p(w_t \mid f_\theta(w_{t-1}, \ldots, w_1))$ that conditions on the vector $\mathbf{h}_t = f_\theta(w_{t-1}, \ldots, w_1)$

Transformer Language Model

Important!

- RNN computation graph grows **linearly** with the number of input tokens
- Transformer-LM computation graph grows **quadratically** with the number of input tokens



Each layer of a Transformer LM consists of several **sublayers:**

1. attention
2. feed-forward neural network
3. layer normalization
4. residual connections

Each hidden vector looks back at the hidden vectors of the **current and previous timesteps in the previous layer.**

The language model part is just like an RNN-LM!

Layer Normalization

- *The Problem*: **internal covariate shift** occurs during training of a deep network when a small change in the low layers amplifies into a large change in the high layers
- *One Solution:* **Layer normalization** normalizes each layer and learns elementwise gain/bias
- Such normalization allows for higher learning rates (for **faster convergence**) without issues of diverging gradients

Given input $\mathbf{a} \in \mathbb{R}^K$, LayerNorm computes output $\mathbf{b} \in \mathbb{R}^K$:

$$\mathbf{b} = \boldsymbol{\gamma} \odot \frac{\mathbf{a} - \mu}{\sigma} \oplus \boldsymbol{\beta}$$

where we have mean $\mu = \frac{1}{K} \sum_{k=1}^{K} a_k$,
standard deviation $\sigma = \sqrt{\frac{1}{K} \sum_{k=1}^{K} (a_k - \mu)^2}$,
and parameters $\boldsymbol{\gamma} \in \mathbb{R}^K$, $\boldsymbol{\beta} \in \mathbb{R}^K$.
$\odot$ and $\oplus$ denote elementwise multiplication and addition.

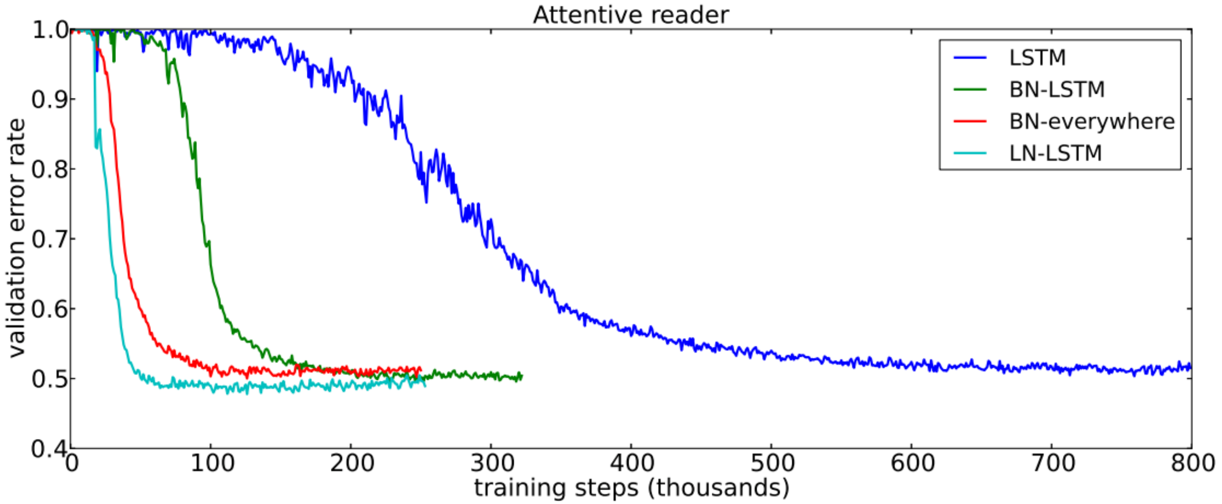
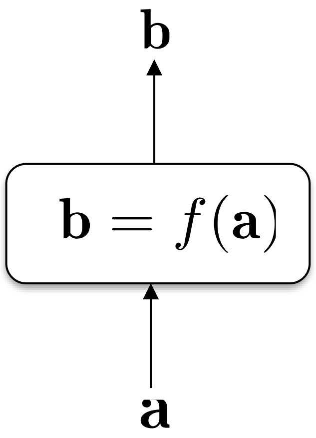


Figure from https://arxiv.org/pdf/1607.06450.pdf

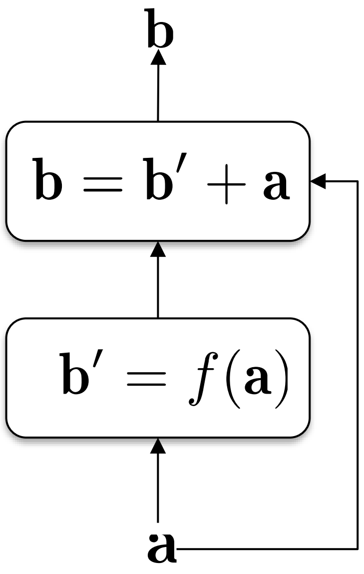
Residual Connections

- *The Problem:* as network depth grows very large, a **performance degradation** occurs that is not explained by overfitting (i.e. train / test error both worsen)
- *One Solution:* **Residual connections** pass a copy of the input alongside another function so that information can flow more directly
- These residual connections allow for **effective training of very deep networks** that perform better than their shallower (though still deep) counterparts

Plain Connection



Residual Connection



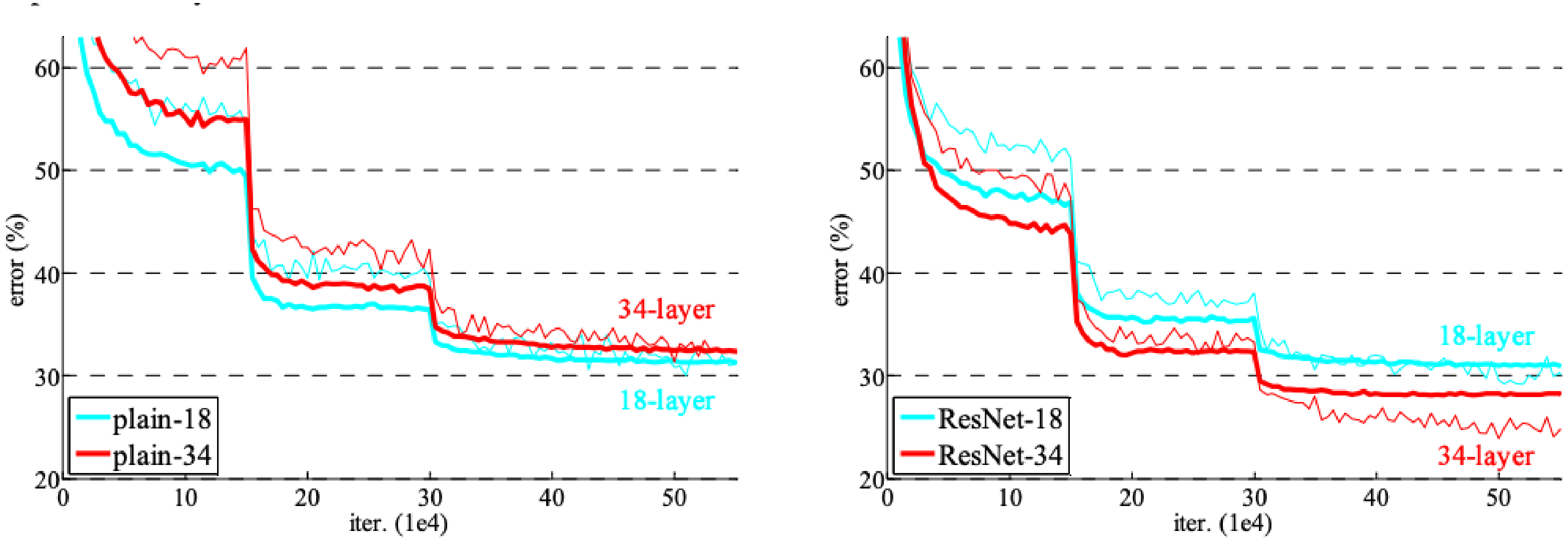
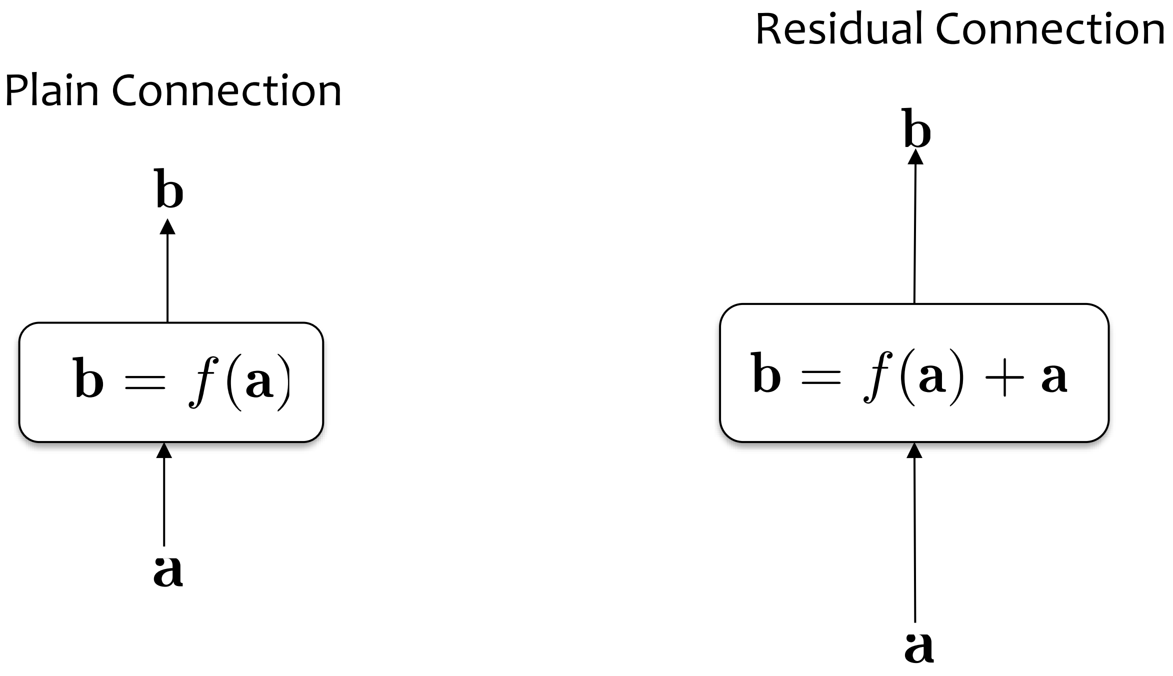


Figure from https://arxiv.org/pdf/1512.03385.pdf

Residual Connections

- *The Problem:* as network depth grows very large, a **performance degradation** occurs that is not explained by overfitting (i.e. train / test error both worsen)
- *One Solution:* **Residual connections** pass a copy of the input alongside another function so that information can flow more directly
- These residual connections allow for **effective training of very deep networks** that perform better than their shallower (though still deep) counterparts

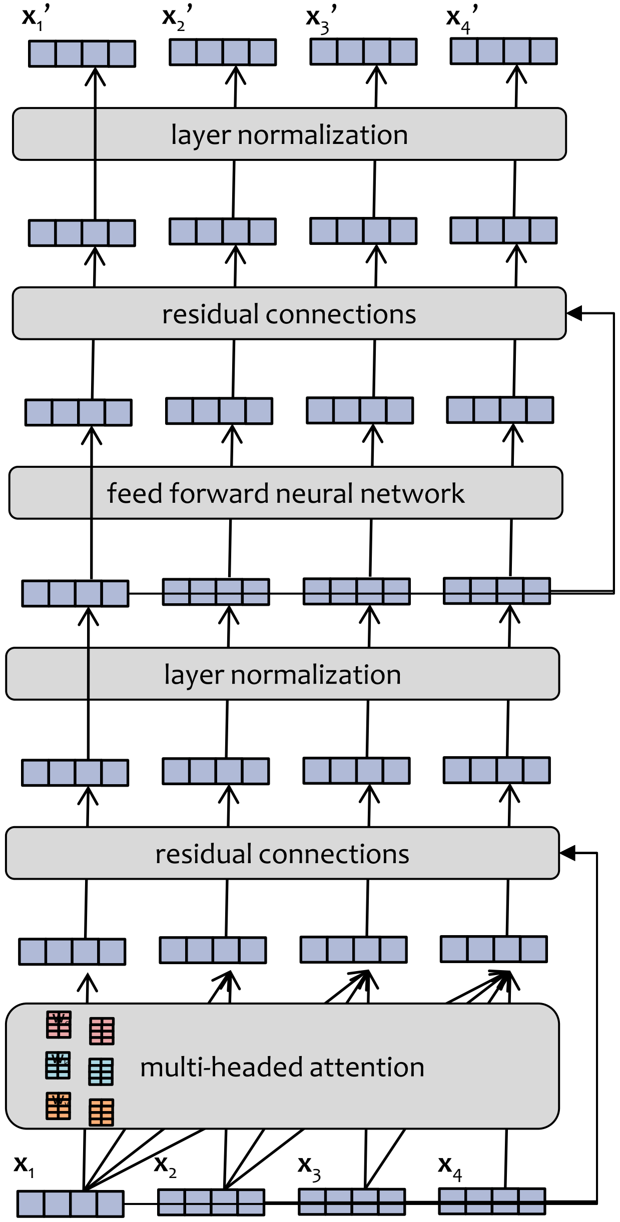


Why are residual connections helpful?

Instead of f(a) having to learn a full transformation of a, f(a) only needs to learn an additive modification of a (i.e. the residual).

Figure from https://arxiv.org/pdf/1512.03385.pdf

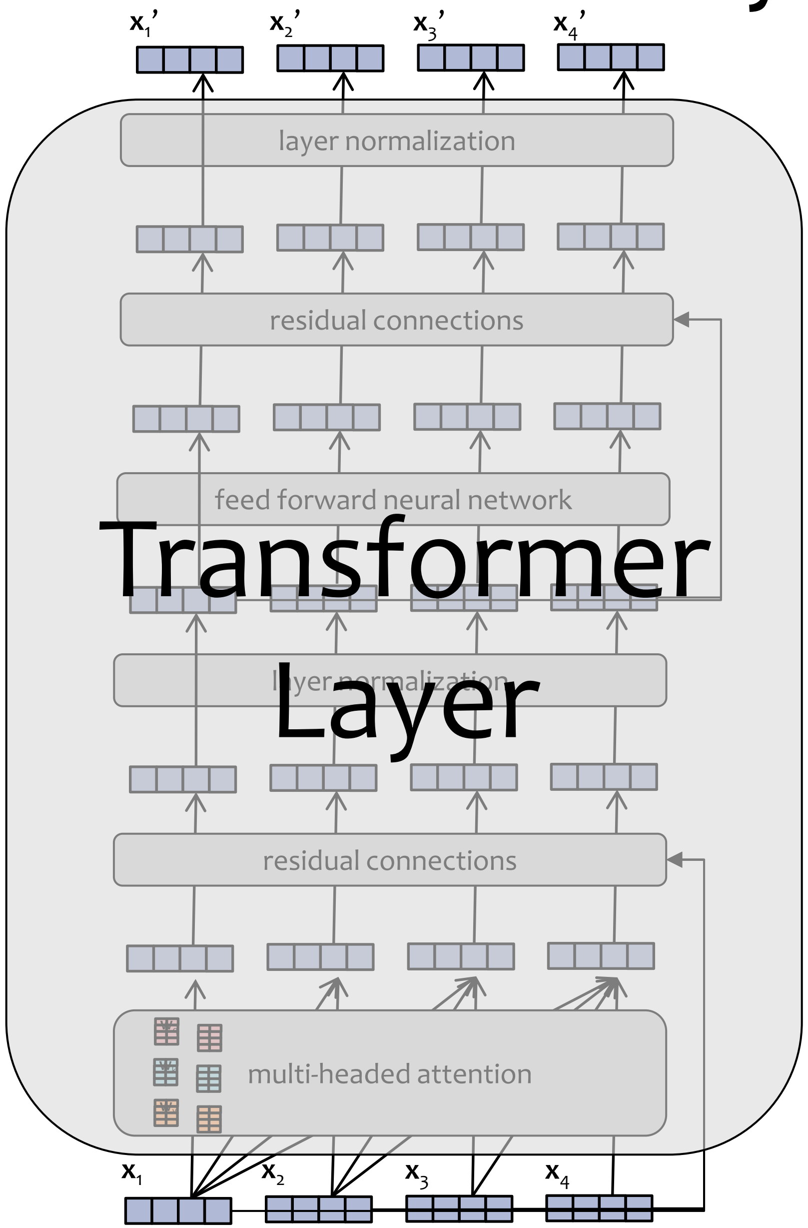
Transformer Layer



Each layer of a Transformer LM consists of several **sublayers:**

1. attention
2. feed-forward neural network
3. layer normalization
4. residual connections

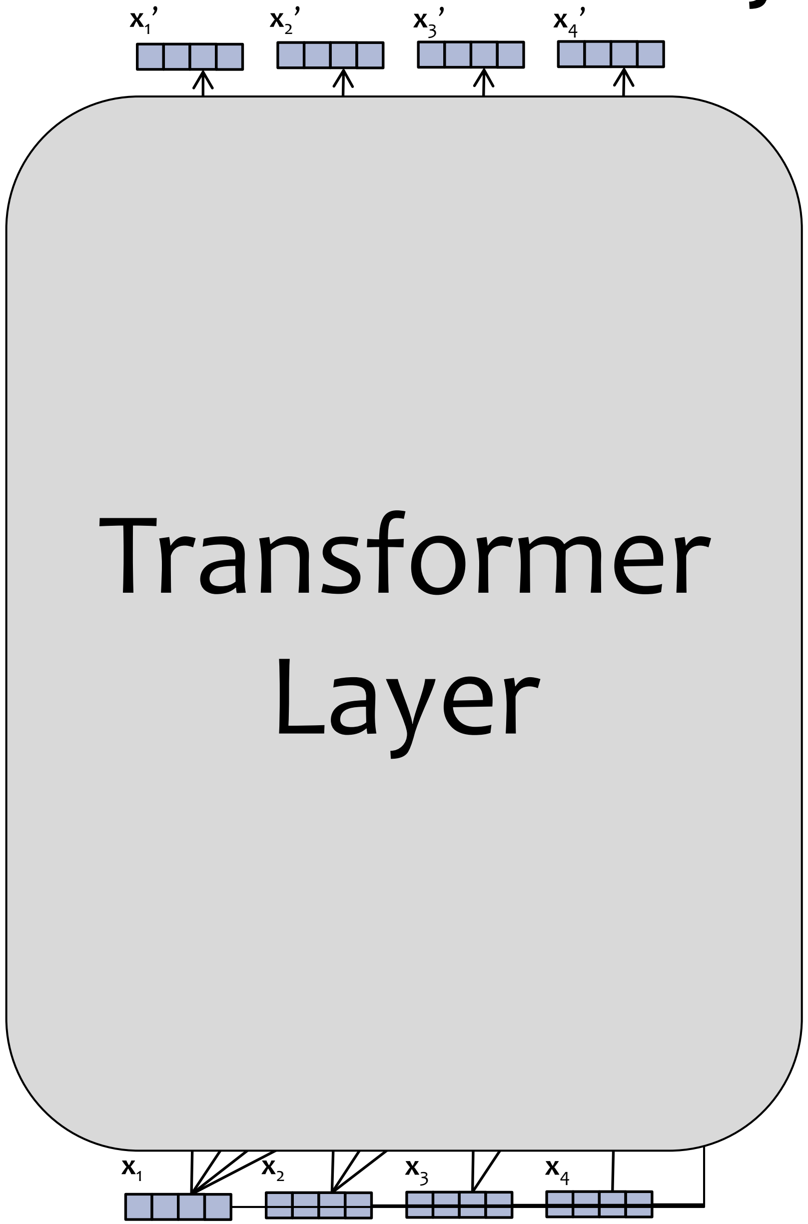
Transformer Layer



Each layer of a Transformer LM consists of several **sublayers:**

1. attention
2. feed-forward neural network
3. layer normalization
4. residual connections

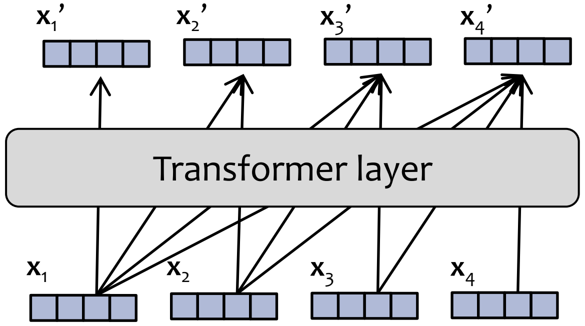
Transformer Layer



Each layer of a Transformer LM consists of several **sublayers:**

1. attention
2. feed-forward neural network
3. layer normalization
4. residual connections

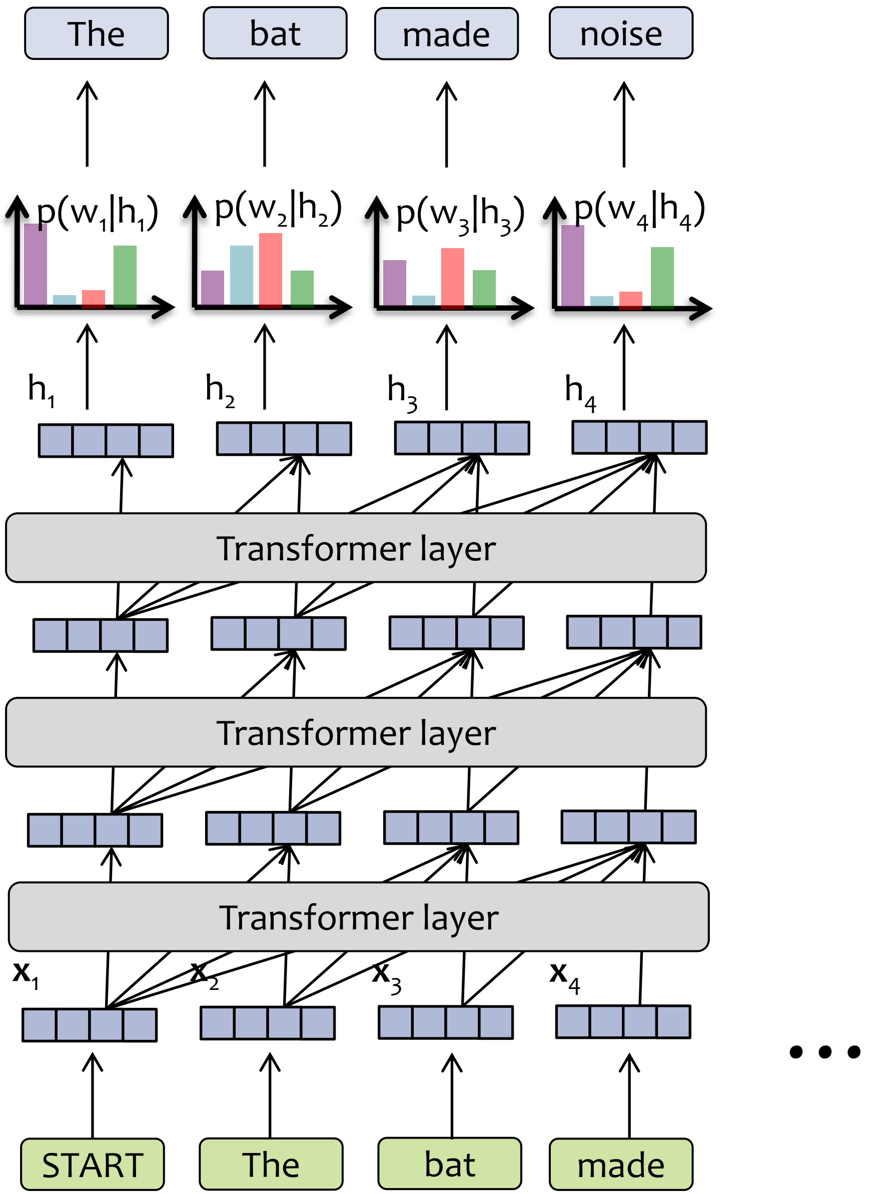
Transformer Layer



Each layer of a Transformer LM consists of several **sublayers:**

1. attention
2. feed-forward neural network
3. layer normalization
4. residual connections

Transformer Language Model



Each layer of a Transformer LM consists of several **sublayers:**

1. attention
2. feed-forward neural network
3. layer normalization
4. residual connections

Each hidden vector looks back at the hidden vectors of the **current and previous timesteps in the previous layer.**

The language model part is just like an RNN-LM!

In-Class Exercise

Question:

Suppose we have the following input embeddings and attention weights:

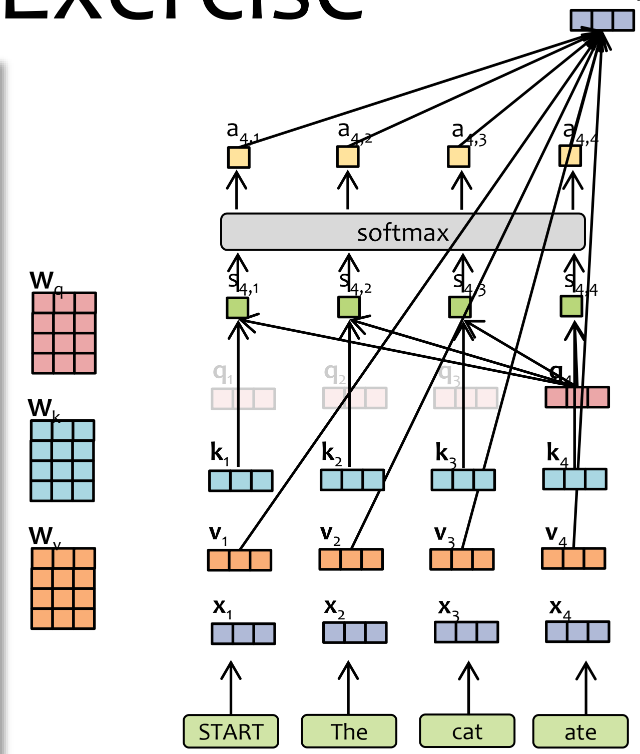
- $x_1 = [1,0,0]$ $a_{4,1} = 0.1$
- $x_2 = [0,1,0]$ $a_{4,2} = 0.2$
- $x_3 = [0,0,2]$ $a_{4,3} = 0.6$
- $x_4 = [0,0,1]$ $a_{4,4} = 0.1$

And $W_v = I$. Then we can compute x_4'.

Now suppose we swap the order of x_2 and x_3 embeddings such that

- $x_2 = [0,0,2]$ $a_{4,2} =$
- $x_3 = [0,1,0]$ $a_{4,3} =$

What is the new value of x_4'?



$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$$

$\mathbf{a}_4 = \text{softmax}(\mathbf{s}_4)$ attention weights

$s_{4,j} = \mathbf{k}_j^T \mathbf{q}_4 / \sqrt{d_k}$ scores

$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j$ queries

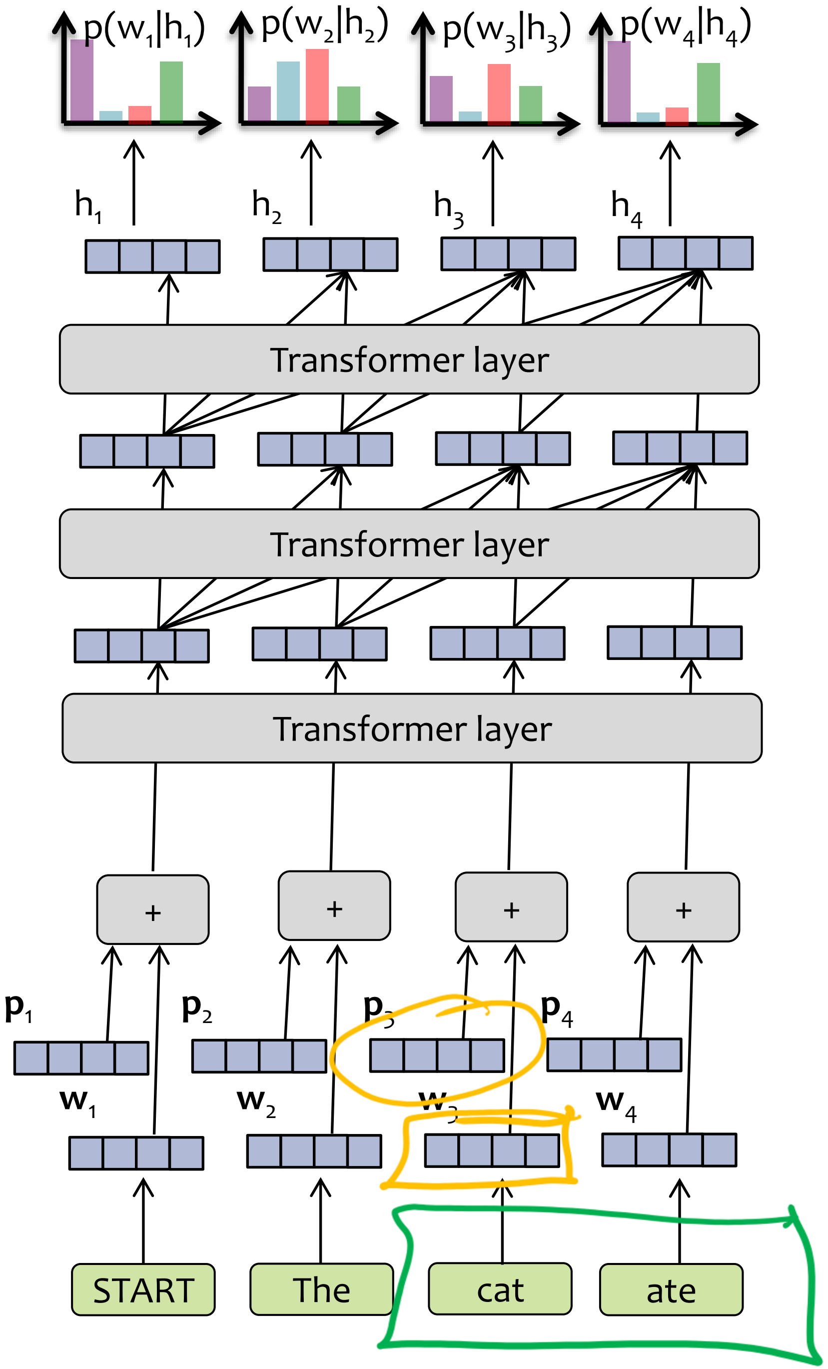
$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j$ keys

$\mathbf{v}_j = \mathbf{W}_v^T \mathbf{x}_j$ values

Answer:

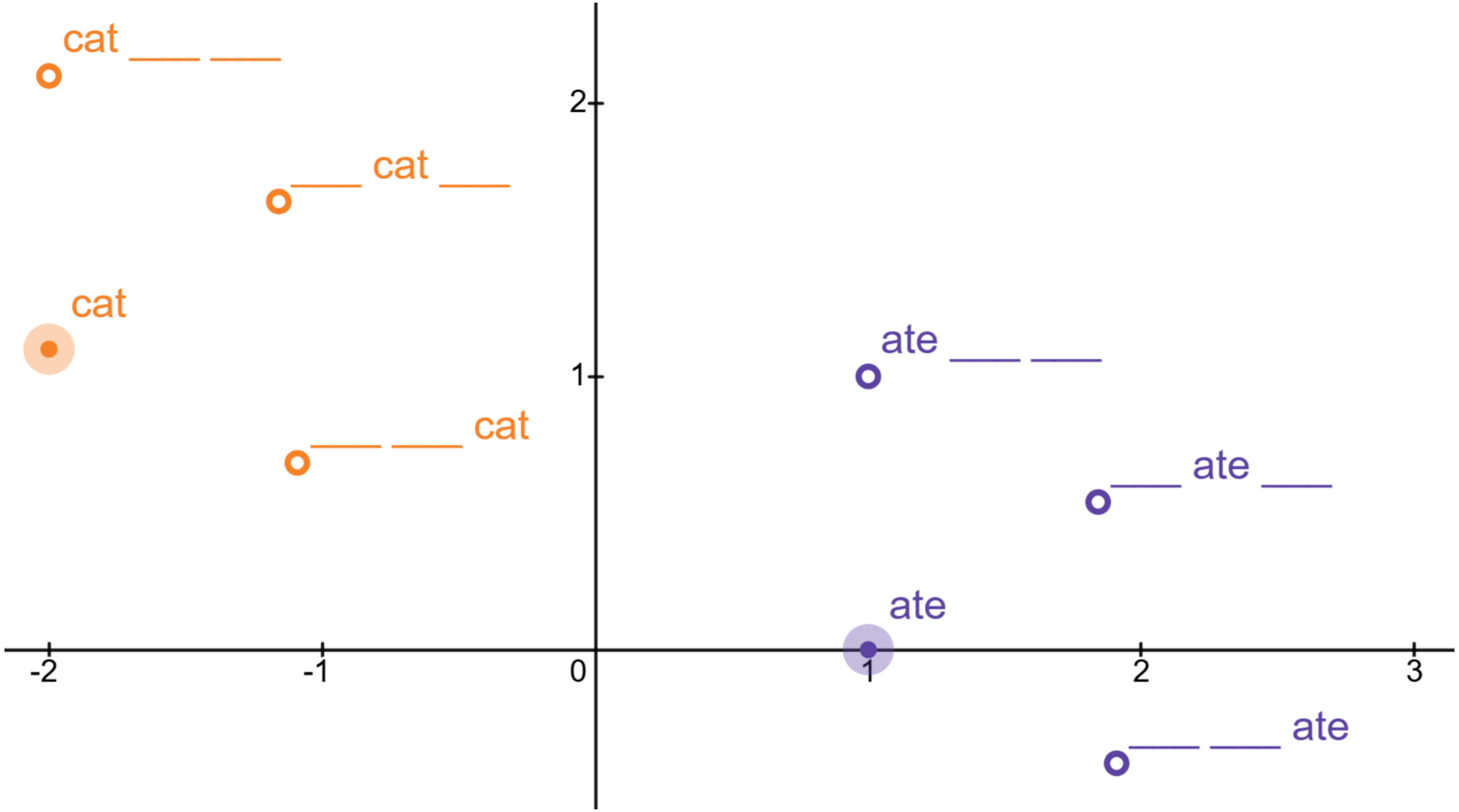
Position Embeddings

- The Problem: Because attention is position invariant, we **need** a way to learn about positions
- The Solution: Use (or learn) a collection of position specific embeddings: $\mathbf{p}_t$ represents what it means to be in position t. And add this to the word embedding $\mathbf{w}_t$.
  The **key idea** is that every word that appears in position t uses the same position embedding $\mathbf{p}_t$
- There are a number of varieties of position embeddings:
  – Some are fixed (based on sine and cosine), whereas others are learned (like word embeddings)
  – Some are absolute (as described above) but we can also use relative position embeddings (i.e. relative to the position of the query vector)



Position Embedding: Graphical Intuition

- Add a vector to each word embedding depending on it's position in the sequence
- Desmos example in $\mathbb{R}^2$ https://www.desmos.com/calculator/z48zbc6tco



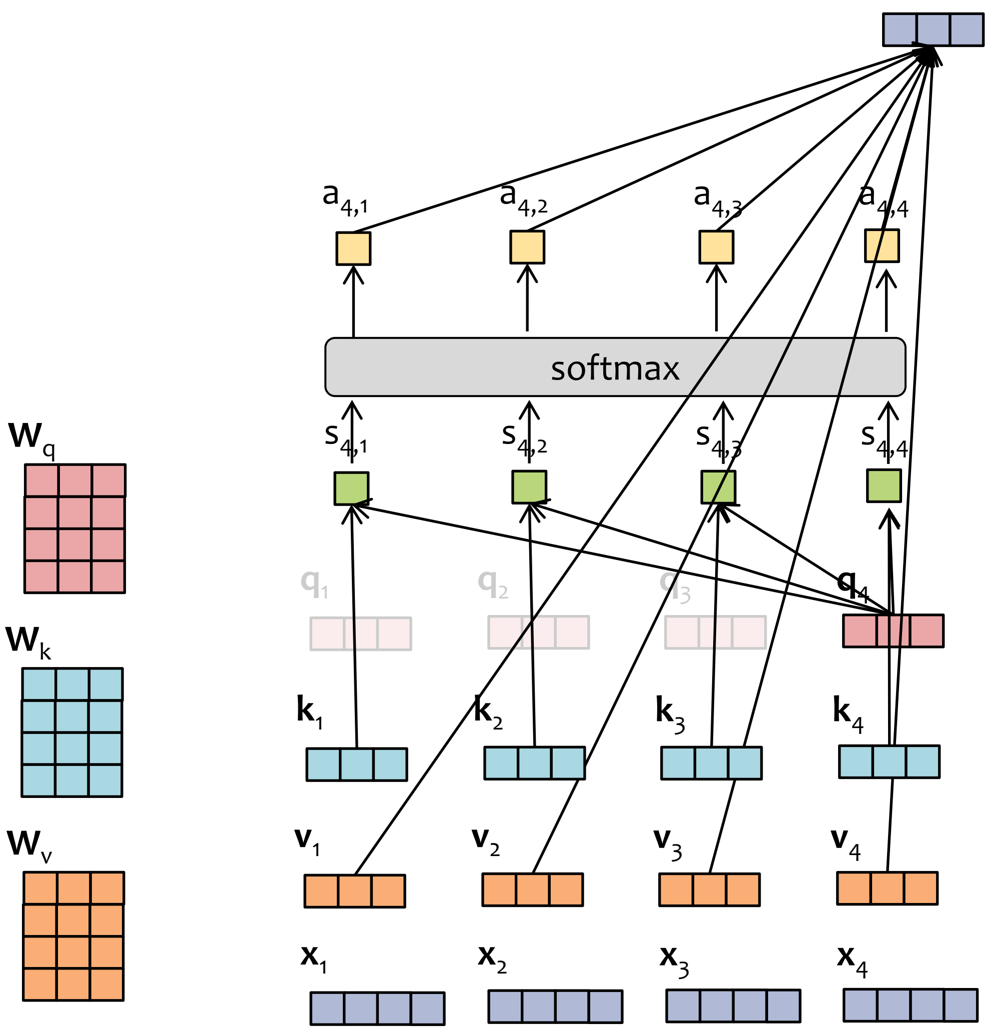
GPT-3

- GPT stands for Generative Pre-trained Transformer
- GPT is just a Transformer LM, but with a huge number of parameters

Model	# layers	dimension of states	dimension of inner states	# attention heads	# params
GPT (2018)	12	768	4*768	12	117M
GPT-2 (2019)	48	1600	4*1600	12	1542M
GPT-3 (2020)	96	12288	4*12288	96	175000M

IMPLEMENTING A TRANSFORMER LM

Matrix Version of Single-Headed Attention



$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j} \mathbf{v}_j$$

$\mathbf{a}_4 = \text{softmax}(\mathbf{s}_4)$ attention weights

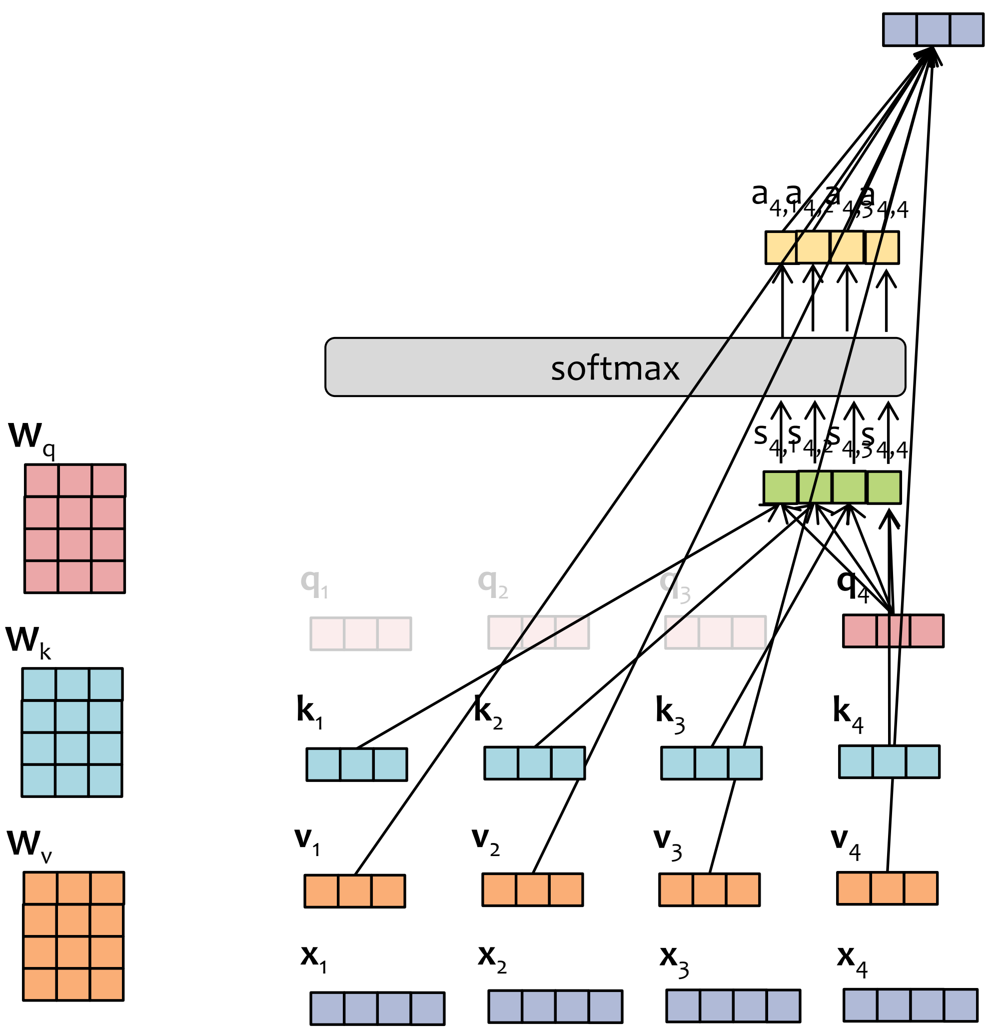
$s_{4,j} = \mathbf{k}_j^T \mathbf{q}_4 / \sqrt{d_k}$ scores

$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j$ queries

$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j$ keys

$\mathbf{v}_j = \mathbf{W}_v^T \mathbf{x}_j$ values

Matrix Version of Single-Headed Attention



$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j} \mathbf{v}_j$$

$\mathbf{a}_4 = \text{softmax}(\mathbf{s}_4)$ attention weights

$s_{4,j} = \mathbf{k}_j^T \mathbf{q}_4 / \sqrt{d_k}$ scores

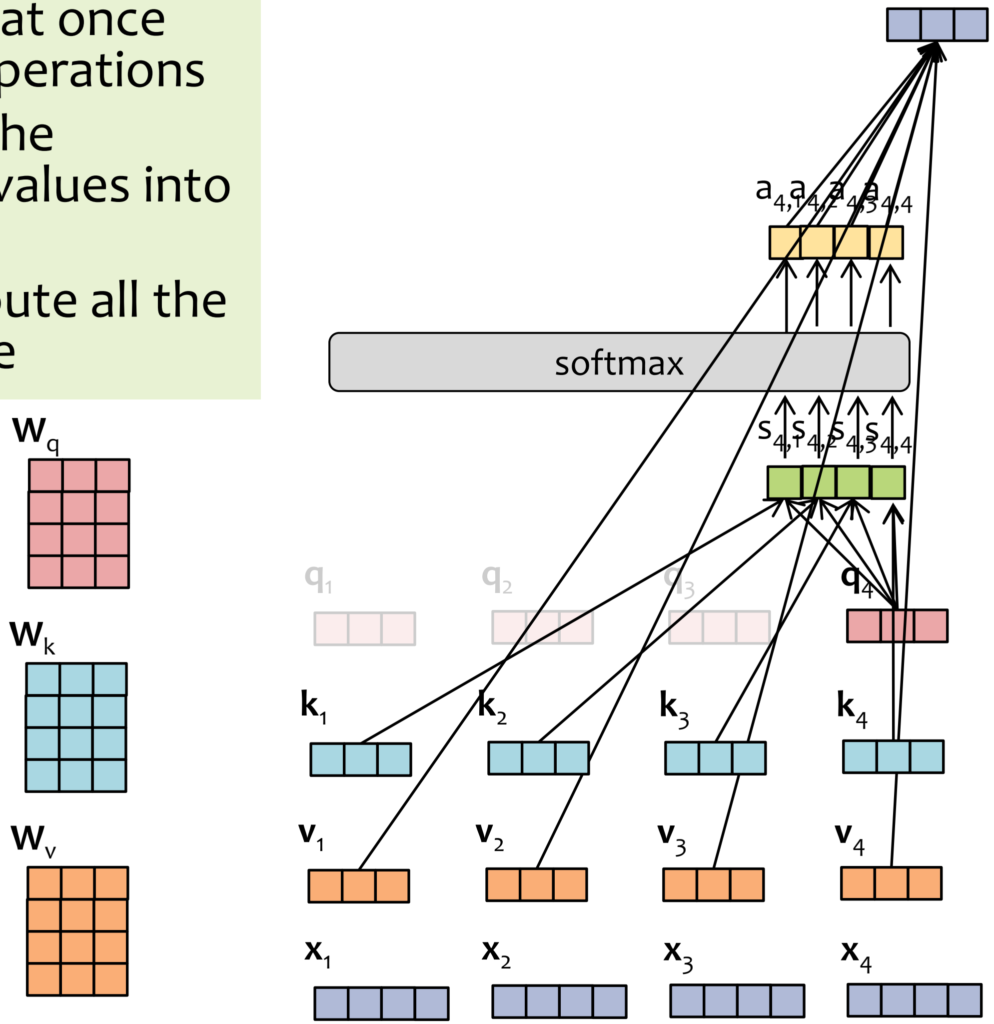
$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j$ queries

$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j$ keys

$\mathbf{v}_j = \mathbf{W}_v^T \mathbf{x}_j$ values

Matrix Version of Single-Headed Attention

- For speed, we compute all the queries at once using matrix operations
- First we pack the queries, keys, values into matrices
- Then we compute all the queries at once



$$\mathbf{X}' = \mathbf{A}\mathbf{V} = \text{softmax}(\mathbf{Q}\mathbf{K}^T/\sqrt{d_k})\mathbf{V}$$

$$\mathbf{A} = [\mathbf{a}_1, \ldots, \mathbf{a}_4]^T = \text{softmax}(\mathbf{S})$$

$$\mathbf{S} = [\mathbf{s}_1, \ldots, \mathbf{s}_4]^T = \mathbf{Q}\mathbf{K}^T/\sqrt{d_k}$$

$$\mathbf{Q} = [\mathbf{q}_1, \ldots, \mathbf{q}_4]^T = \mathbf{X}\mathbf{W}_q$$

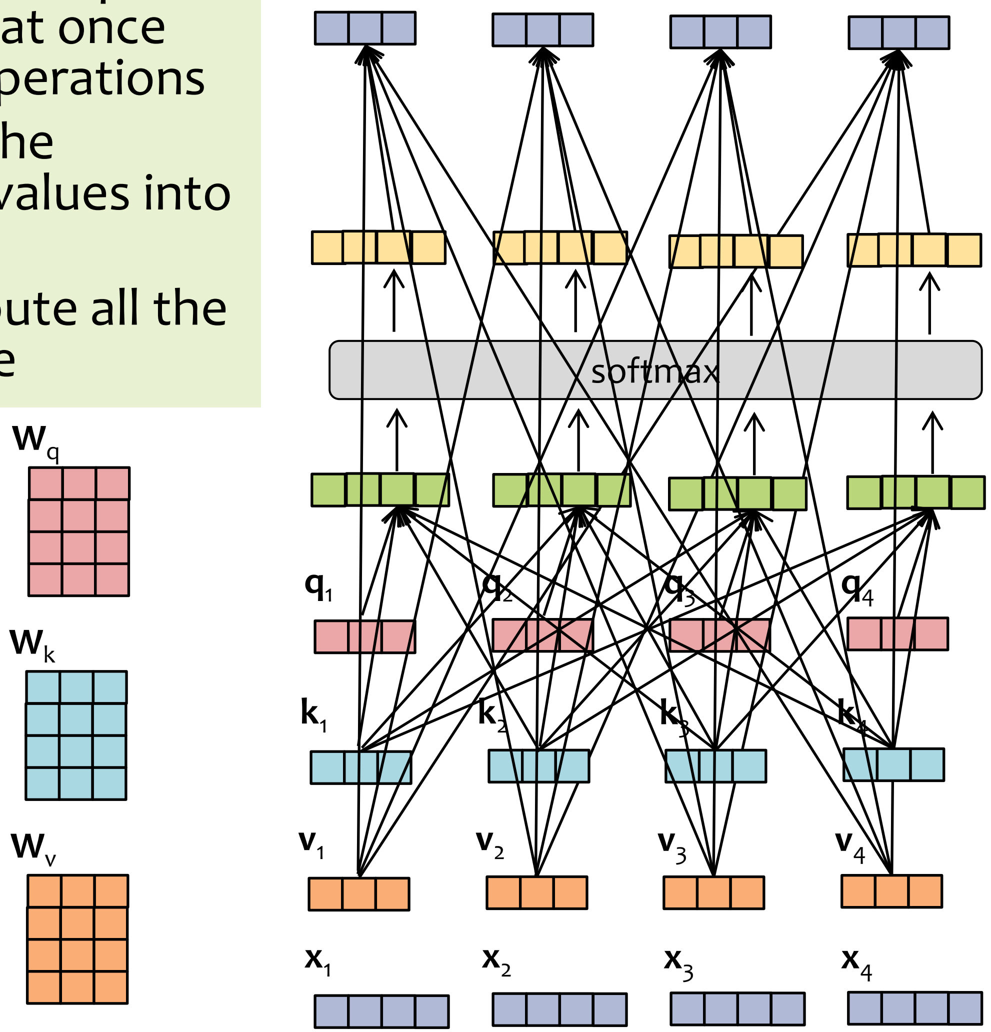
$$\mathbf{K} = [\mathbf{k}_1, \ldots, \mathbf{k}_4]^T = \mathbf{X}\mathbf{W}_k$$

$$\mathbf{V} = [\mathbf{v}_1, \ldots, \mathbf{v}_4]^T = \mathbf{X}\mathbf{W}_v$$

$$\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_4]^T$$

Matrix Version of Single-Headed Attention

- For speed, we compute all the queries at once using matrix operations
- First we pack the queries, keys, values into matrices
- Then we compute all the queries at once



$$\mathbf{X}' = \mathbf{AV} = \text{softmax}(\mathbf{QK}^T/\sqrt{d_k})\mathbf{V}$$

$$\mathbf{A} = [\mathbf{a}_1, \ldots, \mathbf{a}_4]^T = \text{softmax}(\mathbf{S})$$

$$\mathbf{S} = [\mathbf{s}_1, \ldots, \mathbf{s}_4]^T = \mathbf{QK}^T/\sqrt{d_k}$$

$$\mathbf{Q} = [\mathbf{q}_1, \ldots, \mathbf{q}_4]^T = \mathbf{XW}_q$$

$$\mathbf{K} = [\mathbf{k}_1, \ldots, \mathbf{k}_4]^T = \mathbf{XW}_k$$

$$\mathbf{V} = [\mathbf{v}_1, \ldots, \mathbf{v}_4]^T = \mathbf{XW}_v$$

$$\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_4]^T$$

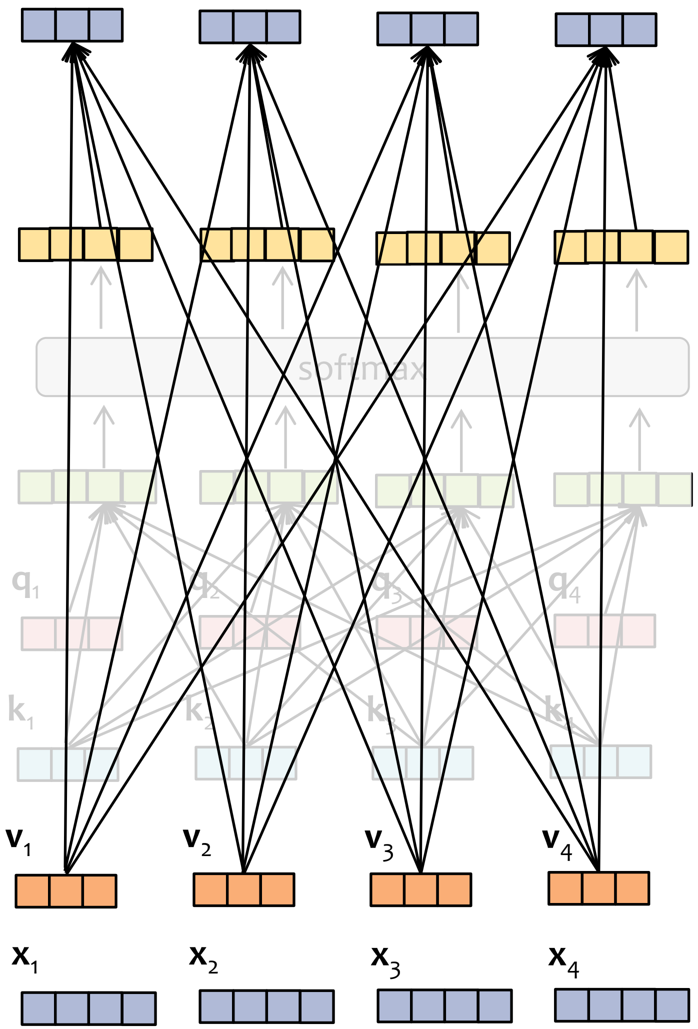
Matrix Version of Single-Headed Attention

Holy cow, that's a lot of new arrows… do we always want/need all of those?

- Suppose we're training our transformer to predict the next token(s) given the input…
- … then attending to tokens that come after the current token is cheating!

So what is this model?

- This version is the *standard* Transformer block. (more on this later!)
- But we want the Transformer LM block
- And that requires masking!



$$\mathbf{X}' = \mathbf{A}\mathbf{V} = \text{softmax}(\mathbf{Q}\mathbf{K}^T/\sqrt{d_k})\mathbf{V}$$

$$\mathbf{A} = [\mathbf{a}_1, \ldots, \mathbf{a}_4]^T = \text{softmax}(\mathbf{S})$$

$$\mathbf{S} = [\mathbf{s}_1, \ldots, \mathbf{s}_4]^T = \mathbf{Q}\mathbf{K}^T/\sqrt{d_k}$$

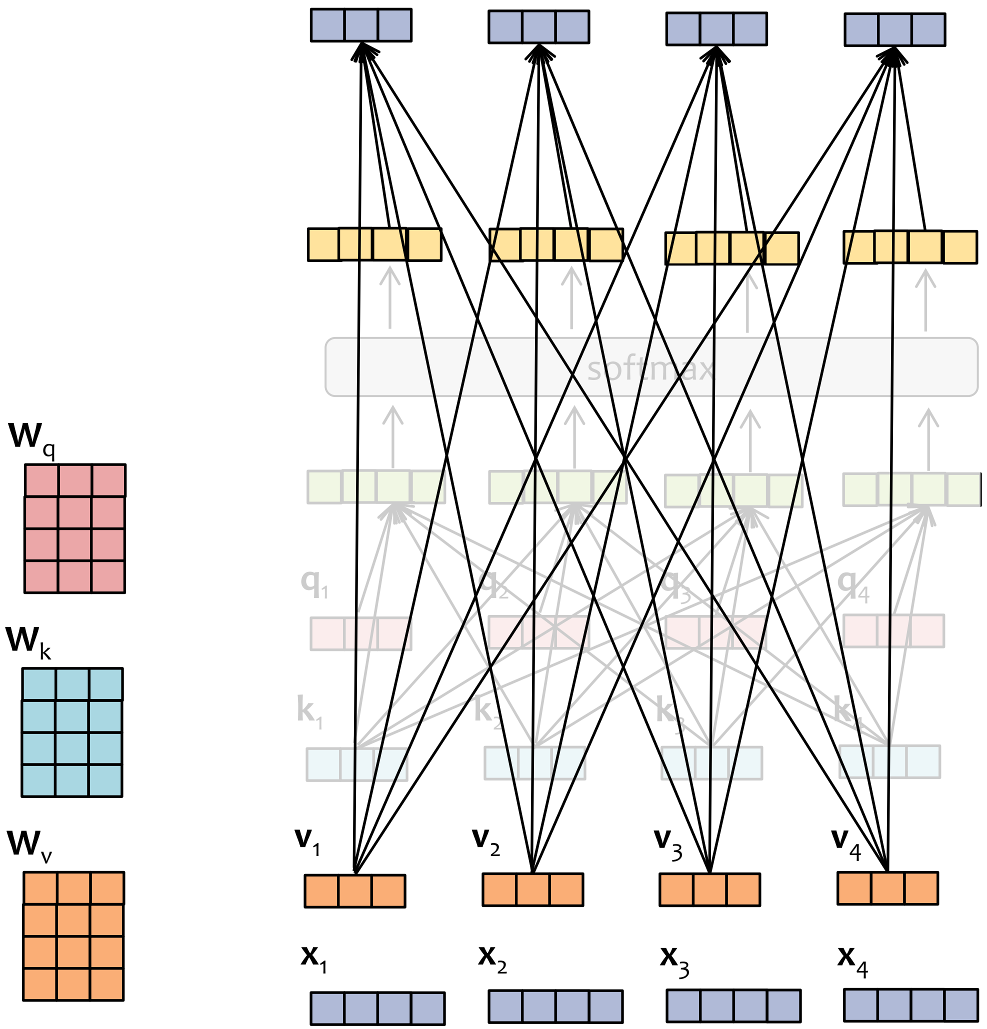
$$\mathbf{Q} = [\mathbf{q}_1, \ldots, \mathbf{q}_4]^T = \mathbf{X}\mathbf{W}_q$$

$$\mathbf{K} = [\mathbf{k}_1, \ldots, \mathbf{k}_4]^T = \mathbf{X}\mathbf{W}_k$$

$$\mathbf{V} = [\mathbf{v}_1, \ldots, \mathbf{v}_4]^T = \mathbf{X}\mathbf{W}_v$$

$$\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_4]^T$$

Matrix Version of Single-Headed Attention



$$\mathbf{X}' = \mathbf{AV} = \text{softmax}(\mathbf{QK}^T / \sqrt{d_k})\mathbf{V}$$

$$\mathbf{A} = \text{softmax}(\mathbf{S})$$

$$\mathbf{S} = \mathbf{QK}^T / \sqrt{d_k}$$

$$\mathbf{Q} = \mathbf{XW}_q$$

$$\mathbf{K} = \mathbf{XW}_k$$

$$\mathbf{V} = \mathbf{XW}_v$$

$$\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_4]^T$$

Question: How is the softmax applied?

A. column-wise

B. row-wise

Answer:

Matrix Version of Single-Headed (Causal) Attention

Insight: if some element in the input to the softmax is -∞, then the corresponding output is 0!

Question: For a causal LM which is the correct matrix?

A:

$$\mathbf{M} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ -\infty & 0 & 0 & 0 \\ -\infty & -\infty & 0 & 0 \\ -\infty & -\infty & -\infty & 0 \end{bmatrix}$$

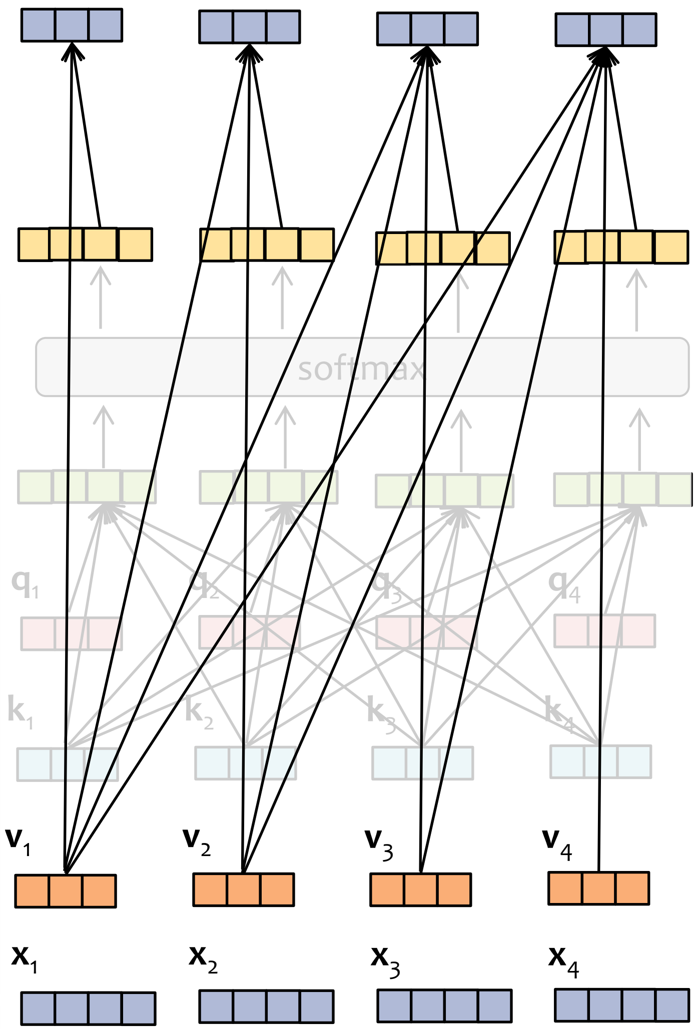
B:

$$\mathbf{M} = \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

C:

$$\mathbf{M} = \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ -\infty & 0 & -\infty & -\infty \\ -\infty & -\infty & 0 & -\infty \\ -\infty & -\infty & -\infty & 0 \end{bmatrix}$$

Answer:



$$\mathbf{X}' = \mathbf{AV} = \text{softmax}(\mathbf{QK}^T / \sqrt{d_k} + \mathbf{M})\mathbf{V}$$

$$\mathbf{A}_{\text{causal}} = \text{softmax}(\mathbf{S} + \mathbf{M})$$

$$\mathbf{S} = \mathbf{QK}^T / \sqrt{d_k}$$

$$\mathbf{Q} = \mathbf{XW}_q$$

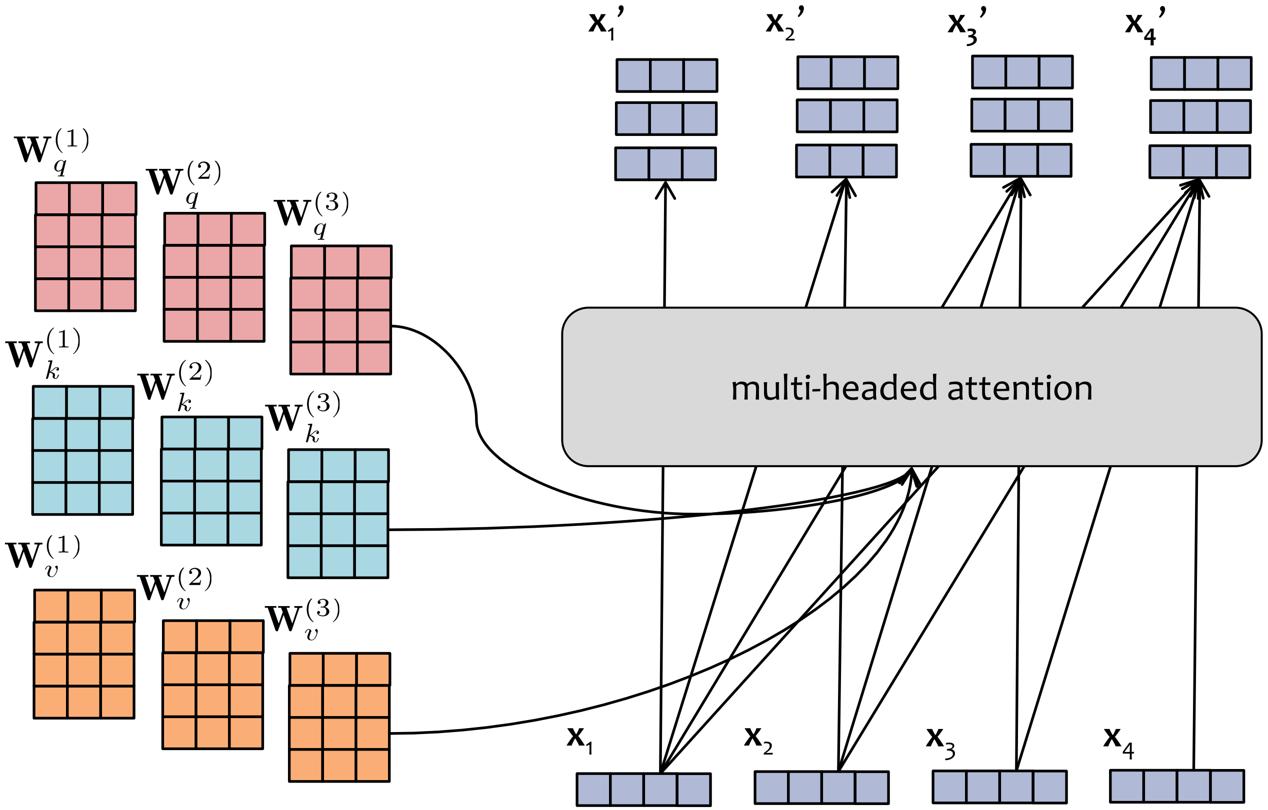
$$\mathbf{K} = \mathbf{XW}_k$$

$$\mathbf{V} = \mathbf{XW}_v$$

$$\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_4]^T$$

In practice, the attention weights are computed for all time steps T, then we mask out (by setting to –inf) all the inputs to the softmax that are for the timesteps to the right of the query.

Matrix Version of Multi-Headed (Causal) Attention



$$\mathbf{X} = \text{concat}(\mathbf{X}'^{(1)}, \mathbf{X}'^{(2)}, \mathbf{X}'^{(3)})$$

$$\mathbf{X}'^{(i)} = \text{softmax}\left(\frac{\mathbf{Q}^{(i)}(\mathbf{K}^{(i)})^T}{\sqrt{d_k}} + \mathbf{M}\right)\mathbf{V}^{(i)}$$

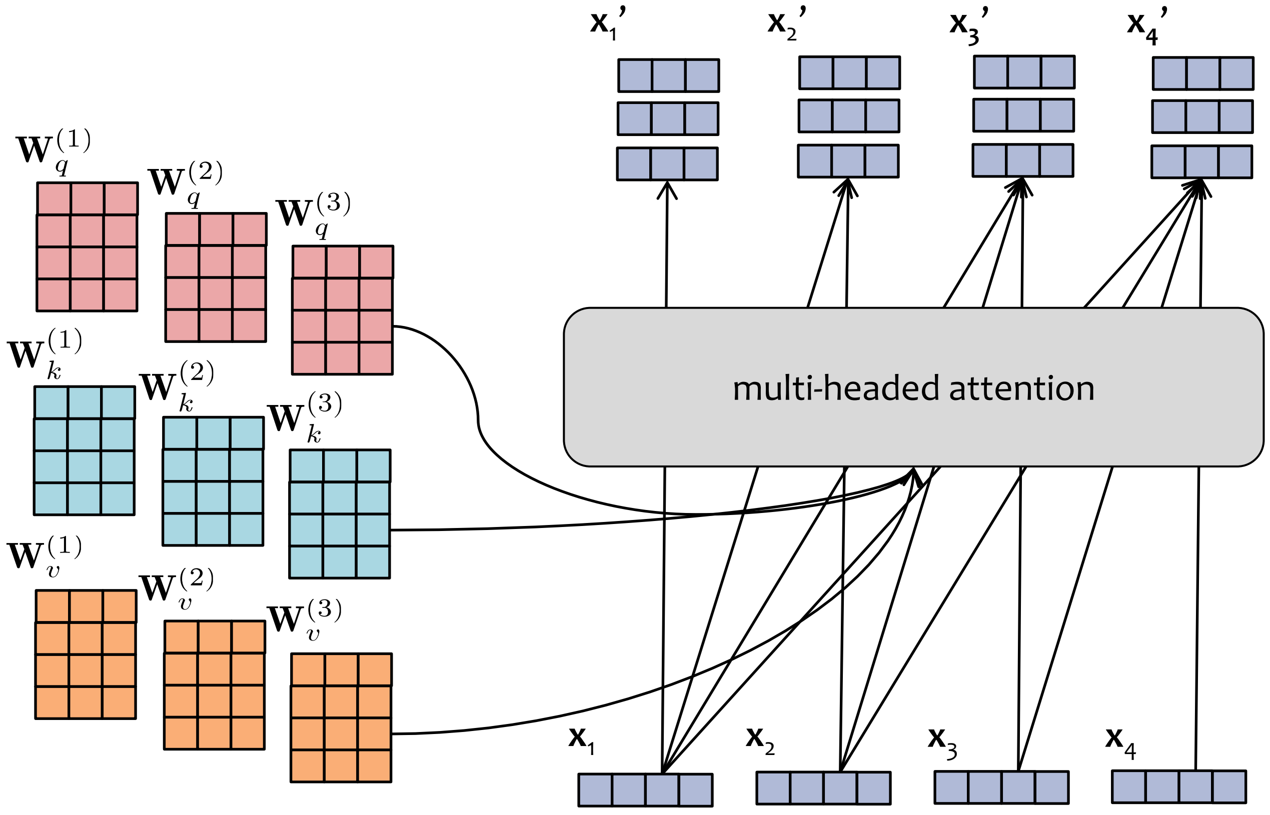
$$\mathbf{Q}^{(i)} = \mathbf{X}\mathbf{W}_q^{(i)}$$

$$\mathbf{K}^{(i)} = \mathbf{X}\mathbf{W}_k^{(i)}$$

$$\mathbf{V}^{(i)} = \mathbf{X}\mathbf{W}_v^{(i)}$$

$$\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_4]^T$$

Matrix Version of Multi-Headed (Causal) Attention



$$\mathbf{X} = \text{concat}(\mathbf{X}'^{(1)}, \ldots, \mathbf{X}'^{(h)})$$

$$\mathbf{X}'^{(i)} = \text{softmax}\left(\frac{\mathbf{Q}^{(i)}(\mathbf{K}^{(i)})^T}{\sqrt{d_k}} + \mathbf{M}\right)\mathbf{V}^{(i)}$$

$$\mathbf{Q}^{(i)} = \mathbf{X}\mathbf{W}_q^{(i)}$$

$$\mathbf{K}^{(i)} = \mathbf{X}\mathbf{W}_k^{(i)}$$

$$\mathbf{V}^{(i)} = \mathbf{X}\mathbf{W}_v^{(i)}$$

$$\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_4]^T$$

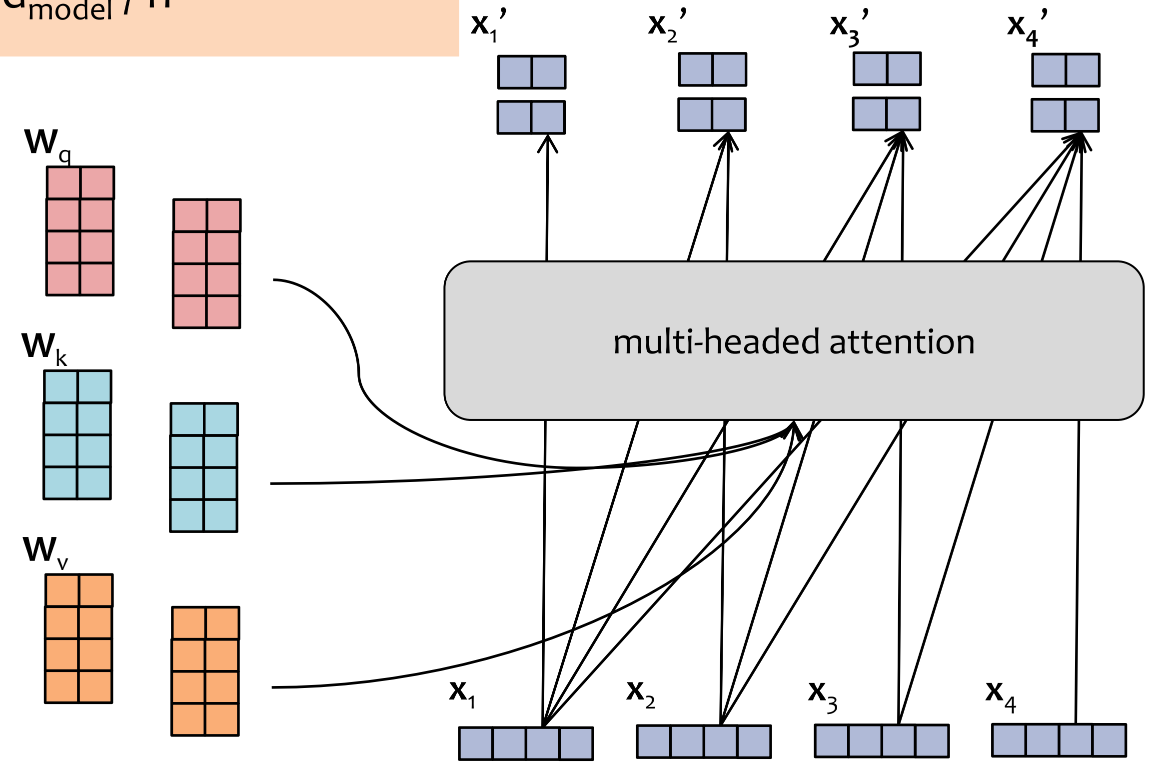
Recall:
To ensure the dimension of the **input** embedding $\mathbf{x}_t$ is the same as the **output** embedding $\mathbf{x}_t'$, Transformers usually choose the embedding sizes and number of heads appropriately:

- d_{model} = dim. of inputs
- d_k = dim. of each output
- h = # of heads
- Choose $d_k = d_{model} / h$

...on of Multi-Headed (Causal) Attention

$$\mathbf{X} = \text{concat}(\mathbf{X}'^{(1)}, \ldots, \mathbf{X}'^{(h)})$$

$$\mathbf{X}'^{(i)} = \text{softmax}\left(\frac{\mathbf{Q}^{(i)}(\mathbf{K}^{(i)})^T}{\sqrt{d_k}} + \mathbf{M}\right)\mathbf{V}^{(i)}$$



$$\mathbf{Q}^{(i)} = \mathbf{X}\mathbf{W}_q^{(i)}$$

$$\mathbf{K}^{(i)} = \mathbf{X}\mathbf{W}_k^{(i)}$$

$$\mathbf{V}^{(i)} = \mathbf{X}\mathbf{W}_v^{(i)}$$

$$\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_4]^T$$

Two parts: Deep Learning and Language Modeling

Recap So Far

Deep Learning

- AutoDiff
  - is a tool for **computing gradients** of a differentiable function, b = f(a)
  - the key building block is a **module** with a forward() and backward()
  - sometimes define f as **code** in forward() by chaining existing modules together
- Computation Graphs
  - are another way to define f (more conducive to slides)
  - so far, we saw two (deep) computation graphs
    - 1) RNN-LM
    - 2) Transformer-LM
    - (Transformer-LM was kind of complicated)

Language Modeling

- key idea: condition on previous words to **sample the next word**
- to define the **probability** of the next word…
  - …n-gram LM uses collection of massive 50k-sided **dice**
  - …RNN-LM or Transformer-LM use a **neural network**

- Learning an LM
  - n-gram LMs are easy to learn: just **count** co-occurrences!
  - so far, we said **nothing about how to learn** an RNN-LM or Transformer-LM
  - So let's figure out that out next…