



# Transformers, AutoDiff + Pre-training, Fine-Tuning, In-context Learning

Matt Gormley & Geoff Gordon

Lecture 19

Nov. 3, 2025

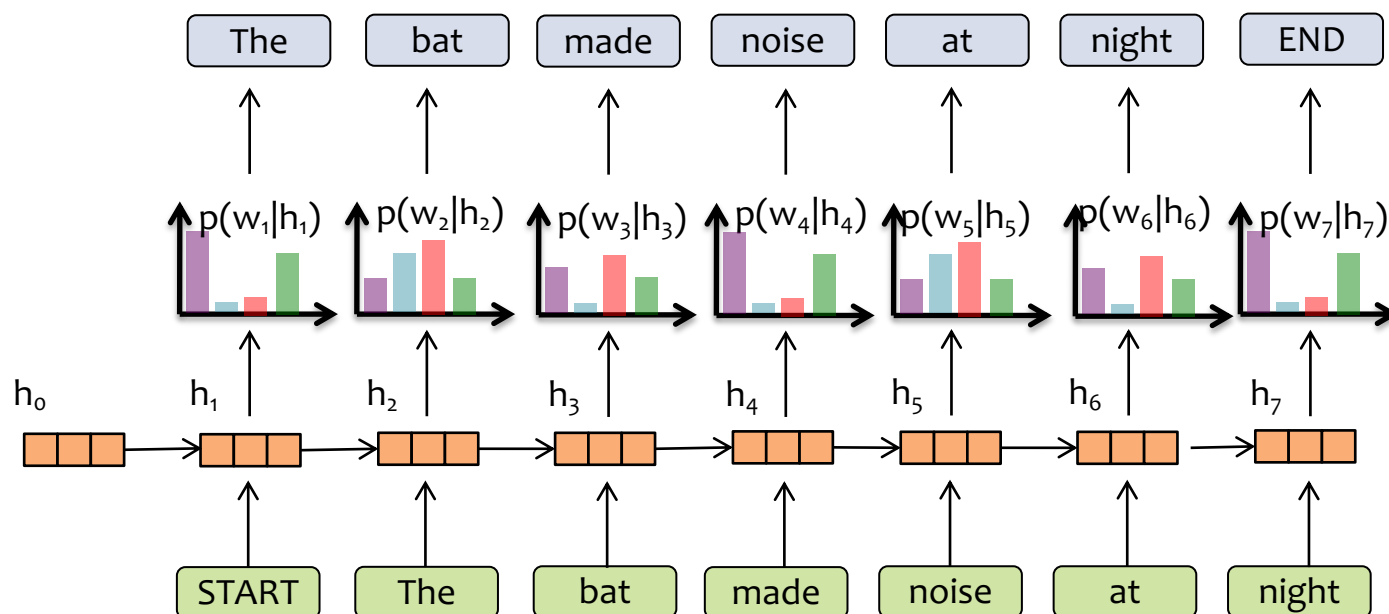
# Reminders

- **Exam 2: Thu, Nov 6, 7:00 pm – 9:00 pm**
  - Scope: Lectures 8 - 16
- **Homework 7: Deep Learning & LLMs**
  - Out: Thu, Nov 6
  - Due: Sun, Nov 16 11:59pm

Generative Pretrained Transformers (GPT)

# **TRANSFORMER LANGUAGE MODELS**

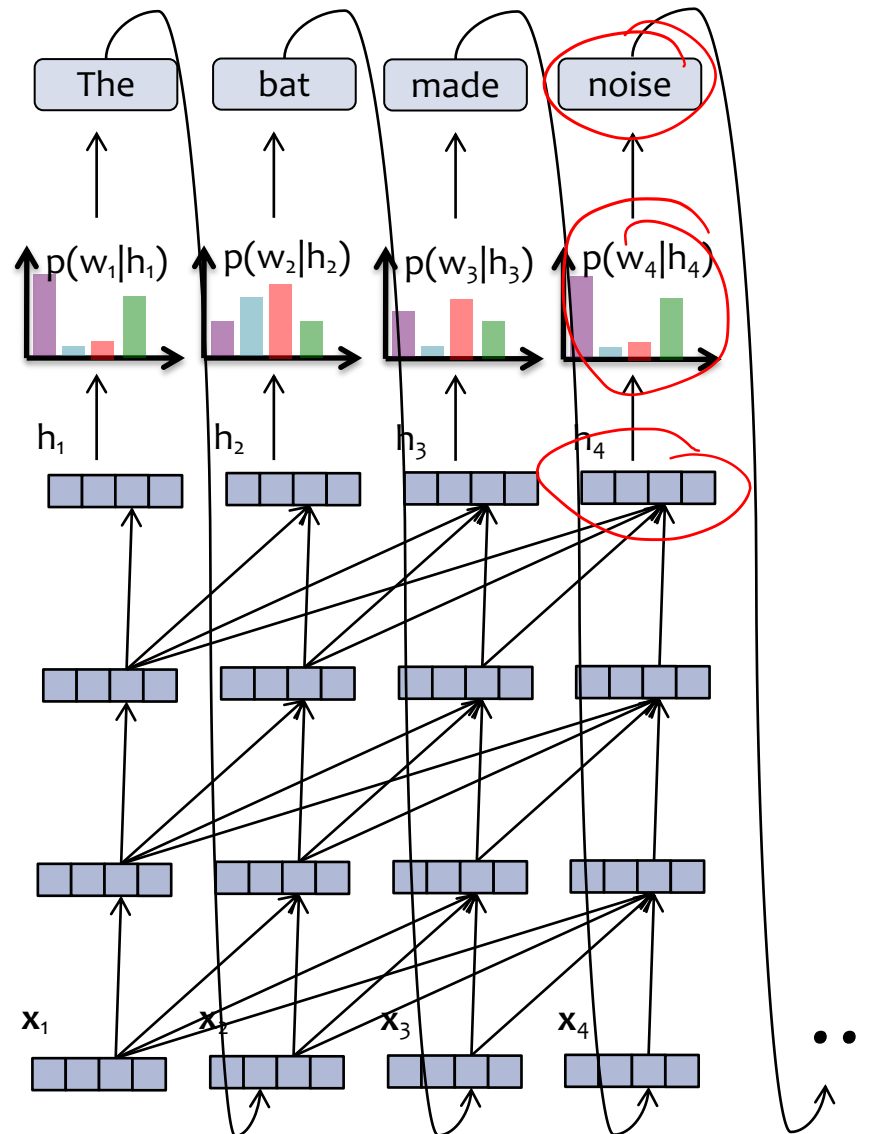
# RNN Language Model



## Key Idea:

- (1) convert all previous words to a **fixed length vector**
- (2) define distribution  $p(w_t | f_{\theta}(w_{t-1}, \dots, w_1))$  that conditions on the vector  $\mathbf{h}_t = f_{\theta}(w_{t-1}, \dots, w_1)$

# Transformer Language Model



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

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

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

# 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  $\mathbf{b} \in \mathbb{R}^K$ :

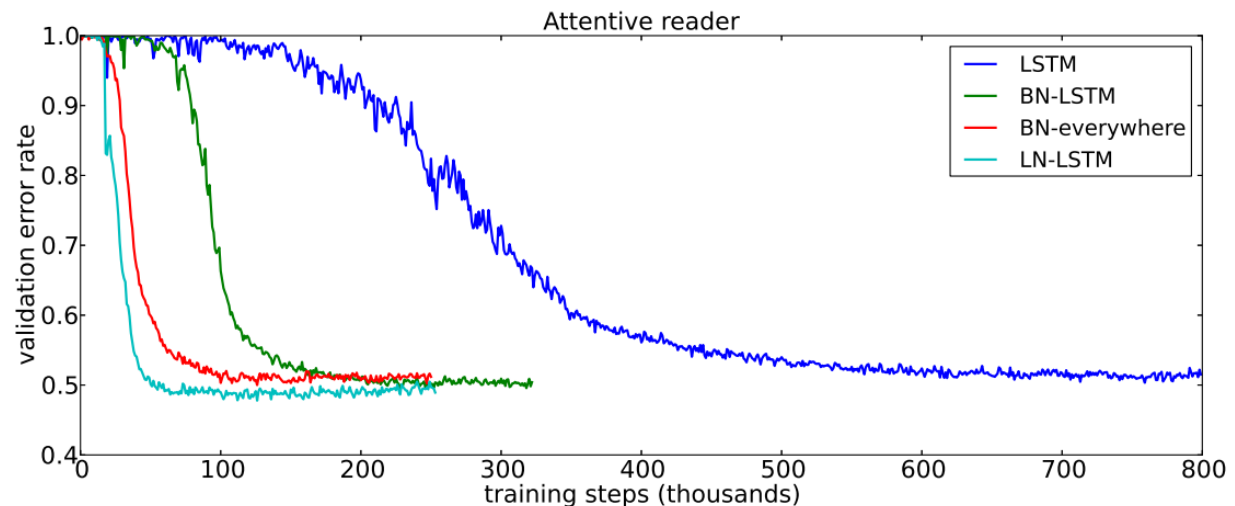
$$\mathbf{b} = \left( \gamma \odot \left( \frac{\mathbf{a} - \mu}{\sigma} \right) \right) \oplus \beta$$

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  $\gamma \in \mathbb{R}^K, \beta \in \mathbb{R}^K$ .

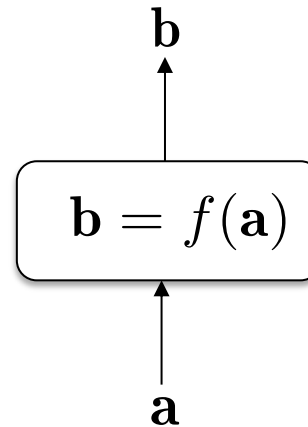
$\odot$  and  $\oplus$  denote elementwise multiplication and addition.



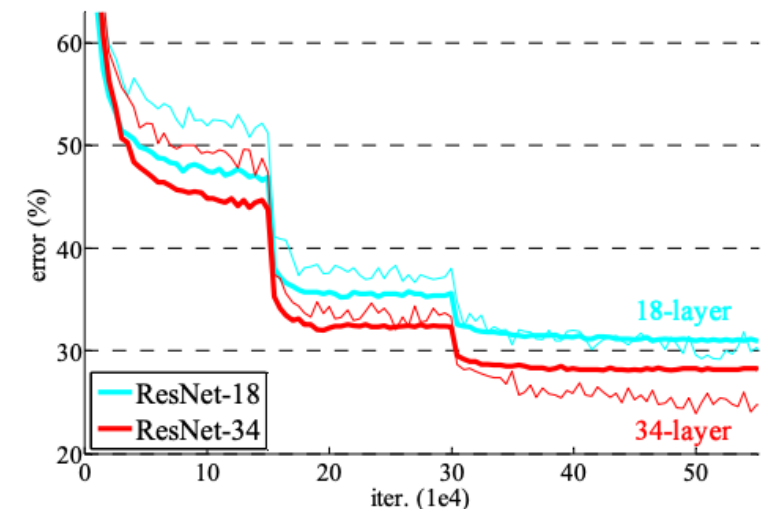
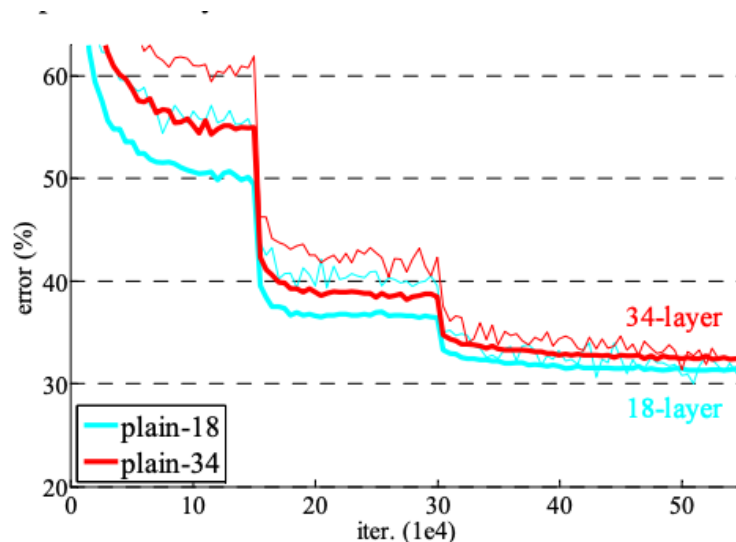
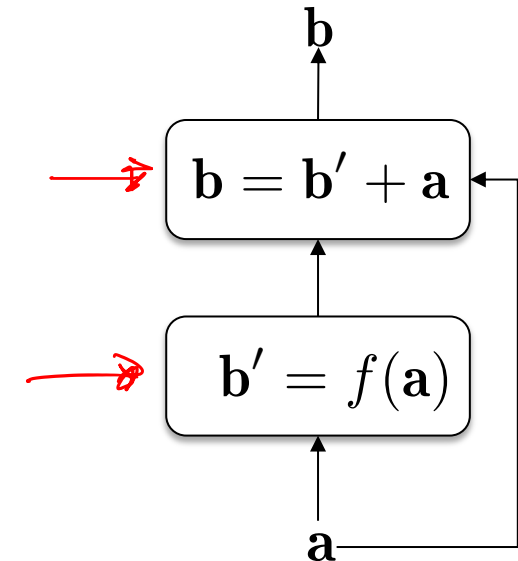
# Residual Connections

- **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



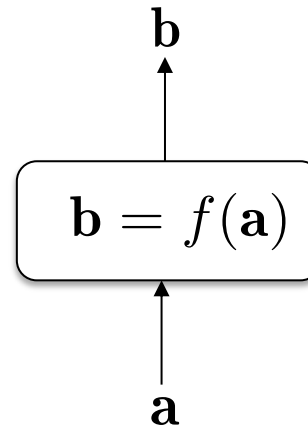
Residual Connection



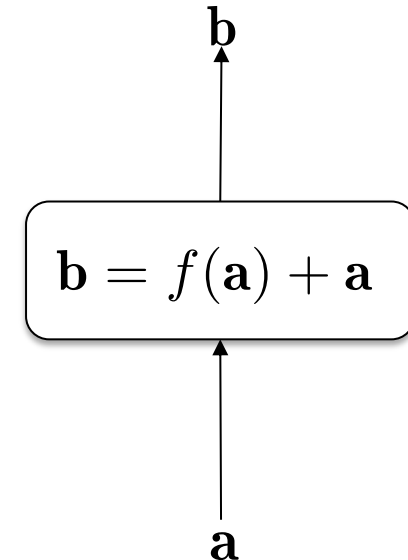
# Residual Connections

- **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



## 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).

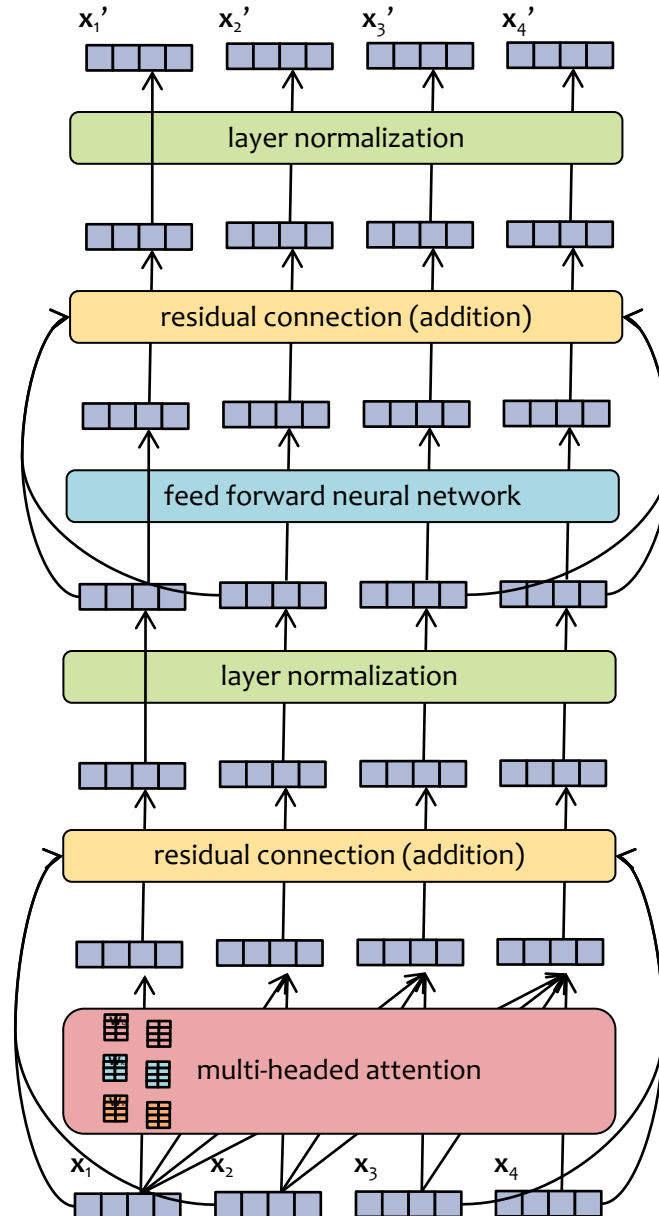


# Transformer Layer

## Post-LN Version:

This is the version of the Transformer Layer that was introduced in the original paper in 2017.

The LayerNorm modules occur at the end of each set of 3 layers.



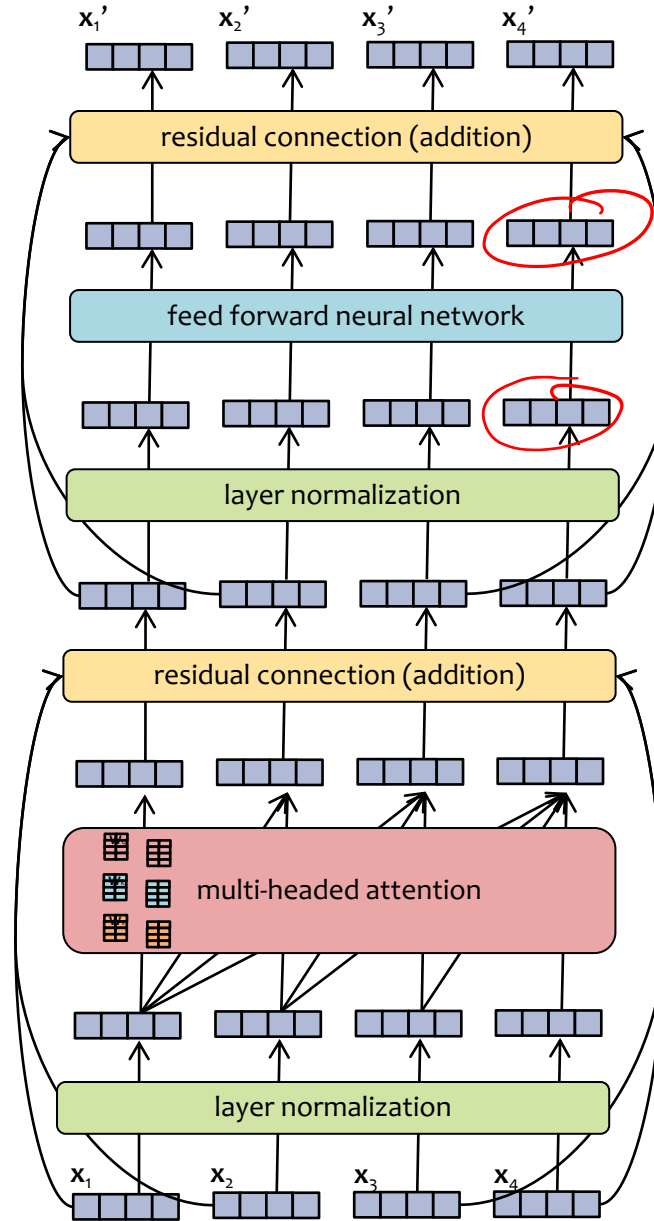
**Each layer** of a Transformer LM consists of several **sublayers**:

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

# Transformer Layer

## Pre-LN Version:

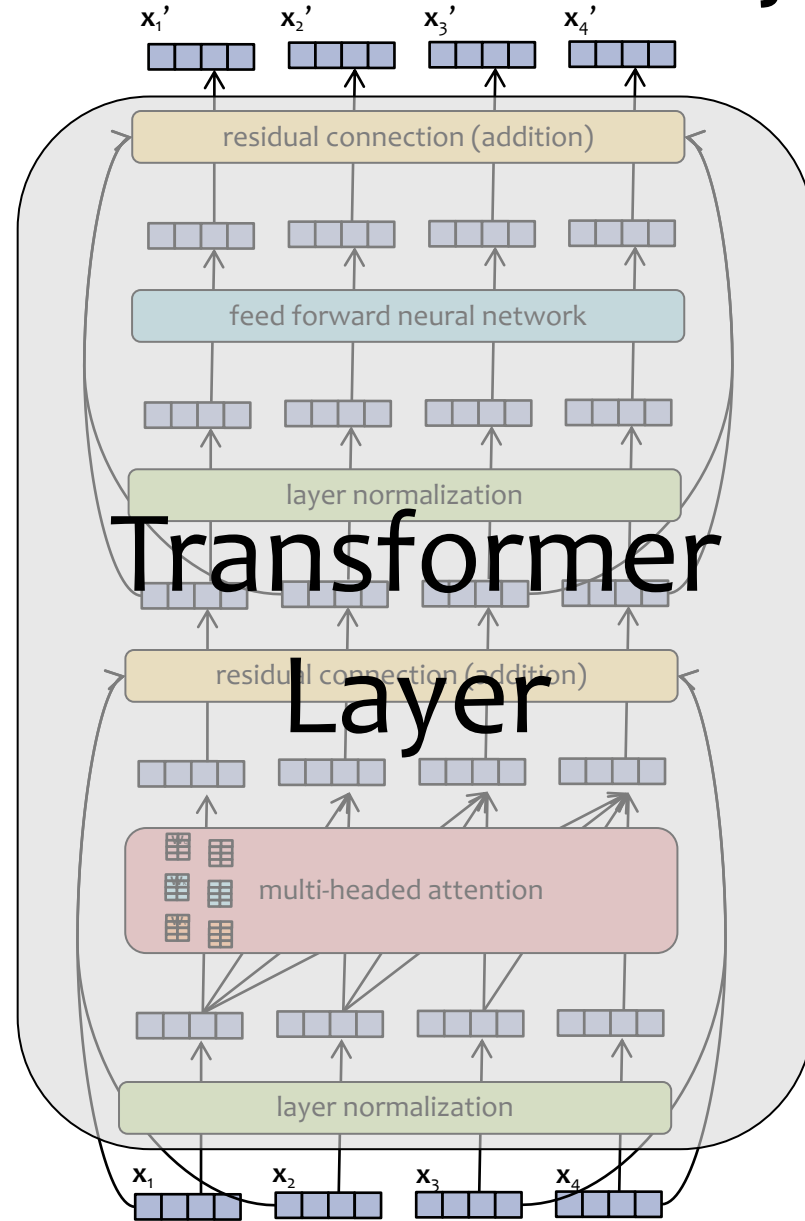
However, subsequent work found that reordering such that the LayerNorm's came at the beginning of each set of 3 layers, the multi-headed attention and feed-forward NN layers tend to be better behaved (i.e. tricks like warm-up are less important).



**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



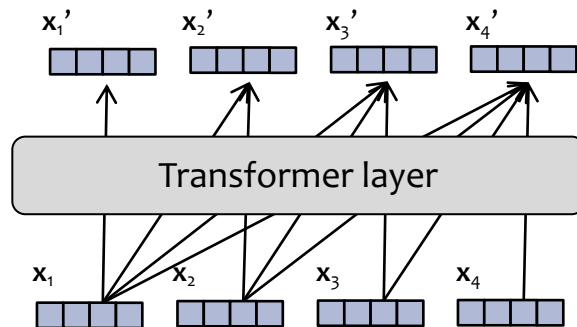
**Each layer** of a Transformer LM consists of several **sublayers**:

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

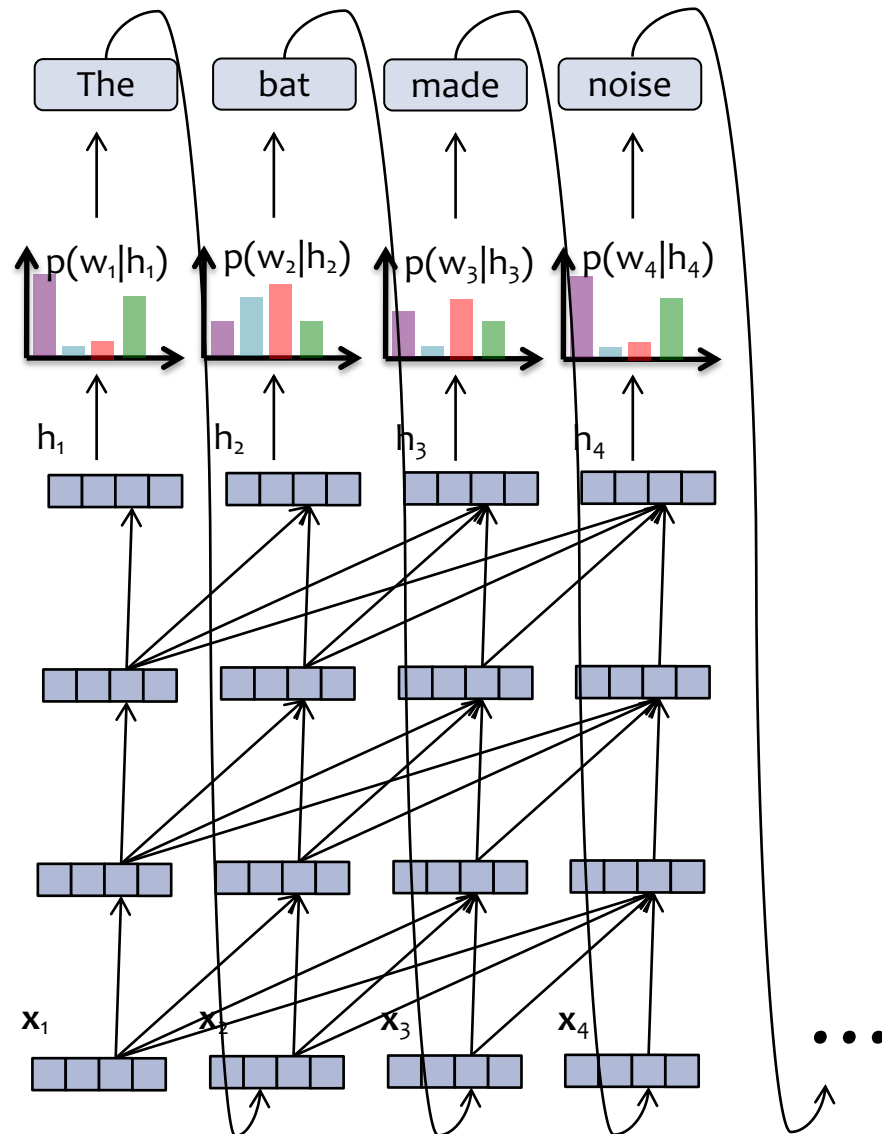
# Transformer Layer

**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



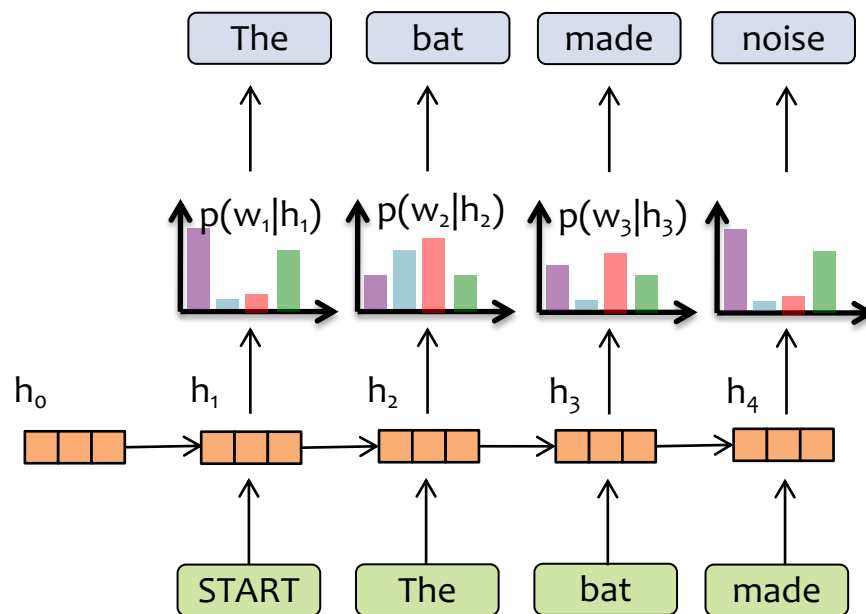
- Each layer** of a Transformer LM consists of several **sublayers**:
1. attention
  2. feed-forward neural network
  3. layer normalization
  4. residual connections

# Computational Complexity

## Important!

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

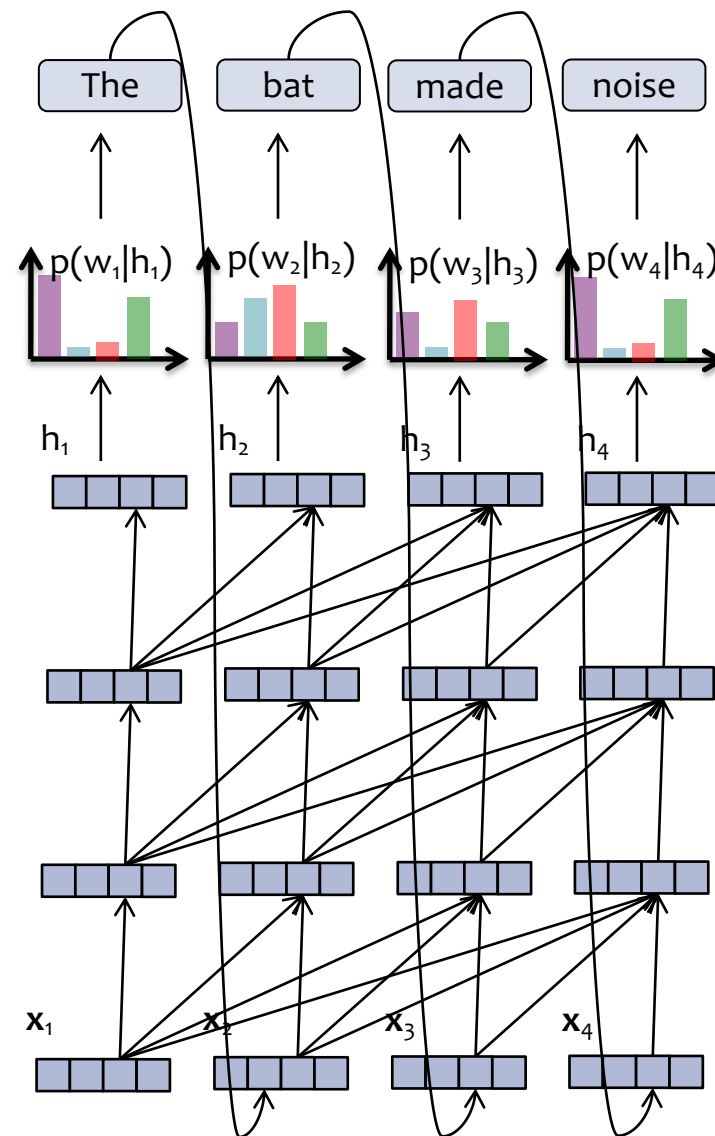
RNN LM



In RNNs, each hidden looks **only at the previous hidden**.

In Transformers, because of attention, each hidden vector looks back at the hidden vectors of the **current and previous timesteps in the previous layer**.

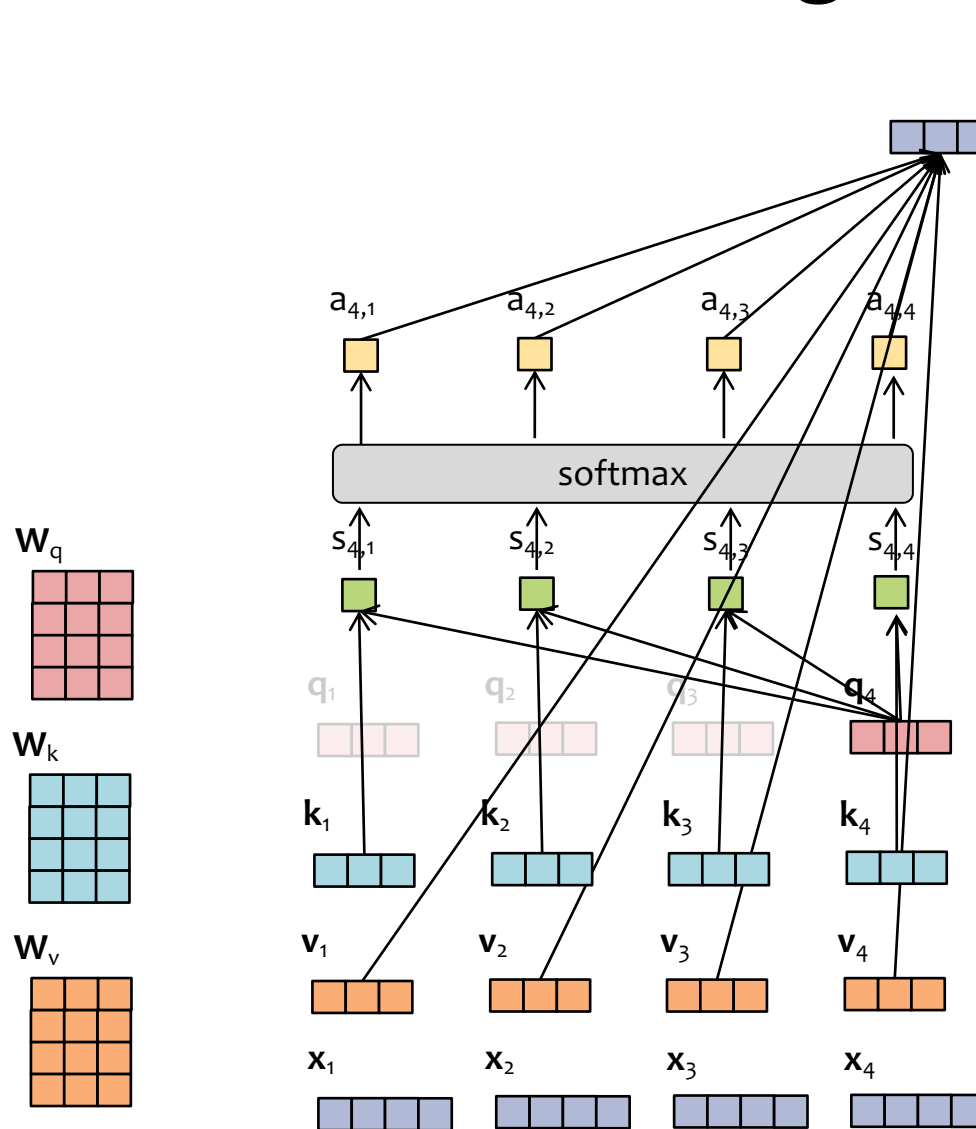
Transformer LM



# **IMPLEMENTING A TRANSFORMER LM**



# Matrix Version of Single-Headed Attention



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

$\mathbf{a}_4 = \text{softmax}(s_4)$  attention weights

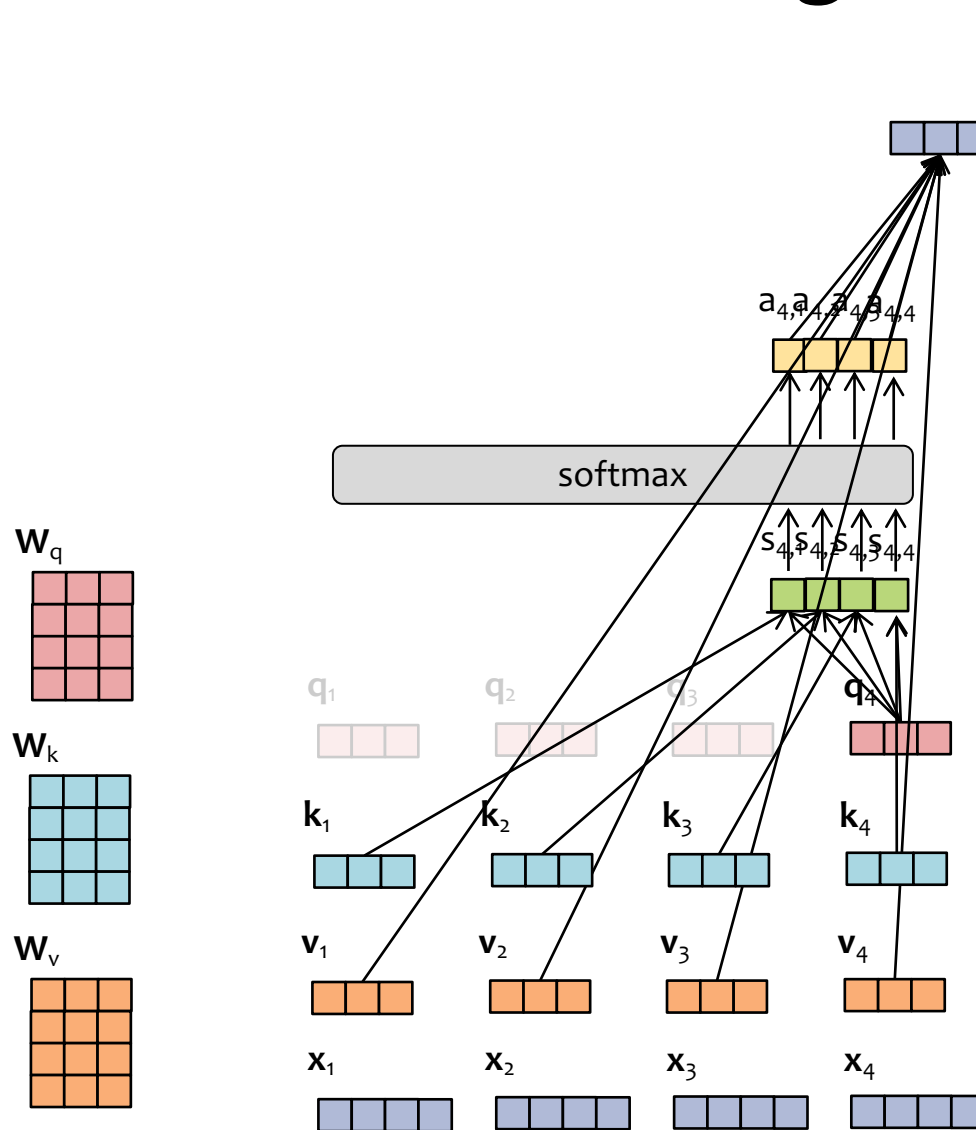
$s_{4,j} = \mathbf{k}_j^T \mathbf{q}_4 / \sqrt{d_k}$  scores

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



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

$\mathbf{a}_4 = \text{softmax}(s_4)$  attention weights

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

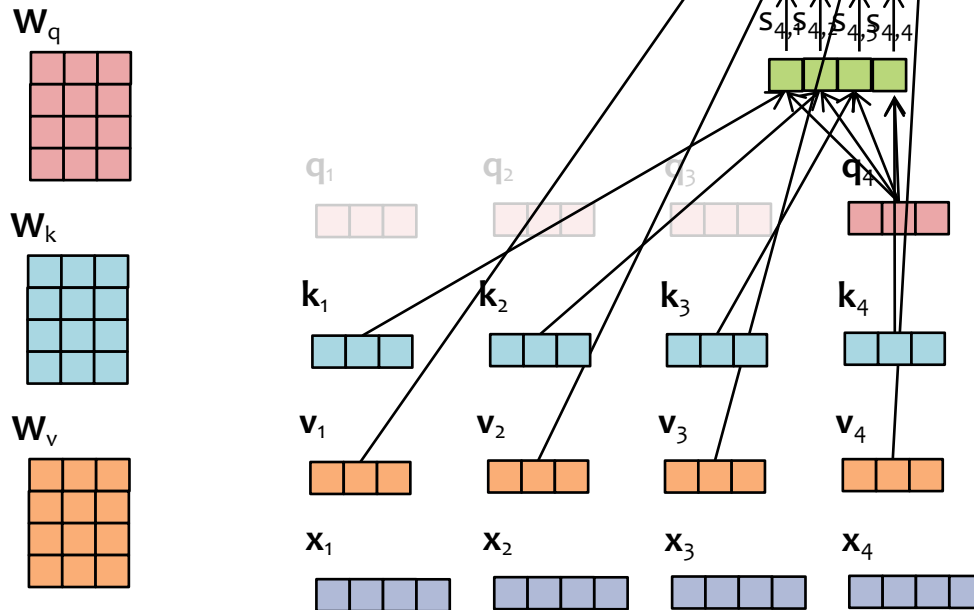
$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j$  queries

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



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

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

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

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

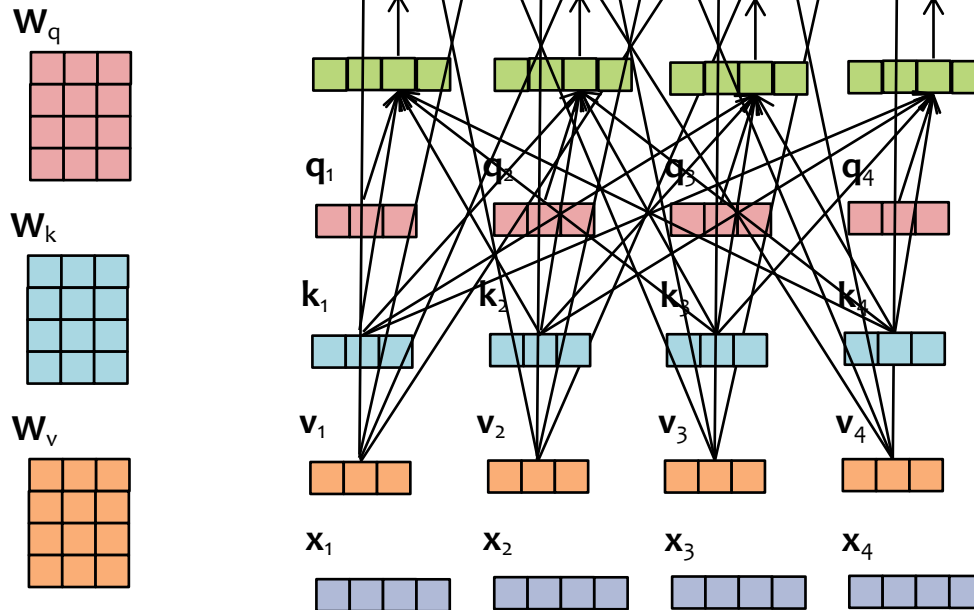
$$\mathbf{K} = [\mathbf{k}_1, \dots, \mathbf{k}_4]^T = \mathbf{X}\mathbf{W}_k$$

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

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

# 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



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

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

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

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

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

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

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

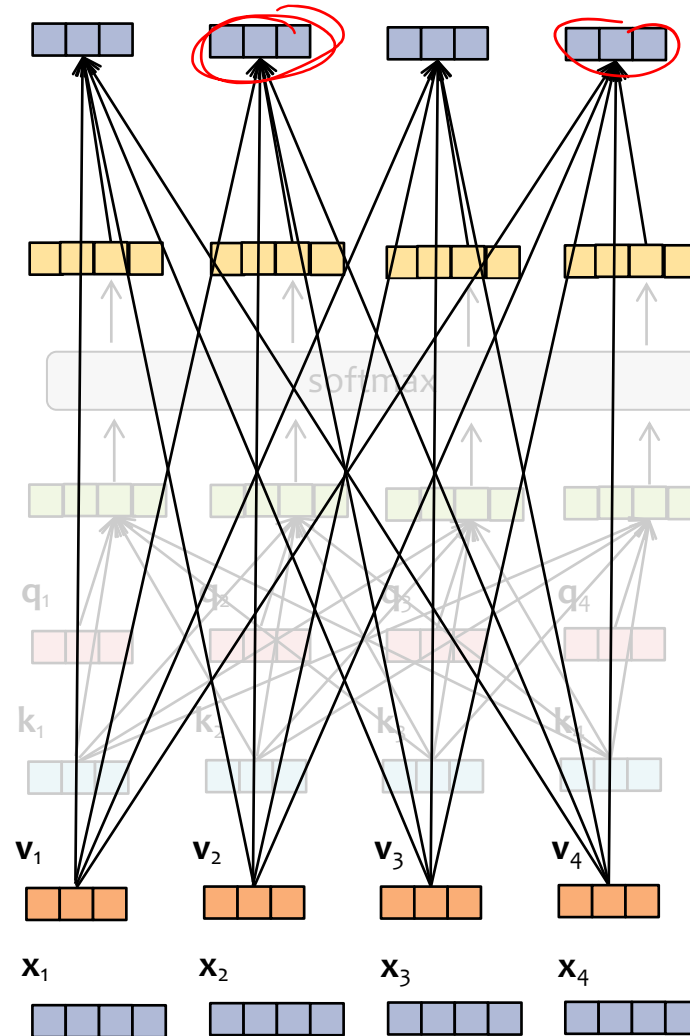
# Matrix Version of Single-Headed Attention

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!



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

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

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

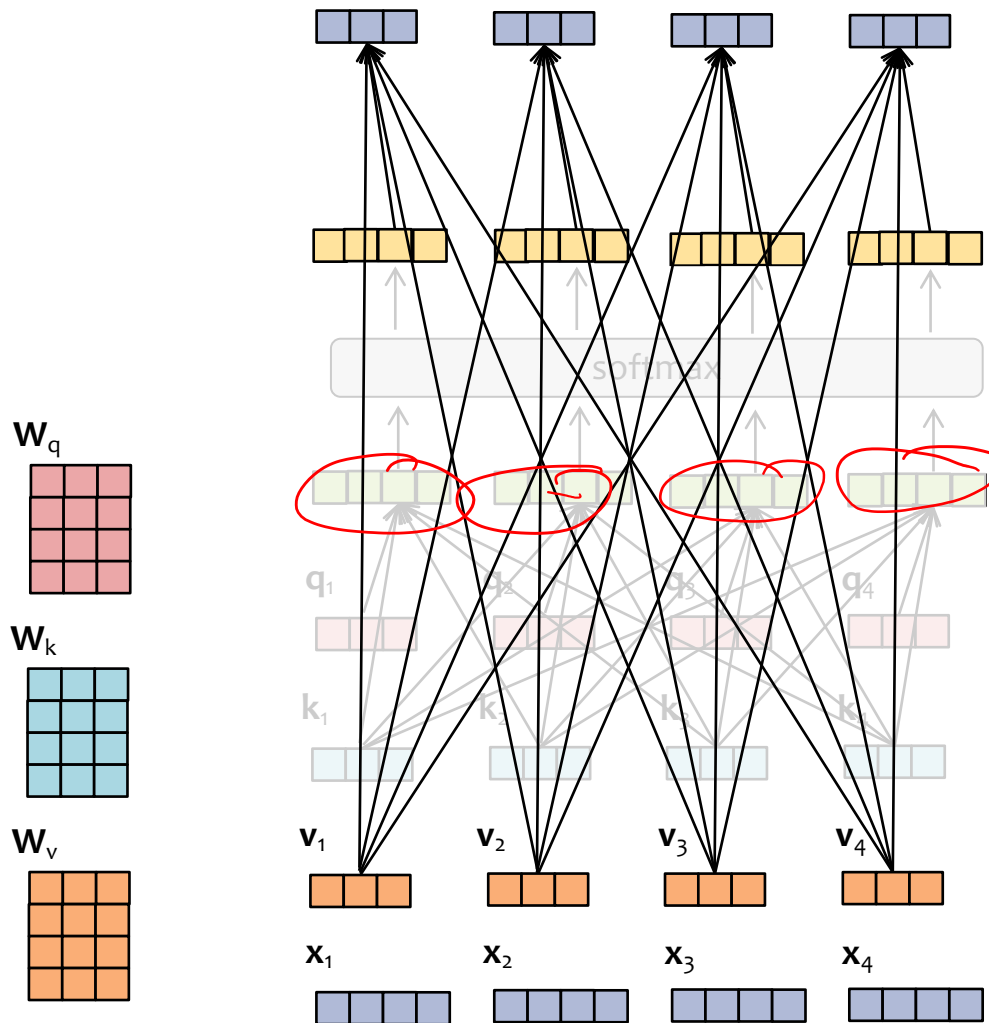
$$\mathbf{Q} = [\mathbf{q}_1, \dots, \mathbf{q}_4]^T = \mathbf{X}\mathbf{W}_q$$

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

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

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

# Matrix Version of Single-Headed Attention



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

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

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

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_q$$

$$\mathbf{K} = \mathbf{X}\mathbf{W}_k$$

$$\mathbf{V} = \mathbf{X}\mathbf{W}_v$$

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

**Poll Question:** How is the softmax applied?

- A. column-wise 32%  
 B. row-wise 68%

**Answer:**

*A = toxic*

# Matrix Version of Single-Headed (Causal) Attention

**Insight:** if some element in the input to the softmax is  $-\infty$ , then the corresponding output is 0!

*A = toxic*

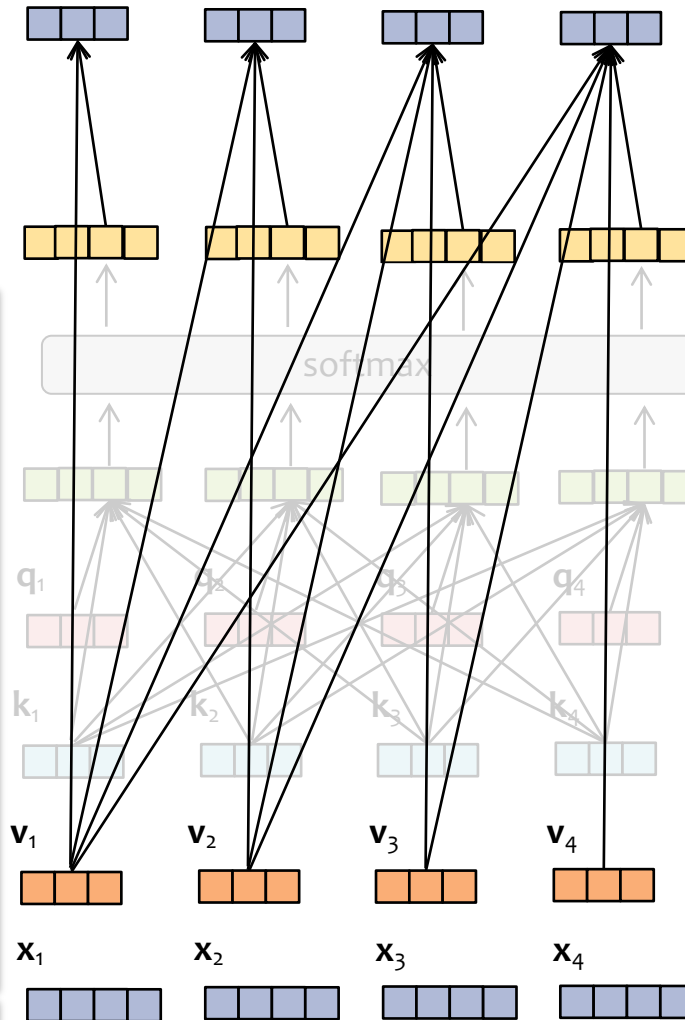
**Poll Question:** For a causal LM which is the correct matrix?

A:  $M = \begin{bmatrix} 0 & 0 & 0 & 0 \\ -\infty & 0 & 0 & 0 \\ -\infty & -\infty & 0 & 0 \\ -\infty & -\infty & -\infty & 0 \end{bmatrix}$  *14%*

**B:**  $M = \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{bmatrix}$  *74%*

C:  $M = \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ -\infty & 0 & -\infty & -\infty \\ -\infty & -\infty & 0 & -\infty \\ -\infty & -\infty & -\infty & 0 \end{bmatrix}$  *10%*

**Answer:**



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

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

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

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_q$$

$$\mathbf{K} = \mathbf{X}\mathbf{W}_k$$

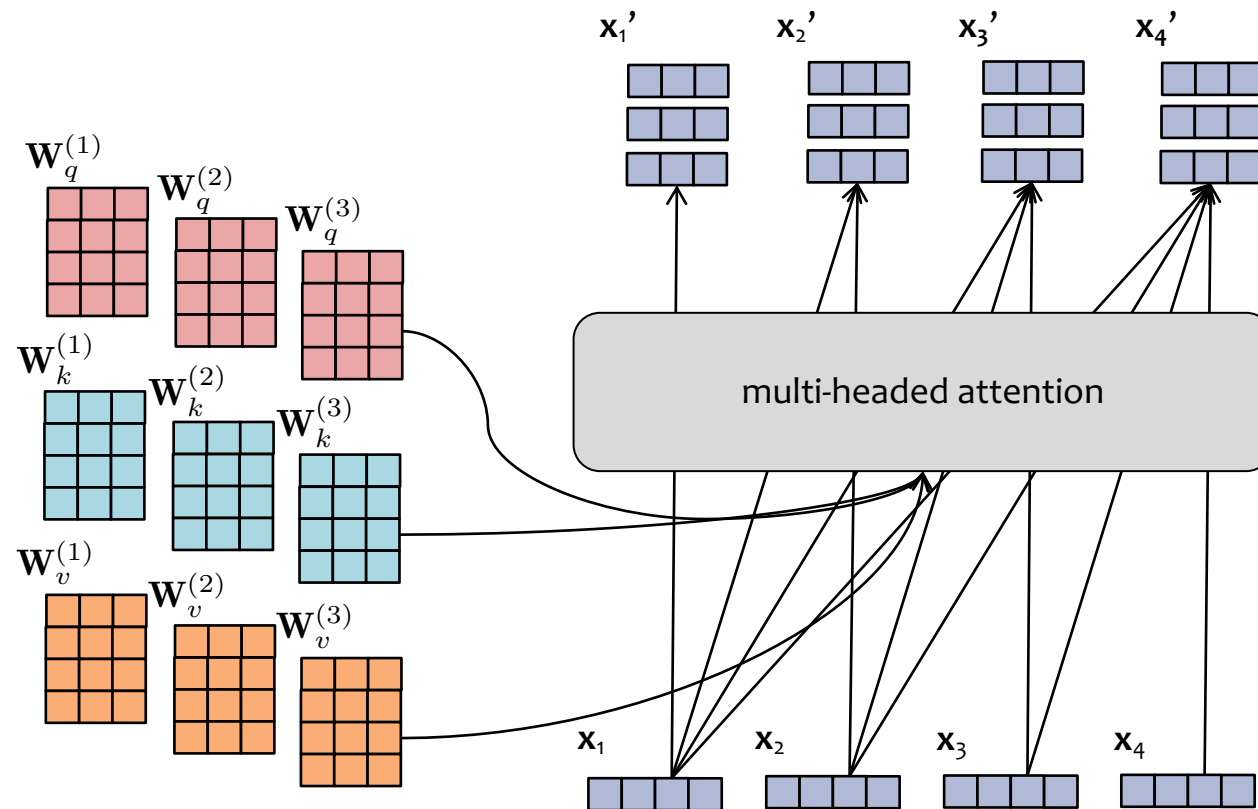
$$\mathbf{V} = \mathbf{X}\mathbf{W}_v$$

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

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

# Matrix Version of Multi-Headed (Causal) Attention

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



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

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

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

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

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

$\forall i \in \{1, \dots, h\}$



# **PRACTICALITIES OF TRANSFORMER LMS**

~~NA = [all zeros]~~

# In-Class Poll

## Poll Question:

Suppose we have the following input embeddings and attention weights:

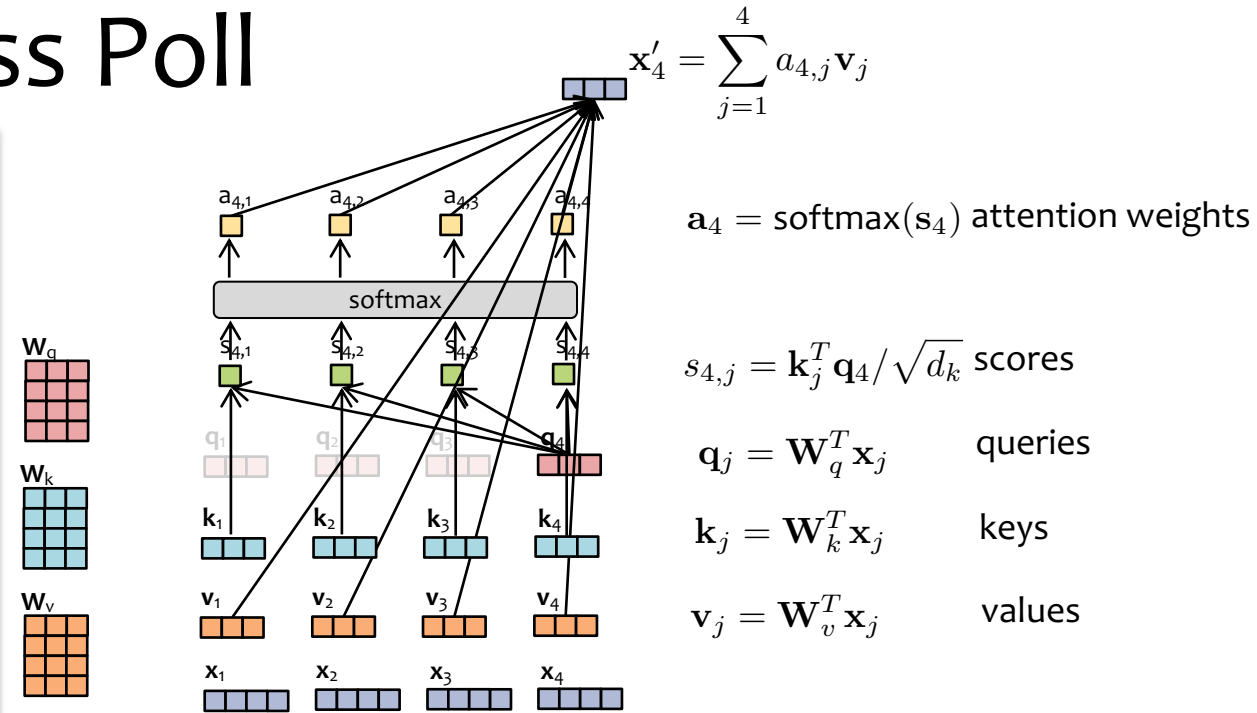
- $x_1 = [1, 0, 0, 0]$   $a_{4,1} = 0.1$
- $x_2 = [0, 1, 0, 0]$   $a_{4,2} = 0.2$
- $x_3 = [0, 0, 2, 0]$   $a_{4,3} = 0.6$
- $x_4 = [0, 0, 0, 1]$   $a_{4,4} = 0.1$

And  $W_v = I$ . Then we can compute  $x'_4$ .

Now suppose we swap the embeddings  $x_2$  and  $x_3$  such that

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

What is the new value of  $x'_4$ ?



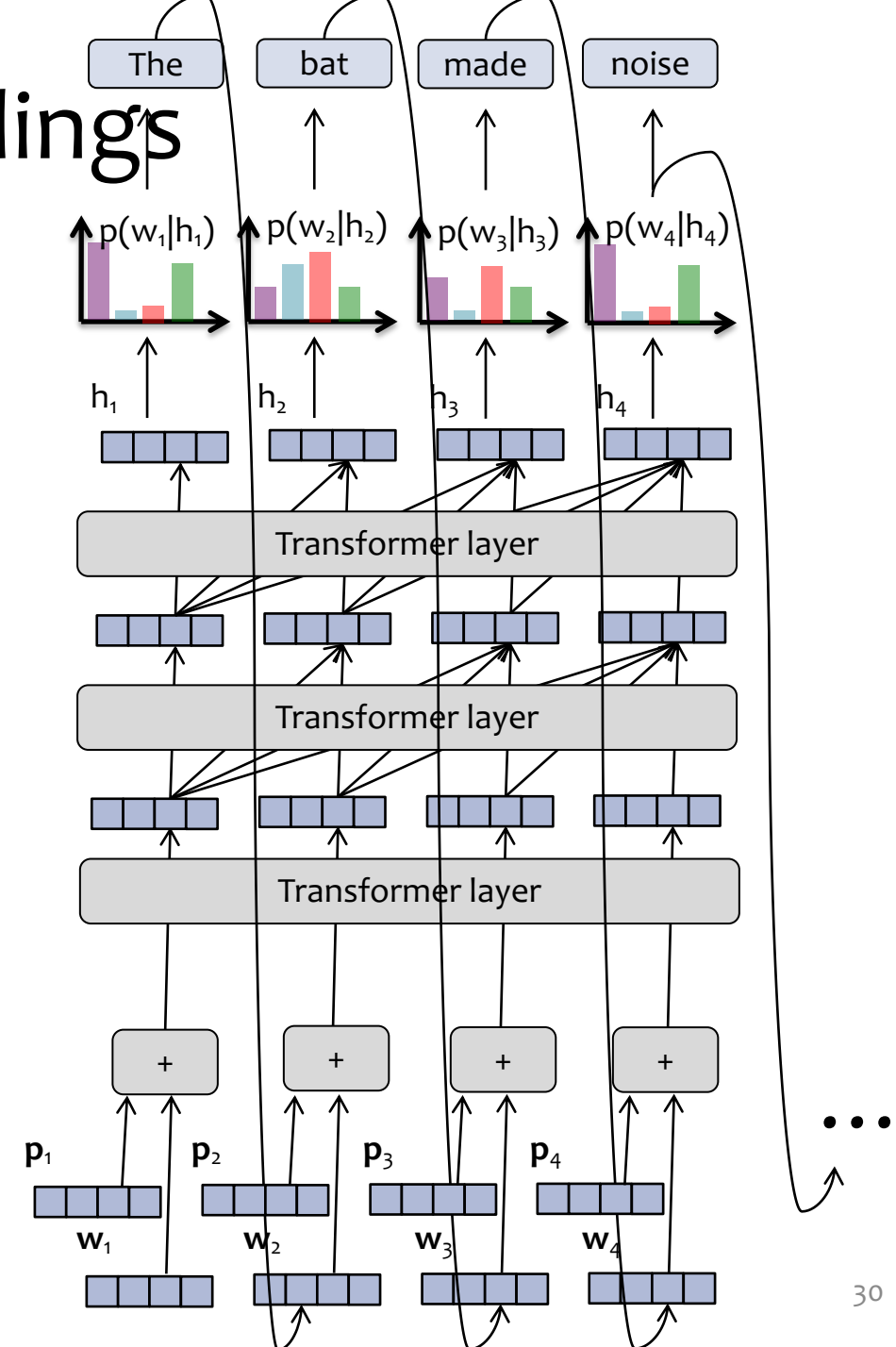
**Answer:** (no toxic option)

WRONG:  $[0.1, 0.6, 0.4, 0.1]$

CORRECT: same  $x'_4$  as before the swap

# 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)



# Batching: Padding and Truncation

- Transformers can be trained very efficiently!  
(This is arguably one of the key reasons they have been so successful.)
- Batching:** Rather than processing one sentence at a time, Transformers take in a batch of  $B$  sentences at a time. The computation is identical for each batch and is trivially parallelized.

$i$	$w_1$	$w_2$	$w_3$	$w_4$	$w_5$	$w_6$	$w_7$	$w_8$	$w_9$	$w_{10}$	$w_{11}$	$w_{12}$
1	In	the	hole	in	the	ground	there	lived	a	hobbit		
2	It	is	our	choices	that	show	what	we	truly	are		
3	It	was	the	best	of	times	it	was	the	worst	of	times
4	Even	miracles	take	a	little	time						
5	The	more	that	you	read	the	more	things	you	will	know	
6	We'll	always	have	each	other	no	matter	what	happens			
7	The	sun	did	not	shine	it	was	too	wet	to	play	
8	The	important	thing	is	to	never	stop	questioning				

# Batching: Padding and Truncation

- Suppose we have 8 training sentences
- We set our block size (maximum sequence length) to 10
- Before collecting them into a batch, we:
  1. **truncate** those sentences that are too long
  2. **pad** the sentences that are too short
  3. convert each token to an **integer** via a lookup table (vocabulary)
  4. convert each token to an **embedding** vector of fixed length

i	w <sub>1</sub>	w <sub>2</sub>	w <sub>3</sub>	w <sub>4</sub>	w <sub>5</sub>	w <sub>6</sub>	w <sub>7</sub>	w <sub>8</sub>	w <sub>9</sub>	w <sub>10</sub>	w <sub>11</sub>	w <sub>12</sub>
1	In	the	hole	in	the	ground	there	lived	a	hobbit		
2	It	is	our	choices	that	show	what	we	truly	are		
3	It	was	the	best	of	times	it	was	the	worst	of	times
4	Even	miracles	take	a	little	time						
5	The	more	that	you	read	the	more	things	you	will	know	
6	We'll	always	have	each	other	no	matter	what	happens			
7	The	sun	did	not	shine	it	was	too	wet	to	play	
8	The	important	thing	is	to	never	stop	questioning				

# Batching: Padding and Truncation

- Suppose we have 8 training sentences
- We set our block size (maximum sequence length) to 10
- Before collecting them into a batch, we:
  1. **truncate** those sentences that are too long
  2. **pad** the sentences that are too short
  3. convert each token to an **integer** via a lookup table (vocabulary)
  4. convert each token to an **embedding** vector of fixed length

i	w <sub>1</sub>	w <sub>2</sub>	w <sub>3</sub>	w <sub>4</sub>	w <sub>5</sub>	w <sub>6</sub>	w <sub>7</sub>	w <sub>8</sub>	w <sub>9</sub>	w <sub>10</sub>	w <sub>11</sub>	w <sub>12</sub>
1	In	the	hole	in	the	ground	there	lived	a	hobbit		
2	It	is	our	choices	that	show	what	we	truly	are		
3	It	was	the	best	of	times	it	was	the	worst	of	times
4	Even	miracles	take	a	little	time	<PAD>	<PAD>	<PAD>	<PAD>		
5	The	more	that	you	read	the	more	things	you	will	know	
6	We'll	always	have	each	other	no	matter	what	happens	<PAD>		
7	The	sun	did	not	shine	it	was	too	wet	to	play	
8	The	important	thing	is	to	never	stop	questioning	<PAD>	<PAD>		

# Batching: Padding and Truncation

- Suppose we have 8 training sentences
- We set our block size (maximum sequence length) to 10
- Before collecting them into a batch, we:
  1. **truncate** those sentences that are too long
  2. **pad** the sentences that are too short
  3. convert each token to an **integer** via a lookup table (vocabulary)
  4. convert each token to an **embedding** vector of fixed length

i	w <sub>1</sub>	w <sub>2</sub>	w <sub>3</sub>	w <sub>4</sub>	w <sub>5</sub>	w <sub>6</sub>	w <sub>7</sub>	w <sub>8</sub>	w <sub>9</sub>	w <sub>10</sub>
1	2	41	17	19	41	13	42	23	6	16
2	3	20	32	10	40	36	53	51	49	8
3	3	50	41	9	30	46	21	50	41	55
4	1	25	39	6	22	45	0	0	0	0
5	4	26	40	56	34	41	26	44	56	54
6	5	7	15	12	31	28	24	53	14	0
7	4	38	11	29	35	21	50	48	52	47
8	4	18	43	20	47	27	37	33	0	0

## Vocabulary:

```
{  
    '<PAD>': 0,  
    'Even': 1,  
    'In': 2,  
    'It': 3,  
    'The': 4,  
    'We'll': 5,  
    'a': 6,  
    'always': 7,  
    'are': 8,  
    'best': 9,  
    ...  
    'what': 53,  
    'will': 54,  
    'worst': 55,  
    'you': 56  
}
```

# Batching: Padding and Truncation

- Suppose we have 8 training sentences
- We set our block size (maximum sequence length) to 10
- Before collecting them into a batch, we:
  1. **truncate** those sentences that are too long
  2. **pad** the sentences that are too short
  3. convert each token to an **integer** via a lookup table (vocabulary)
  4. convert each token to an **embedding** vector of fixed length

i	w <sub>1</sub>	w <sub>2</sub>	w <sub>3</sub>	w <sub>4</sub>	w <sub>5</sub>	w <sub>6</sub>	w <sub>7</sub>	w <sub>8</sub>	w <sub>9</sub>	w <sub>10</sub>
1										
2										
3										
4										
5										
6										
7										
8										

Embeddings:		
{		
0	:	
1	:	
2	:	
3	:	
4	:	
5	:	
6	:	
7	:	
...		
55	:	
56	:	
}		



# Tokenization

## Subword-based Tokenizer:

Input: “Geoff is givin’ a lectrue on transformers”

Output: [“geoff”, “is”, “giv”, “##in”, “ ‘ ”, “a”, “lec” “##true”, “on”, “transform”, “##ers”]

## Pros/Cons:

- Split long or rare words into smaller, semantically meaningful components or subwords
- No out-of-vocabulary words – any non-subword token can be constructed from other subwords (always includ all characters as subwords)
- Examples algorithms for learning a subword tokenization:
  - Byte-Pair-Encoding (BPE), WordPiece, SentencePiece

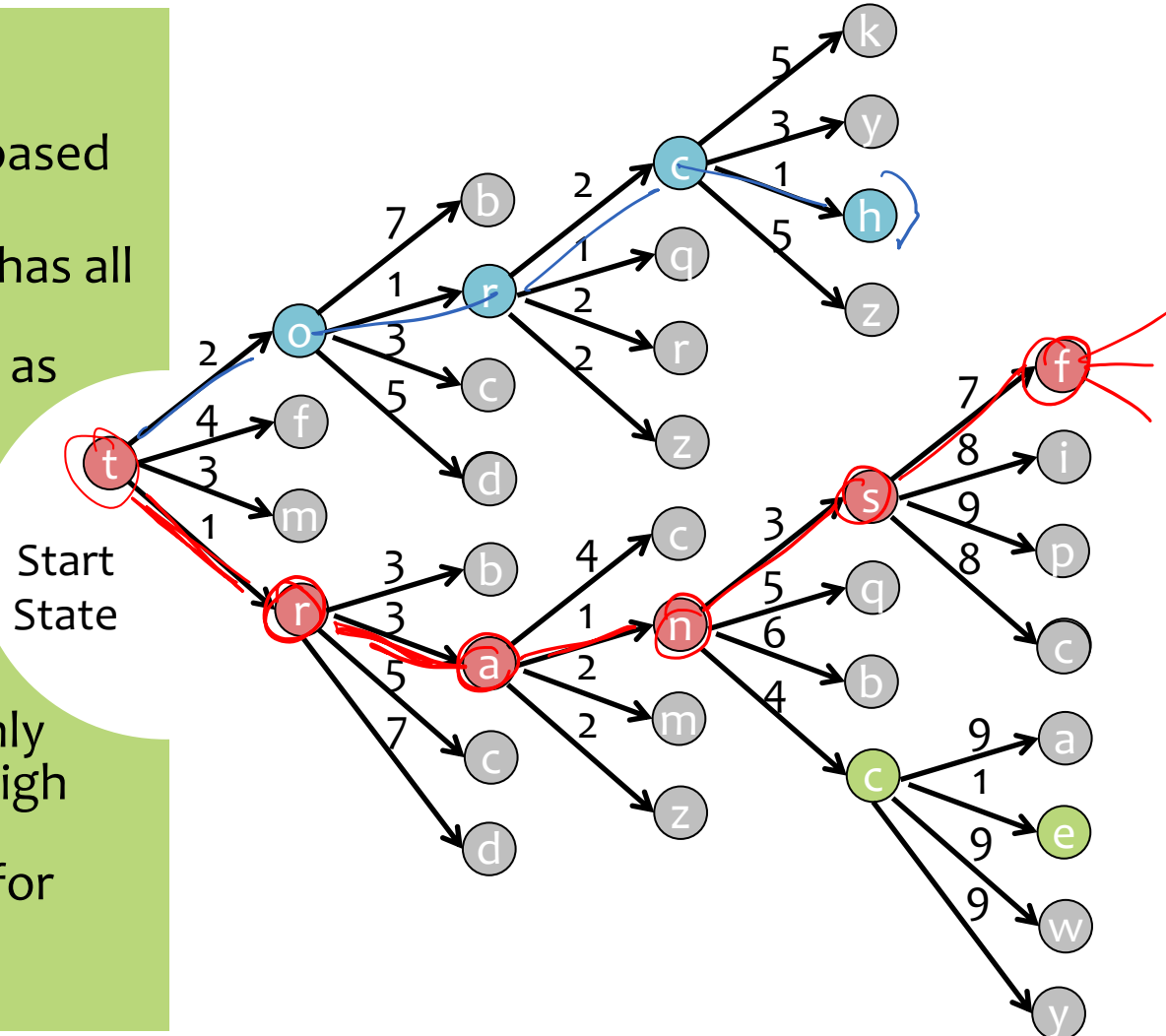
# **GREEDY DECODING FOR A LANGUAGE MODEL**

# Greedy Decoding for a Language Model

## Setup:

- Assume a character-based tokenizer
- Each node has all characters {a,b,c,...,z} as neighbors

- Here we only show the high probability neighbors for space



## Goal:

- Search space consists of nodes (partial sentences) and weighted by negative log probability
- Goal is to find the highest probably (lowest negative log probability) path from root to a leaf

## Greedy Search:

- At each node, selects the edge with lowest negative log probability
- **Heuristic** method of search (i.e. does not necessarily find the best path)
- Computation time: **linear** in max path length

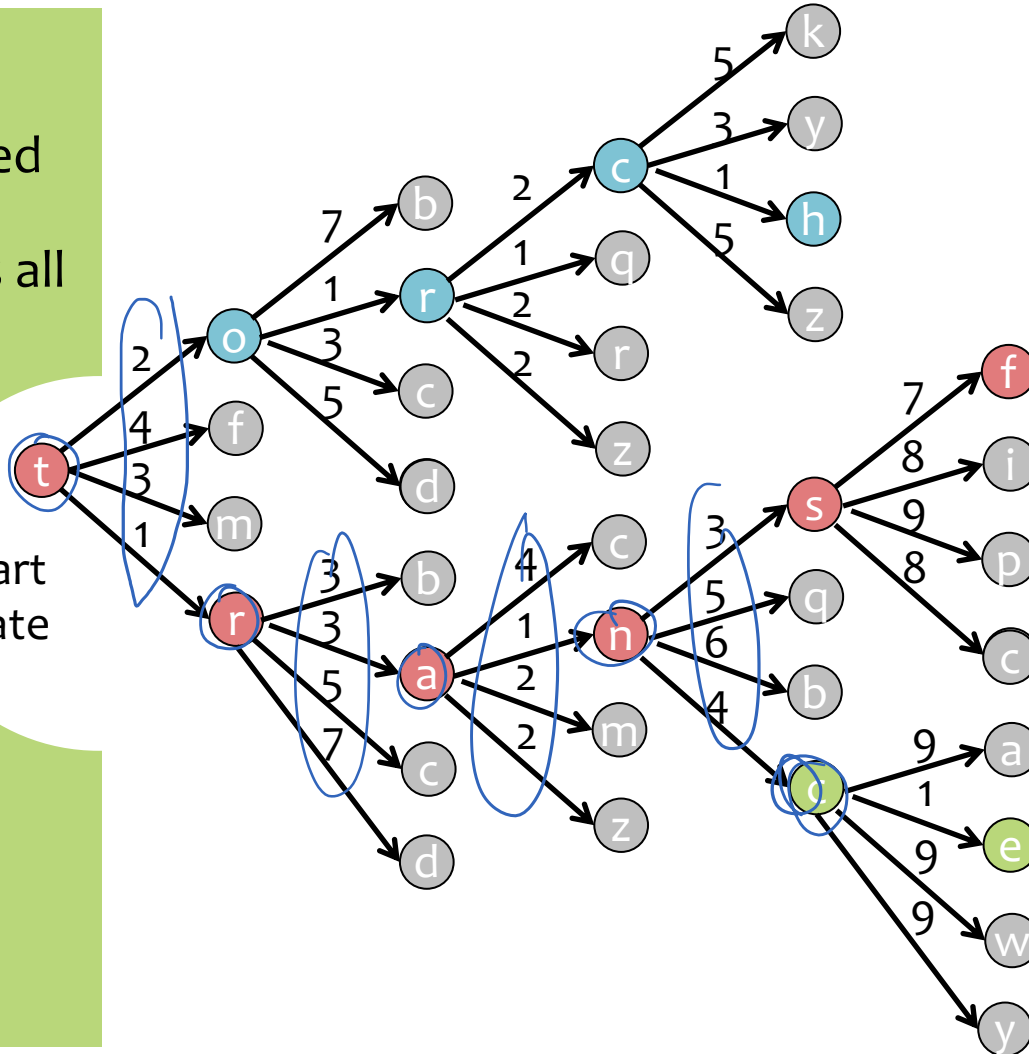
# Sampling from a Language Model

## Setup:

- Assume a character-based tokenizer
- Each node has all characters {a,b,c,...,z} as neighbors

Start State

- Here we only show the high probability neighbors for space



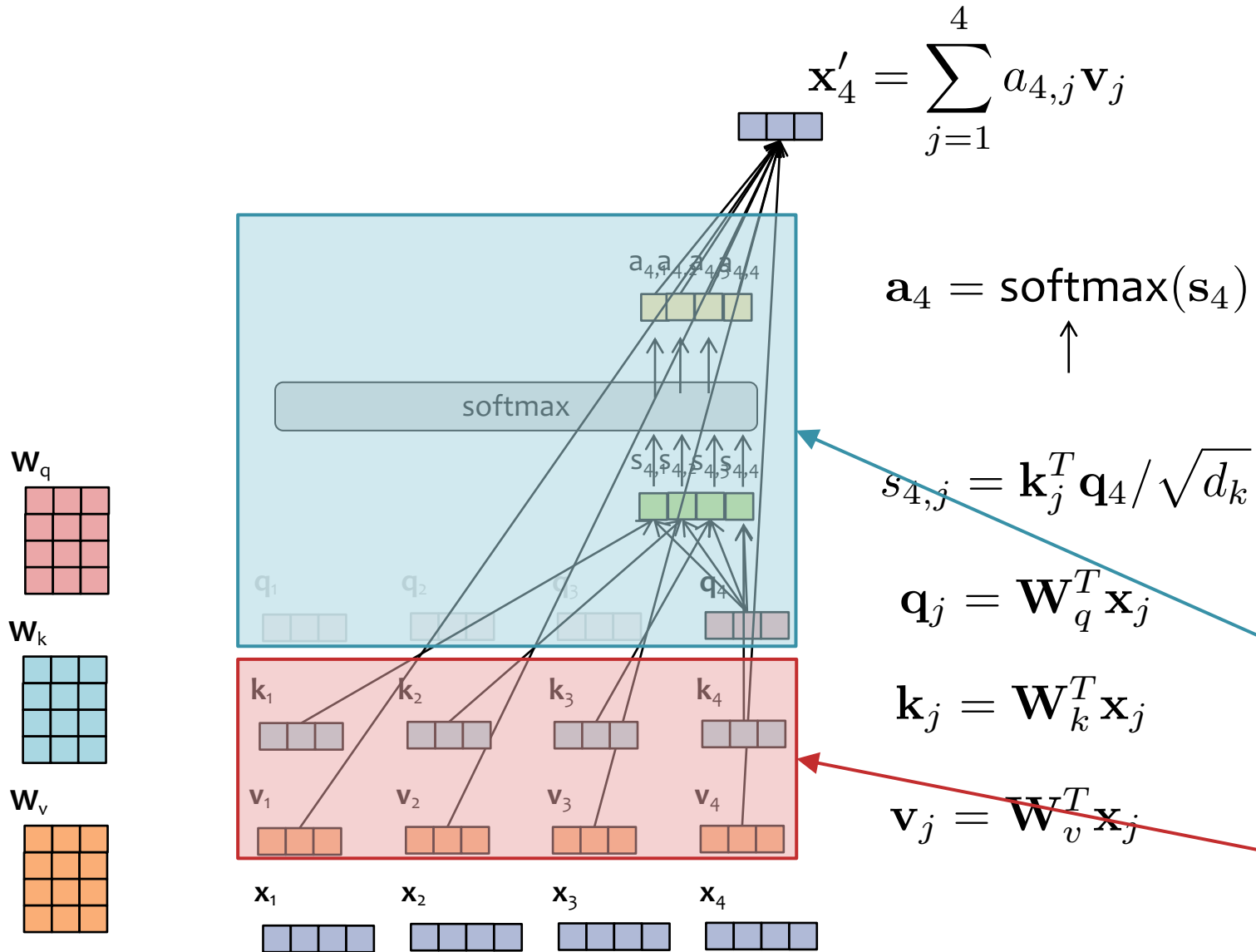
## Goal:

- Search space consists of nodes (partial sentences) and weighted by negative log probability
- Goal is to sample a path from root to a leaf with probability according to the probability of that path

## Ancestral Sampling:

- At each node, randomly pick an edge with probability (converting from negative log probability)
- **Exact** method of sampling, assuming a locally normalized distribution (i.e. samples a path according to its total probability)
- Computation time: **linear** in max path length

# Key-Value Cache



During generation, at each timestep:

- we reuse all previous keys and values (i.e. we need to cache them)
- but we can get rid of the queries, similarity scores, and attention weights

Discarded after this timestep

The KV Cache: each time we compute the new  $k_j$  and  $v_j$  for this timestep and cache them for subsequent timesteps

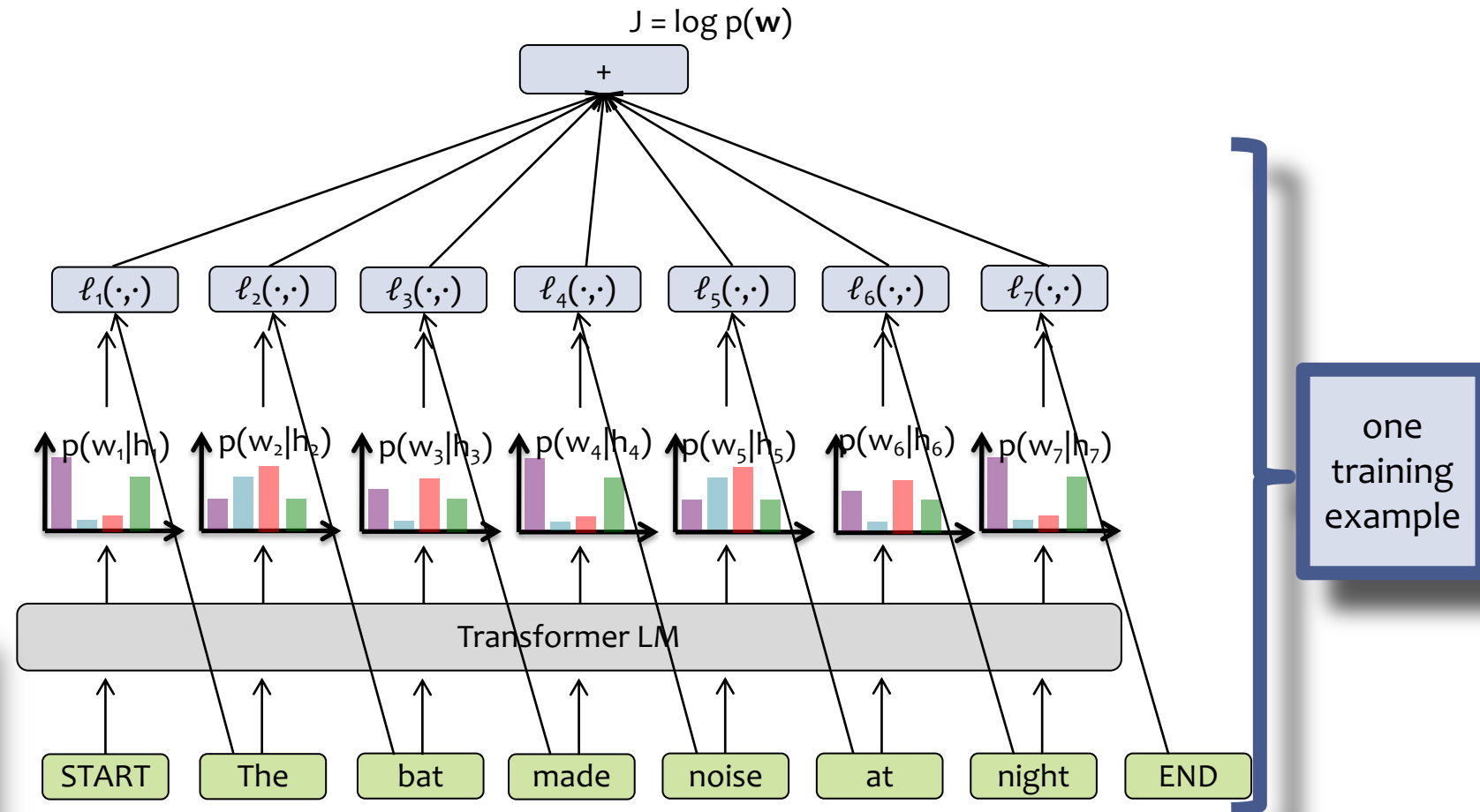
# **LEARNING A TRANSFORMER LM**

# Learning a Transformer LM

- Each training example is a sequence (e.g. sentence), so we have training data  $D = \{\mathbf{w}^{(1)}, \mathbf{w}^{(2)}, \dots, \mathbf{w}^{(N)}\}$
- The objective function for a Deep LM (e.g. RNN-LM or Transformer-LM) is typically the log-likelihood of the training examples:  
 $J(\theta) = \sum_i \log p_{\theta}(\mathbf{w}^{(i)})$
- We train by mini-batch SGD (or your favorite flavor of mini-batch SGD)

Training a Transformer-LM is the same, except we swap in a different deep language model.

$$\begin{aligned}\log p(\mathbf{w}) &= \log p(w_1, w_2, w_3, \dots, w_T) \\ &= \log p(w_1 | h_1) + \log p(w_2 | h_2) + \dots + \log p(w_T | h_T)\end{aligned}$$



# Language Models over Time

## An aside:

- State-of-the-art language models currently tend to rely on **transformer networks** (e.g. GPT-3)
- RNN-LMs comprised most of the early neural LMs that **led to** current SOTA architectures

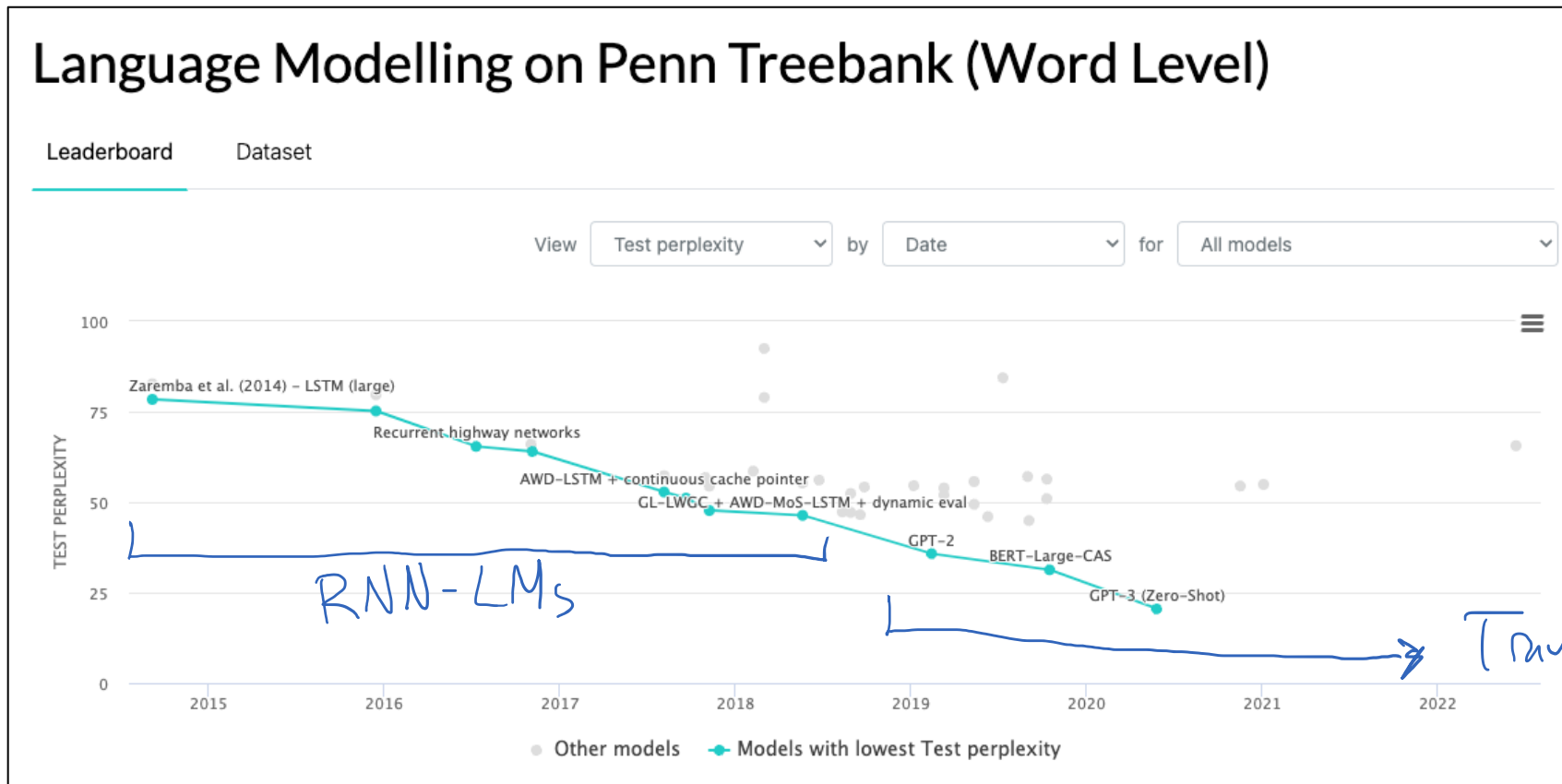


Figure from <https://paperswithcode.com/sota/language-modelling-on-penn-treebank-word>



# How big are LLMs?

- GPT stands for Generative Pre-trained Transformer
- An large language model (LLM) with a GPT architecture 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	3072	12	117M
GPT-2 (2019)	48	1600	--	--	1542M
GPT-3 (2020)	96	12288	4*12288	96	175000M

# Why does efficiency matter?

## Case Study: GPT-3

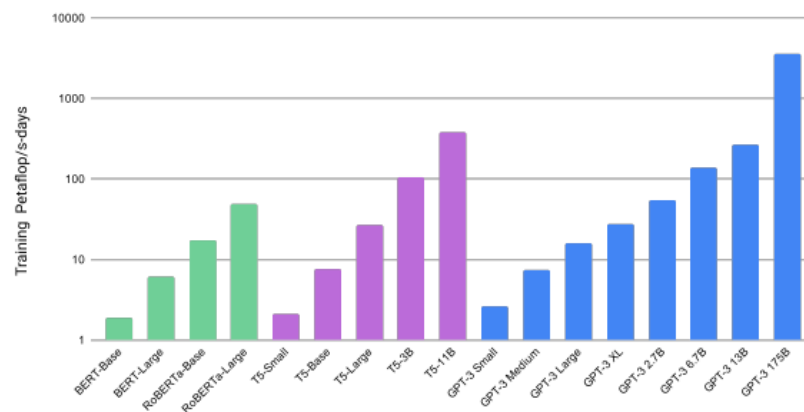
- # of training tokens = 500 billion
- # of parameters = 175 billion
- # of cycles = 50 petaflop/s-days (each of which are  $8.64 \times 10^{19}$  flops)

Dataset	Quantity (tokens)	Weight in training mix	Epochs elapsed when training for 300B tokens
Common Crawl (filtered)	410 billion	60%	0.44
WebText2	19 billion	22%	2.9
Books1	12 billion	8%	1.9
Books2	55 billion	8%	0.43
Wikipedia	3 billion	3%	3.4

**Table 2.2: Datasets used to train GPT-3.** “Weight in training mix” refers to the fraction of examples during training that are drawn from a given dataset, which we intentionally do not make proportional to the size of the dataset. As a result, when we train for 300 billion tokens, some datasets are seen up to 3.4 times during training while other datasets are seen less than once.

Model Name	$n_{\text{params}}$	$n_{\text{layers}}$	$d_{\text{model}}$	$n_{\text{heads}}$	$d_{\text{head}}$	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	$6.0 \times 10^{-4}$
GPT-3 Medium	350M	24	1024	16	64	0.5M	$3.0 \times 10^{-4}$
GPT-3 Large	760M	24	1536	16	96	0.5M	$2.5 \times 10^{-4}$
GPT-3 XL	1.3B	24	2048	24	128	1M	$2.0 \times 10^{-4}$
GPT-3 2.7B	2.7B	32	2560	32	80	1M	$1.6 \times 10^{-4}$
GPT-3 6.7B	6.7B	32	4096	32	128	2M	$1.2 \times 10^{-4}$
GPT-3 13B	13.0B	40	5140	40	128	2M	$1.0 \times 10^{-4}$
GPT-3 175B or “GPT-3”	175.0B	96	12288	96	128	3.2M	$0.6 \times 10^{-4}$

**Table 2.1: Sizes, architectures, and learning hyper-parameters (batch size in tokens and learning rate) of the models which we trained.** All models were trained for a total of 300 billion tokens.



**Figure 2.2: Total compute used during training.** Based on the analysis in Scaling Laws For Neural Language Models [KMH<sup>+</sup>20] we train much larger models on many fewer tokens than is typical. As a consequence, although GPT-3 3B is almost 10x larger than RoBERTa-Large (355M params), both models took roughly 50 petaflop/s-days of compute during pre-training. Methodology for these calculations can be found in Appendix D.

# Recap

Two parts: **Deep Learning** and **Language Modeling**

## 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)
  - we are considering various (deep) computation graphs: (1) CNN (2) RNN (3) RNN-LM (4) Transformer-LM
- Learning a Deep Network
  - deep networks (e.g. CNN/RNN) are trained by optimizing an objective function with SGD
  - compute gradients with AutoDiff

## 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!
  - a RNN-LM / Transformer-LM is trained just like other deep neural networks

# **MODULE-BASED AUTOMATIC DIFFERENTIATION**

## Automatic Differentiation – Reverse Mode (aka. Backpropagation)

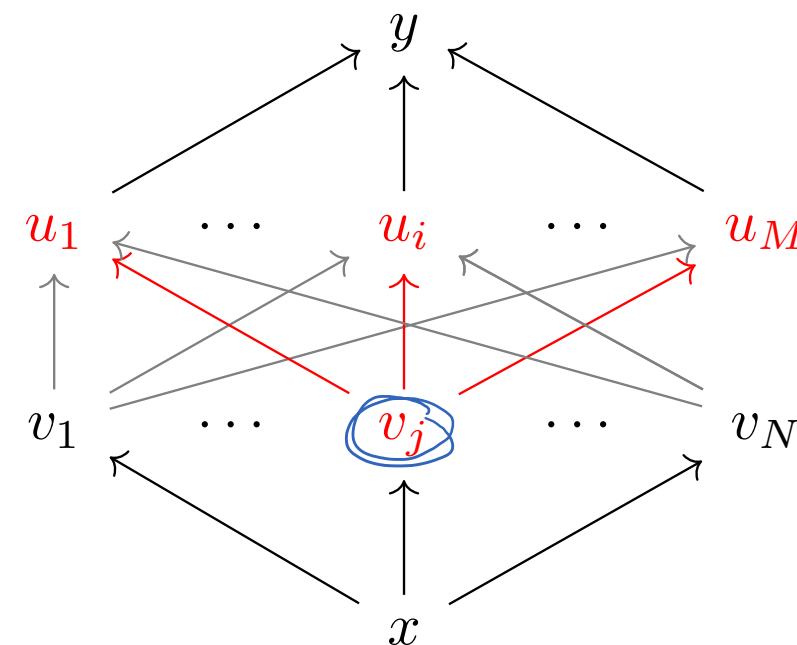
Forward Computation

1. Write an **algorithm** for evaluating the function  $y = f(\mathbf{x})$ . The algorithm defines a **directed acyclic graph**, where each variable is a node (i.e. the “**computation graph**”)
2. Visit each node in **topological order**.  
For variable  $u_i$  with inputs  $v_1, \dots, v_N$ 
  - a. Compute  $u_i = g_i(v_1, \dots, v_N)$
  - b. Store the result at the node

Backward Computation (Version A)

1. **Initialize**  $dy/dy = 1$ .
2. Visit each node  $v_j$  in **reverse topological order**.  
Let  $u_1, \dots, u_M$  denote all the nodes with  $v_j$  as an input  
Assuming that  $y = h(\mathbf{u}) = h(u_1, \dots, u_M)$   
and  $\mathbf{u} = g(\mathbf{v})$  or equivalently  $u_i = g_i(v_1, \dots, v_j, \dots, v_N)$  for all  $i$ 
  - a. We already know  $dy/du_i$  for all  $i$
  - b. Compute  $dy/dv_j$  as below (Choice of algorithm ensures computing  $(du_i/dv_j)$  is easy)

$$\frac{dy}{dv_j} = \sum_{i=1}^M \frac{dy}{du_i} \frac{du_i}{dv_j}$$

Return partial derivatives  $dy/du_i$  for all variables

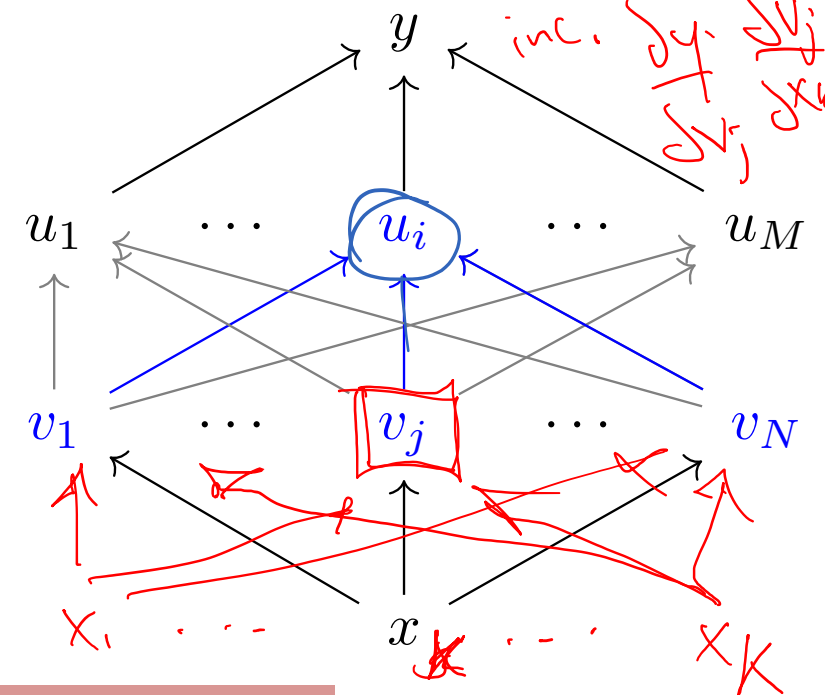
## Automatic Differentiation – Reverse Mode (aka. Backpropagation)

Forward Computation

1. Write an **algorithm** for evaluating the function  $y = f(x)$ . The algorithm defines a **directed acyclic graph**, where each variable is a node (i.e. the “**computation graph**”)
2. Visit each node in **topological order**.  
For variable  $u_i$  with inputs  $v_1, \dots, v_N$ 
  - a. Compute  $u_i = g_i(v_1, \dots, v_N)$
  - b. Store the result at the node

Backward Computation (Version B)

1. **Initialize** all partial derivatives  $dy/du_i$  to 0 and  $dy/dy = 1$ .
2. Visit each node in **reverse topological order**.  
For variable  $u_i = g_i(v_1, \dots, v_N)$ 
  - a. We already know  $dy/du_i$
  - b. Increment  $dy/dv_j$  by  $(dy/du_i)(du_i/dv_j)$   
(Choice of algorithm ensures computing  $(du_i/dv_j)$  is easy)

Return partial derivatives  $dy/du_i$  for all variables

# Backpropagation: Procedural Method

## Algorithm 1 Forward Computation

```
1: procedure NNFORWARD(Training example  $(x, y)$ , Params  $\alpha, \beta$ )
2:    $\mathbf{a} = \alpha \mathbf{x}$ 
3:    $\mathbf{z} = \sigma(\mathbf{a})$ 
4:    $\mathbf{b} = \beta \mathbf{z}$ 
5:    $\hat{\mathbf{y}} = \text{softmax}(\mathbf{b})$ 
6:    $J = -\mathbf{y}^T \log \hat{\mathbf{y}}$ 
7:    $\mathbf{o} = \text{object}(\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J)$ 
8:   return intermediate quantities  $\mathbf{o}$ 
```

## Algorithm 2 Backpropagation

```
1: procedure NNBACKWARD(Training example  $(x, y)$ , Params  $\alpha, \beta$ ,
  Intermediates  $\mathbf{o}$ )
2:   Place intermediate quantities  $\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J$  in  $\mathbf{o}$  in scope
3:    $\mathbf{g}_{\hat{\mathbf{y}}} = -\mathbf{y} \div \hat{\mathbf{y}}$ 
4:    $\mathbf{g}_{\mathbf{b}} = \mathbf{g}_{\hat{\mathbf{y}}}^T (\text{diag}(\hat{\mathbf{y}}) - \hat{\mathbf{y}} \hat{\mathbf{y}}^T)$ 
5:    $\mathbf{g}_{\beta} = \mathbf{g}_{\mathbf{b}}^T \mathbf{z}^T$ 
6:    $\mathbf{g}_{\mathbf{z}} = \beta^T \mathbf{g}_{\mathbf{b}}^T$ 
7:    $\mathbf{g}_{\mathbf{a}} = \mathbf{g}_{\mathbf{z}} \odot \mathbf{z} \odot (1 - \mathbf{z})$ 
8:    $\mathbf{g}_{\alpha} = \mathbf{g}_{\mathbf{a}} \mathbf{x}^T$ 
9:   return parameter gradients  $\mathbf{g}_{\alpha}, \mathbf{g}_{\beta}$ 
```

## Drawbacks of Procedural Method

1. Hard to reuse / adapt for other models
2. (Possibly) harder to make individual steps more efficient
3. Hard to find source of error if finite-difference check reports an error (since it tells you only that there is an error somewhere in those 17 lines of code)

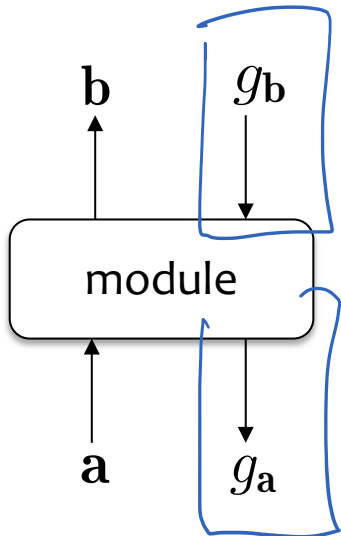
# Module-based AutoDiff

- **Key Idea:**
  - componentize the computation of the neural-network into layers
  - each layer consolidates multiple **real-valued nodes** in the computation graph (a subset of them) into one **vector-valued node** (aka. a **module**)
- Each **module** is capable of two actions:

1. Forward computation of output  $\mathbf{b} = [b_1, \dots, b_B]$  given input  $\mathbf{a} = [a_1, \dots, a_A]$  via some differentiable function  $f$ . That is

$$\mathbf{b} = f(\mathbf{a}).$$

2. Backward computation of the gradient of the input  $\mathbf{g}_a = \nabla_{\mathbf{a}} J = [\frac{\partial J}{\partial a_1}, \dots, \frac{\partial J}{\partial a_A}]$  given the gradient of output  $\mathbf{g}_b = \nabla_{\mathbf{b}} J = [\frac{\partial J}{\partial b_1}, \dots, \frac{\partial J}{\partial b_B}]$ , where  $J$  is the final real-valued output of the entire computation graph. This is done via the chain rule  $\frac{\partial J}{\partial a_i} = \sum_{j=1}^B \frac{\partial J}{\partial b_j} \frac{db_j}{da_i}$  for all  $i \in \{1, \dots, A\}$ .




$$\mathbf{g}_a = \frac{\partial J}{\partial \mathbf{a}}$$
$$\mathbf{g}_b = \frac{\partial J}{\partial \mathbf{b}}$$



# Module-based AutoDiff (oOP Version)


Object-Oriented Implementation:

- Let each module be an **object**
- Then allow the **control flow** dictate the creation of the **computation graph**
- No longer need to implement NNBackward( $\cdot$ ), just follow the computation graph in **reverse topological order**



```
1 class Sigmoid(Module)
2     method forward(a)
3          $b = \sigma(a)$ 
4         return b
5     method backward(a, b, gb)
6          $g_a = g_b \odot b \odot (1 - b)$ 
7         return ga
```

```
1 class Softmax(Module)
2     method forward(a)
3          $b = \text{softmax}(a)$ 
4         return b
5     method backward(a, b, gb)
6          $g_a = g_b^T (\text{diag}(b) - bb^T)$ 
7         return ga
```



```
1 class Linear(Module)
2     method forward(a,  $\omega$ )
3          $b = \omega a$ 
4         return b
5     method backward(a,  $\omega$ , b, gb)
6          $g_\omega = g_b a^T$ 
7          $g_a = \omega^T g_b$ 
8         return g $\omega$ , ga
```

```
1 class CrossEntropy(Module)
2     method forward(a,  $\hat{a}$ )
3          $b = -a^T \log \hat{a}$ 
4         return b
5     method backward(a,  $\hat{a}$ , b, gb)
6          $g_{\hat{a}} = -g_b (a \div \hat{a})$ 
7         return ga
```

# Module-based AutoDiff (oOP Version)

```
1 class NeuralNetwork(Module):
2
3     method init()
4         lin1_layer = Linear()
5         sig_layer = Sigmoid()
6         lin2_layer = Linear()
7         soft_layer = Softmax()
8         ce_layer = CrossEntropy()
9
10    method forward(Tensor x, Tensor y, Tensor  $\alpha$ , Tensor  $\beta$ )
11        a = lin1_layer.apply_fwd(x,  $\alpha$ )
12        z = sig_layer.apply_fwd(a)
13        b = lin2_layer.apply_fwd(z,  $\beta$ )
14         $\hat{y}$  = soft_layer.apply_fwd(b)
15        J = ce_layer.apply_fwd(y,  $\hat{y}$ )
16        return J.out_tensor
17
18    method backward(Tensor x, Tensor y, Tensor  $\alpha$ , Tensor  $\beta$ )
19        tape_bwd()
20        return lin1_layer.in_gradients[1], lin2_layer.in_gradients[1]
```

`global tape = stack()` = `[lin1, sig, lin2, soft, ce]`

```
1 global tape = stack()
2
3 class Module:
4
5     method init()
6         out_tensor = null
7         out_gradient = Tensor(1)
8
9     method apply_fwd(List in_modules)
10        in_tensors = [x.out_tensor for x in in_modules]
11        out_tensor = forward(in_tensors)
12        tape.push(self)
13        return self
14
15    method apply_bwd():
16        in_gradients = backward(in_tensors, out_tensor, out_gradient)
17        for i in 1, ..., len(in_modules):
18            in_modules[i].out_gradient += in_gradients[i]
19        return self
20
21    function tape_bwd():
22        while len(tape) > 0
23            m = tape.pop()
24            m.apply_bwd()
```

# Module-based AutoDiff (oOP Version)

```
1 class NeuralNetwork(Module):
2
3     method init()
4         lin1_layer = Linear()
5         sig_layer = Sigmoid()
6         lin2_layer = Linear()
7         soft_layer = Softmax()
8         ce_layer = CrossEntropy()
9
10    method forward(Tensor  $\mathbf{x}$ , Tensor  $\mathbf{y}$ , Tensor  $\alpha$ , Tensor  $\beta$ )
11         $\mathbf{a}$  = lin1_layer.apply_fwd( $\mathbf{x}$ ,  $\alpha$ )
12         $\mathbf{z}$  = sig_layer.apply_fwd( $\mathbf{a}$ )
13         $\mathbf{b}$  = lin2_layer.apply_fwd( $\mathbf{z}$ ,  $\beta$ )
14         $\hat{\mathbf{y}}$  = soft_layer.apply_fwd( $\mathbf{b}$ )
15         $J$  = ce_layer.apply_fwd( $\mathbf{y}$ ,  $\hat{\mathbf{y}}$ )
16        return  $J$ .out_tensor
17
18    method backward(Tensor  $\mathbf{x}$ , Tensor  $\mathbf{y}$ , Tensor  $\alpha$ , Tensor  $\beta$ )
19        tape_bwd()
20        return lin1_layer.in_gradients[1], lin2_layer.in_gradients[1]
```

```
1 global tape = stack()
2
3 class Module:
4
5     method init()
6         out_tensor = null
7         out_gradient = Tensor(1)
8
9     method apply_fwd(List in_modules)
10        in_tensors = [x.out_tensor for x in in_modules]
11        out_tensor = forward(in_tensors)
12        tape.push(self)
13        return self
14
15    method apply_bwd():
16        in_gradients = backward(in_tensors, out_tensor, out_gradient)
17        for i in 1, ..., len(in_modules):
18            in_modules[i].out_gradient += in_gradients[i]
19        return self
20
21    function tape_bwd():
22        while len(tape) > 0
23            m = tape.pop()
24            m.apply_bwd()
```

# PyTorch

The same simple neural network we defined in pseudocode can also be defined in PyTorch.

```
1 # Define model
2 class NeuralNetwork(nn.Module):
3     def __init__(self):
4         super(NeuralNetwork, self).__init__()
5         self.flatten = nn.Flatten()
6         self.linear1 = nn.Linear(28*28, 512)
7         self.sigmoid = nn.Sigmoid()
8         self.linear2 = nn.Linear(512, 512)
9
10    def forward(self, x):
11        x = self.flatten(x)
12        a = self.linear1(x)
13        z = self.sigmoid(a)
14        b = self.linear2(z)
15        return b
16
17 # Take one step of SGD
18 def one_step_of_sgd(X, y):
19     model = NeuralNetwork().to(device)
20     loss_fn = nn.CrossEntropyLoss()
21     optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
22
23     # Compute prediction error
24     pred = model(X)
25     loss = loss_fn(pred, y)
26
27     # Backpropagation
28     optimizer.zero_grad()
29     loss.backward()
30     optimizer.step()
```

# PyTorch

**Q:** Why don't we call `linear.forward()` in PyTorch?

**A:** This is just syntactic sugar. There's a special method in Python `__call__` that allows you to define what happens when you treat an object as if it were a function.

In other words, running the following:

```
linear(x)
```

is equivalent to running:

```
linear.__call__(x)
```

which in PyTorch is (nearly) the same as running:

```
linear.forward(x)
```

This is because PyTorch defines every Module's `__call__` method to be something like this:

```
def __call__(self):  
    self.forward()
```

# PyTorch

**Q:** Why don't we pass in the parameters to a PyTorch Module?

**A:** This just makes your code cleaner.

In PyTorch, you store the parameters inside the Module and “mark” them as parameters that should contribute to the eventual gradient used by an optimizer

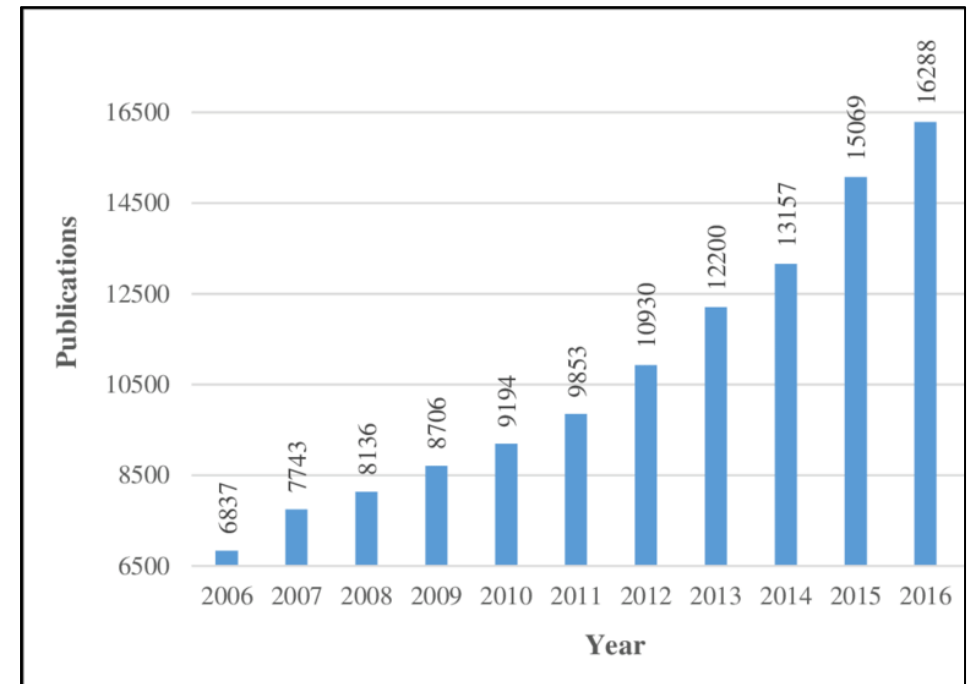
```
0  method forward(Tensor x , Tensor y , Tensor  $\alpha$  , Tensor  $\beta$ )
11      a =lin1_layer.apply_fwd(x,  $\alpha$ )
12      z =sig_layer.apply_fwd(a)
13      b =lin1_layer.apply_fwd(z,  $\beta$ )
14       $\hat{y}$  =soft_layer.apply_fwd(b)
15      J =ce_layer.apply_fwd(y,  $\hat{y}$ )
16      return J.out_tensor
```

```
7
10  def forward(self, x):
11      x = self.flatten(x)
12      a = self.linear1(x)
13      z = self.sigmoid(a)
14      b = self.linear2(z)
15      return b
```

# **PRE-TRAINING VS. FINE-TUNING**

# The Start of Deep Learning

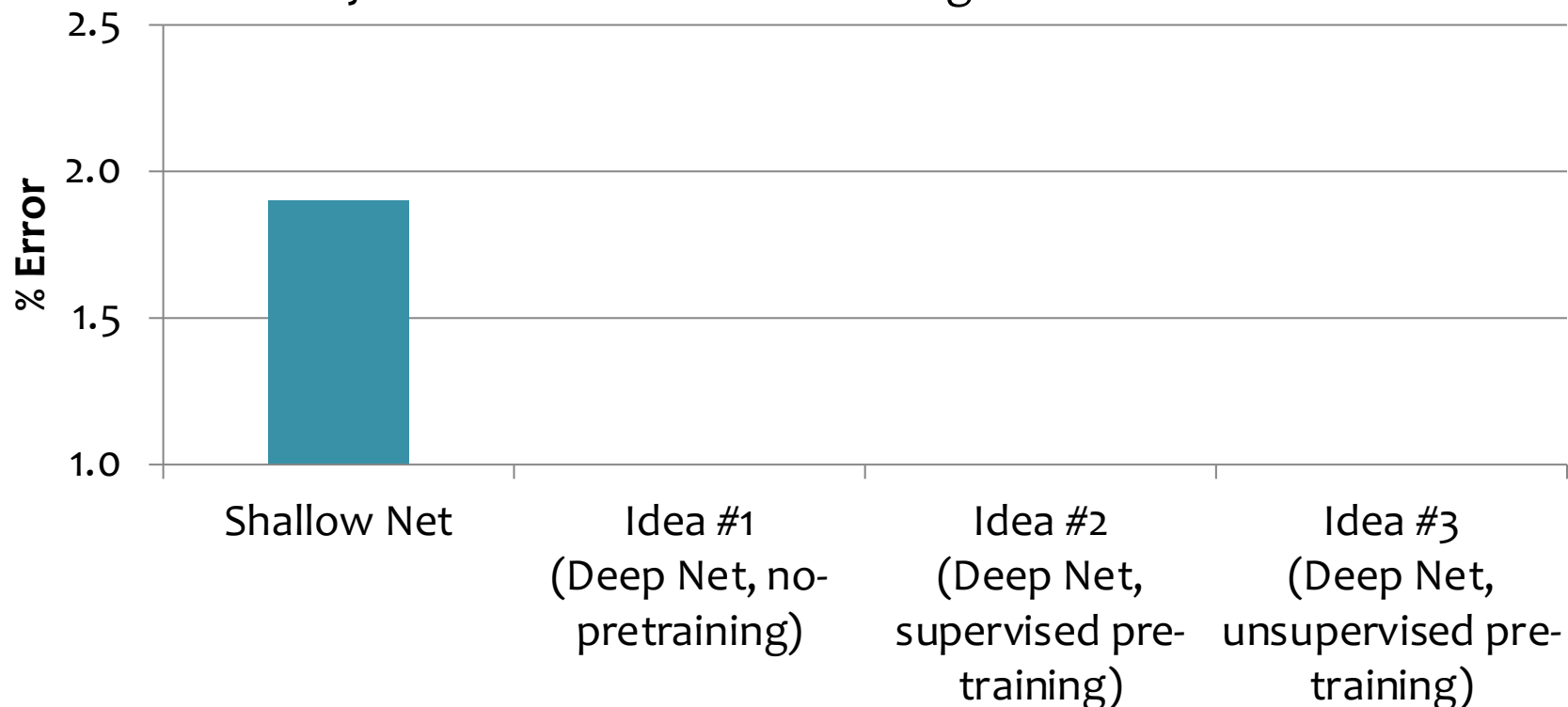
- The architectures of modern deep learning have a long history:
  - 1960s: Rosenblatt's 3-layer multi-layer perceptron, ReLU )
  - 1970-80s: RNNs and CNNs
  - 1990s: linearized self-attention
- The spark for deep learning came in 2006 thanks to **pre-training** (e.g., Hinton & Salakhutdinov, 2006)





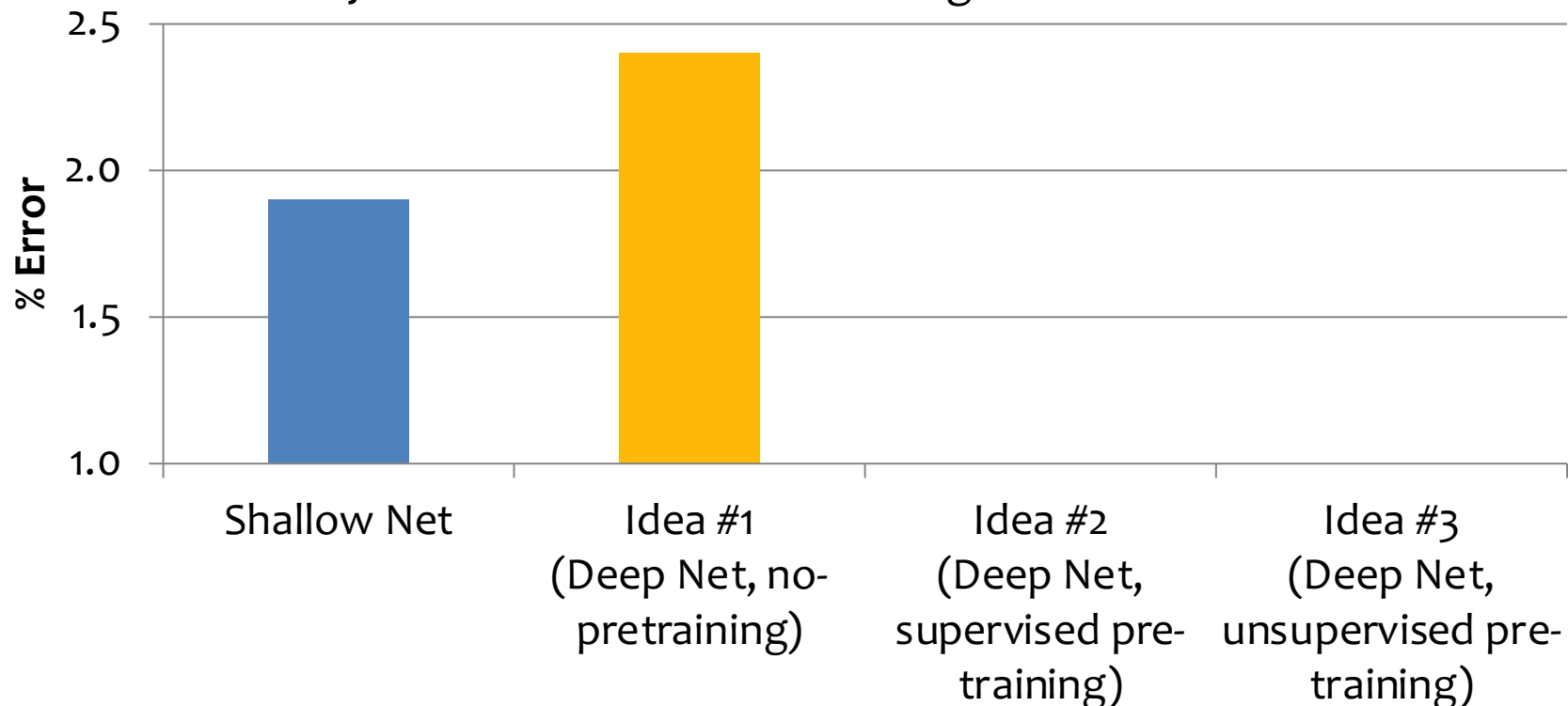
# Pre-Training and Fine-Tuning on MNIST

- Results from Bengio et al. (2006) on MNIST digit classification task
- Percent error (lower is better)
- Some methods first do pre-training
- Every method includes fine-tuning on labeled data



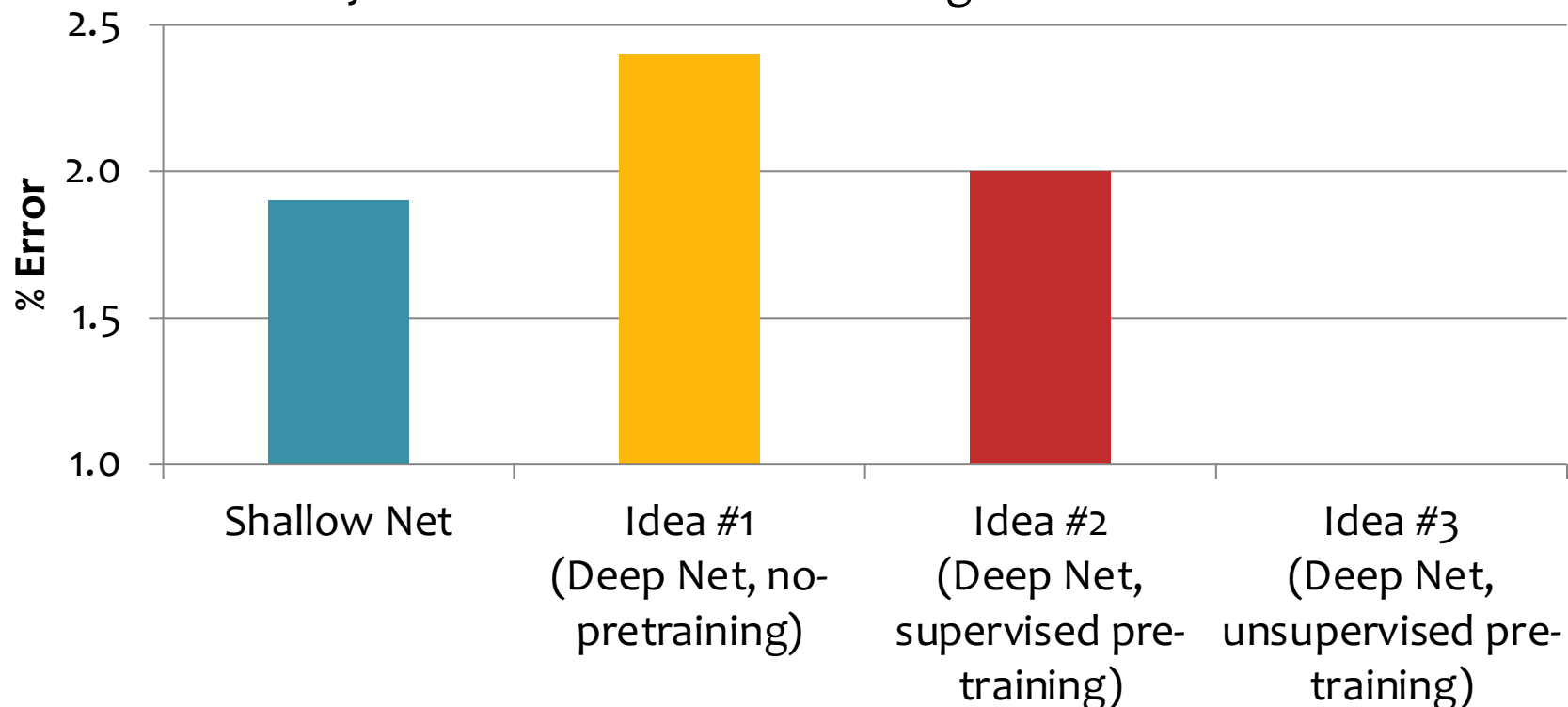
# Pre-Training and Fine-Tuning on MNIST

- Results from Bengio et al. (2006) on MNIST digit classification task
- Percent error (lower is better)
- Some methods first do pre-training
- Every method includes fine-tuning on labeled data



# Pre-Training and Fine-Tuning on MNIST

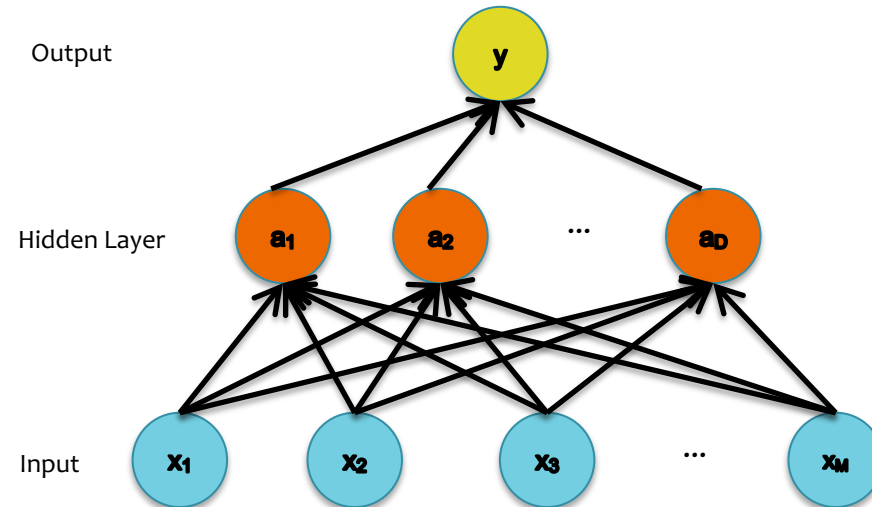
- Results from Bengio et al. (2006) on MNIST digit classification task
- Percent error (lower is better)
- Some methods first do pre-training
- Every method includes fine-tuning on labeled data



# Unsupervised Autoencoder Pre-Training for Vision

## Unsupervised pre-training of the first layer:

- What should it predict?
- What else do we observe?
- **The input!**

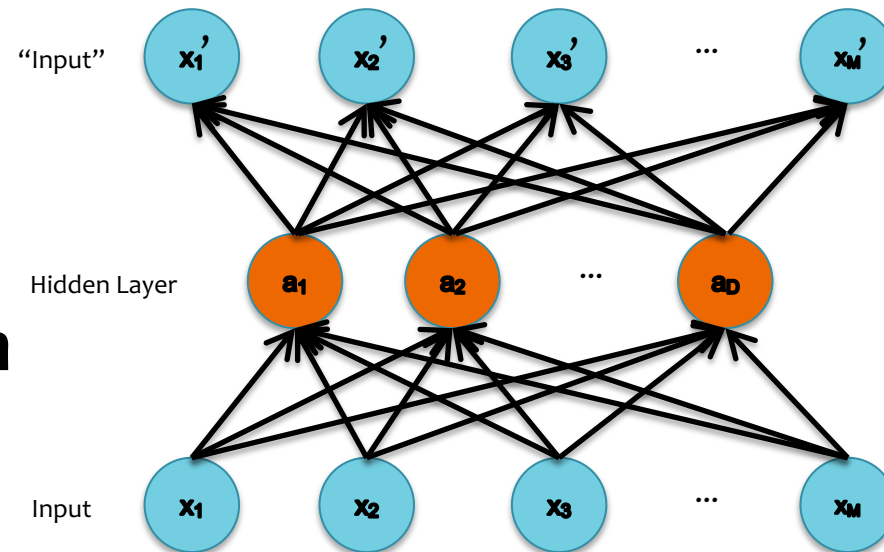


# Unsupervised Autoencoder Pre-Training for Vision

## Unsupervised pre-training of the first layer:

- What should it predict?
- What else do we observe?
- **The input!**

**This topology defines an Auto-encoder.**



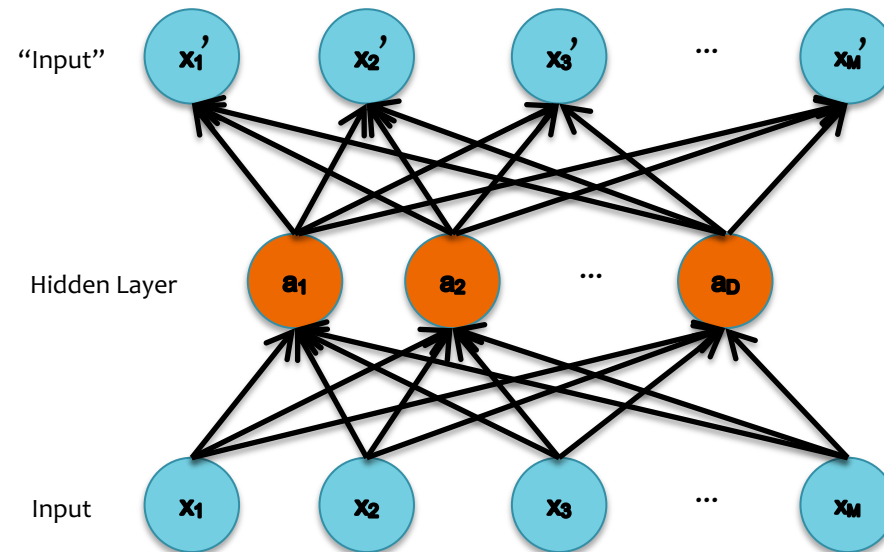
# Unsupervised Autoencoder Pre-Training for Vision

Key idea: Encourage  $z$  to give small reconstruction error:

- $x'$  is the *reconstruction* of  $x$
- $\text{Loss} = ||x - \text{DECODER}(\text{ENCODER}(x))||^2$
- Train with the same backpropagation algorithm for 2-layer Neural Networks with  $x_m$  as both input and output.

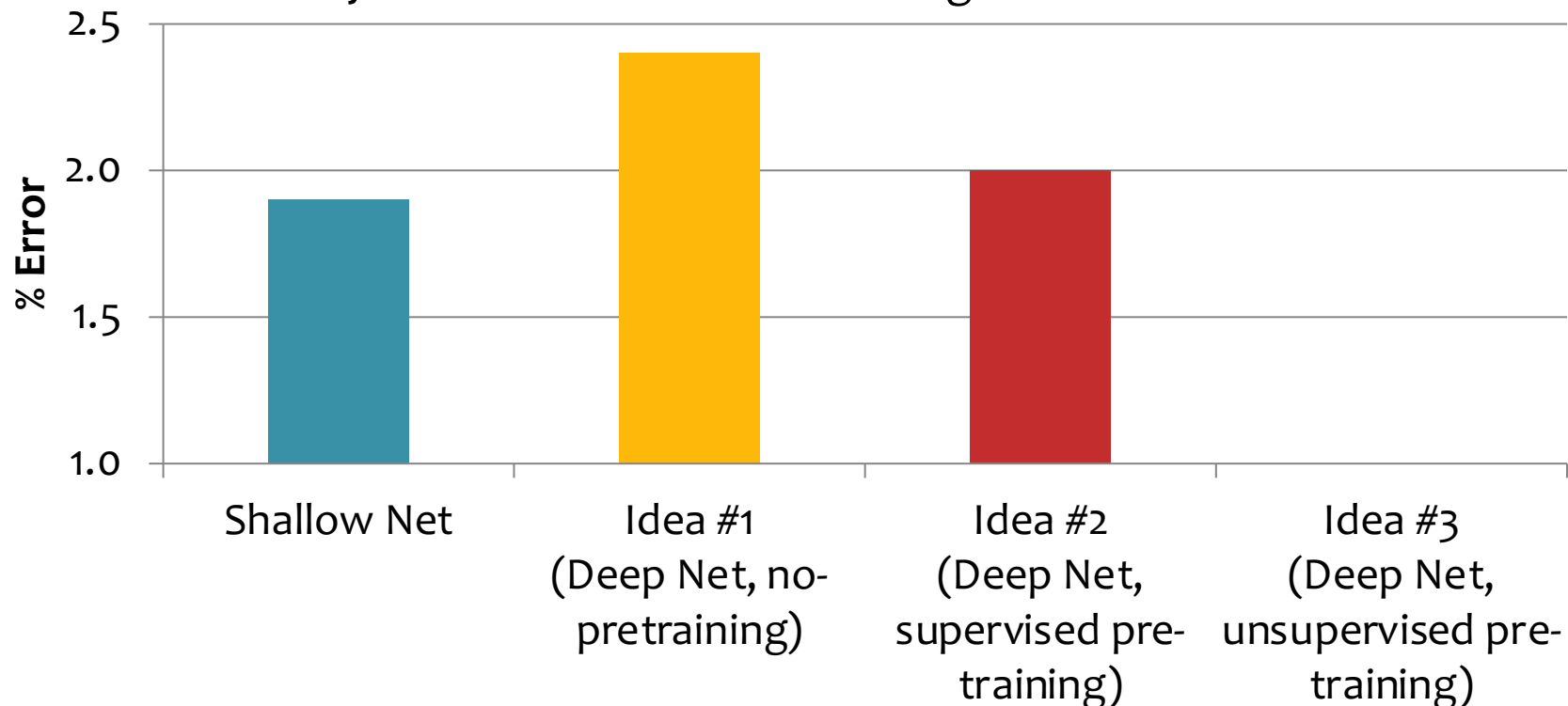
DECODER:  $x' = h(W'z)$

ENCODER:  $z = h(Wx)$



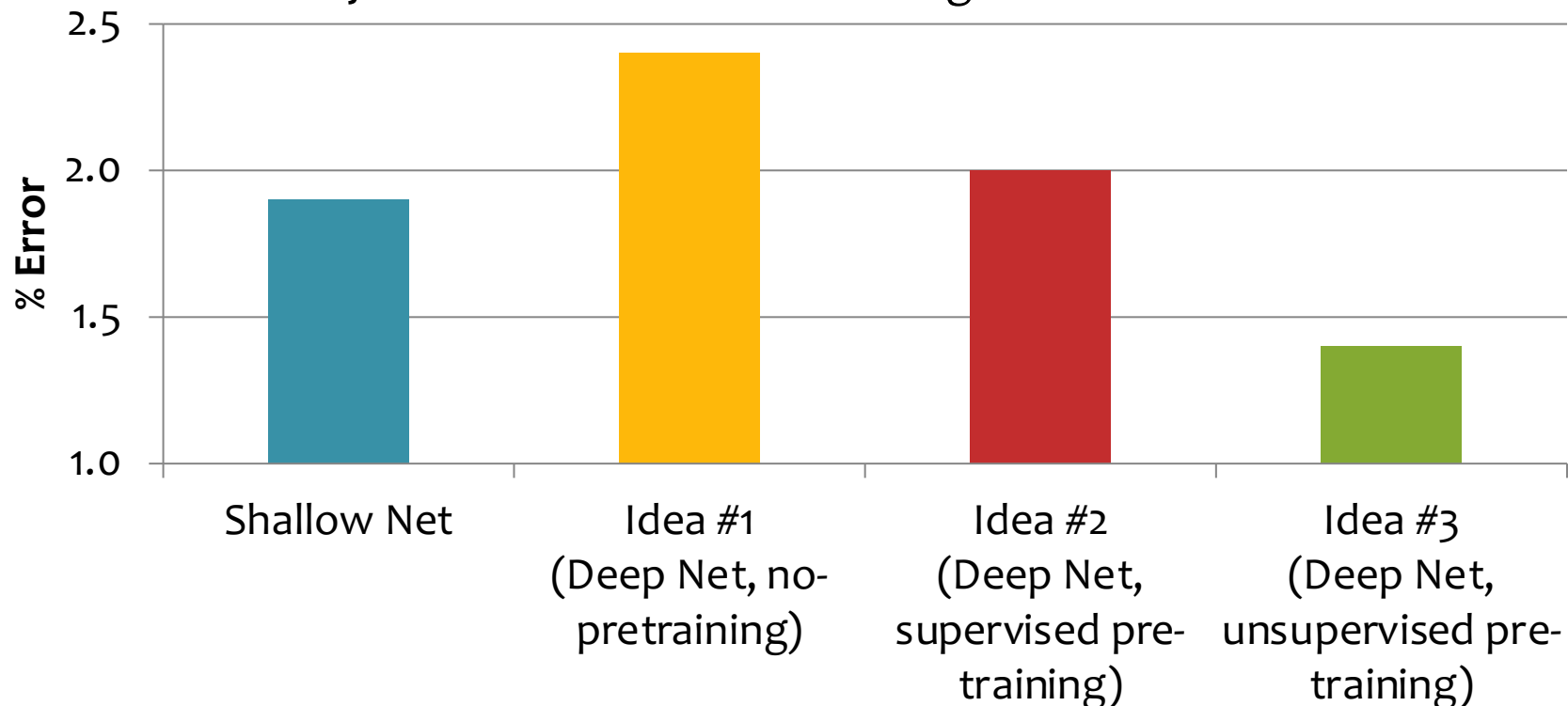
# Pre-Training and Fine-Tuning on MNIST

- Results from Bengio et al. (2006) on MNIST digit classification task
- Percent error (lower is better)
- Some methods first do pre-training
- Every method includes fine-tuning on labeled data



# Pre-Training and Fine-Tuning on MNIST

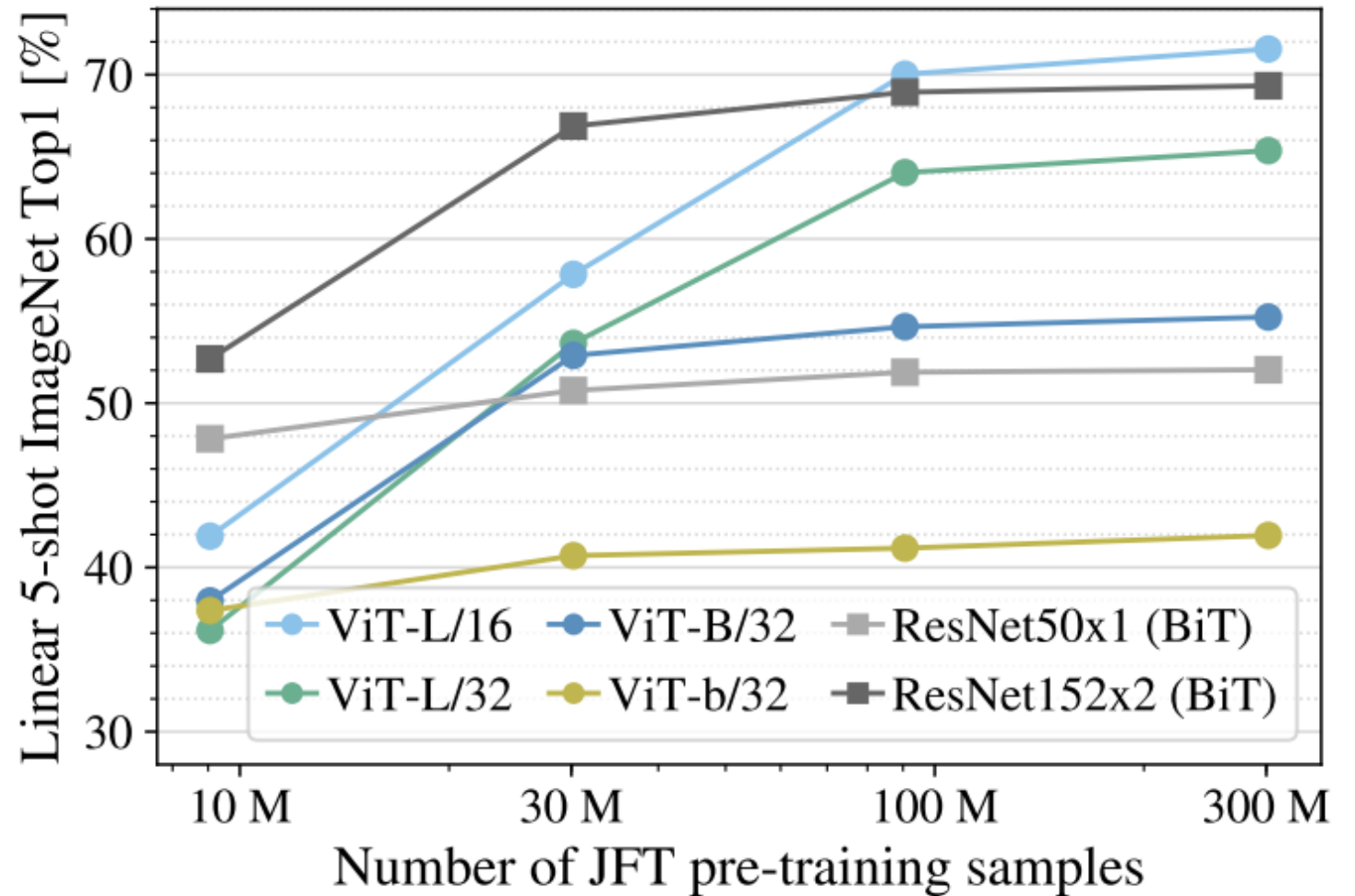
- Results from Bengio et al. (2006) on MNIST digit classification task
- Percent error (lower is better)
- Some methods first do pre-training
- Every method includes fine-tuning on labeled data



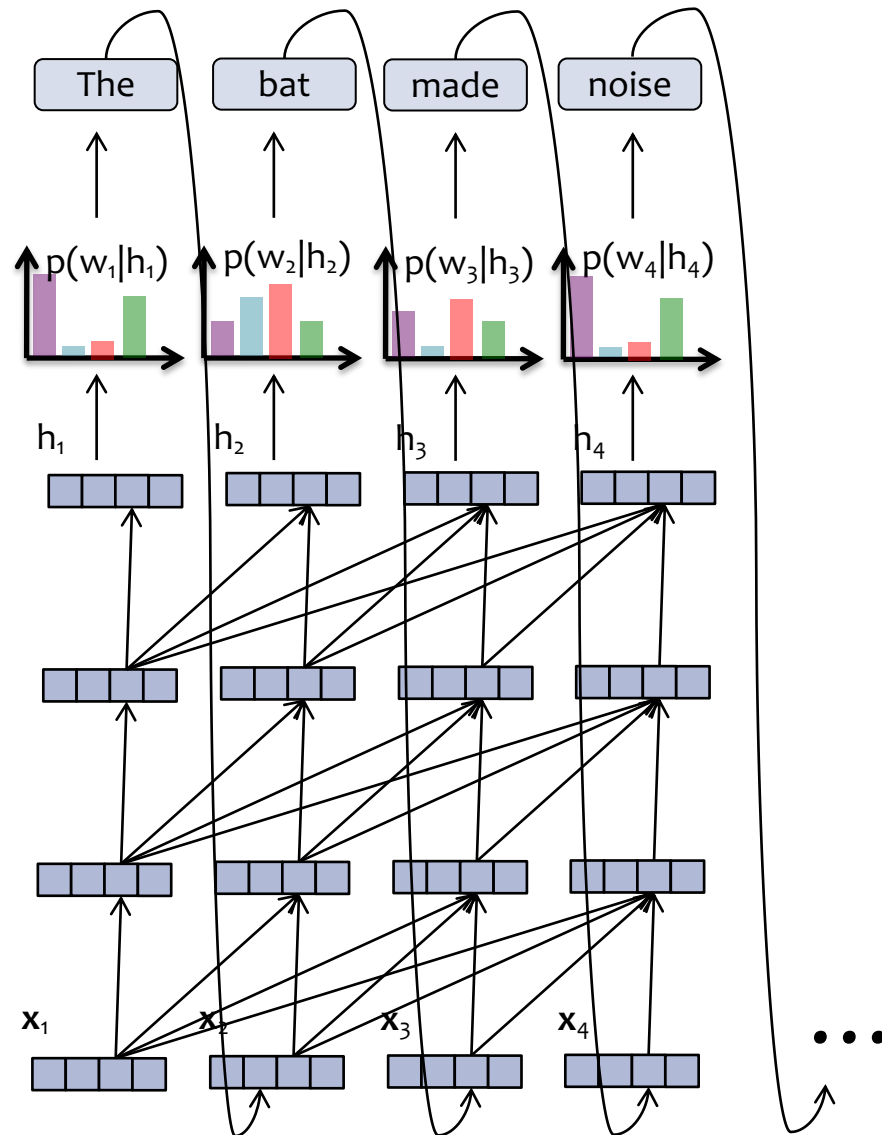


# Supervised Pre-Training for Vision

- Nowadays, we tend to just do supervised pre-training on a massive labeled dataset
- Vision Transformer's success was largely due to using a much larger pre-training dataset



# Unsupervised Pre-Training for an LLM



**Generative pre-training** for a deep language model:

- each training example is an (unlabeled) sentence
- the objective function is the likelihood of the observed sentence

Practically, we can **batch** together many such training examples to make training more efficient

# Training Data for LLMs

## GPT-3 Training Data:

Dataset	Quantity (tokens)	Weight in training mix	Epochs elapsed when training for 300B tokens
Common Crawl (filtered)	410 billion	60%	0.44
WebText2	19 billion	22%	2.9
Books1	12 billion	8%	1.9
Books2	55 billion	8%	0.43
Wikipedia	3 billion	3%	3.4

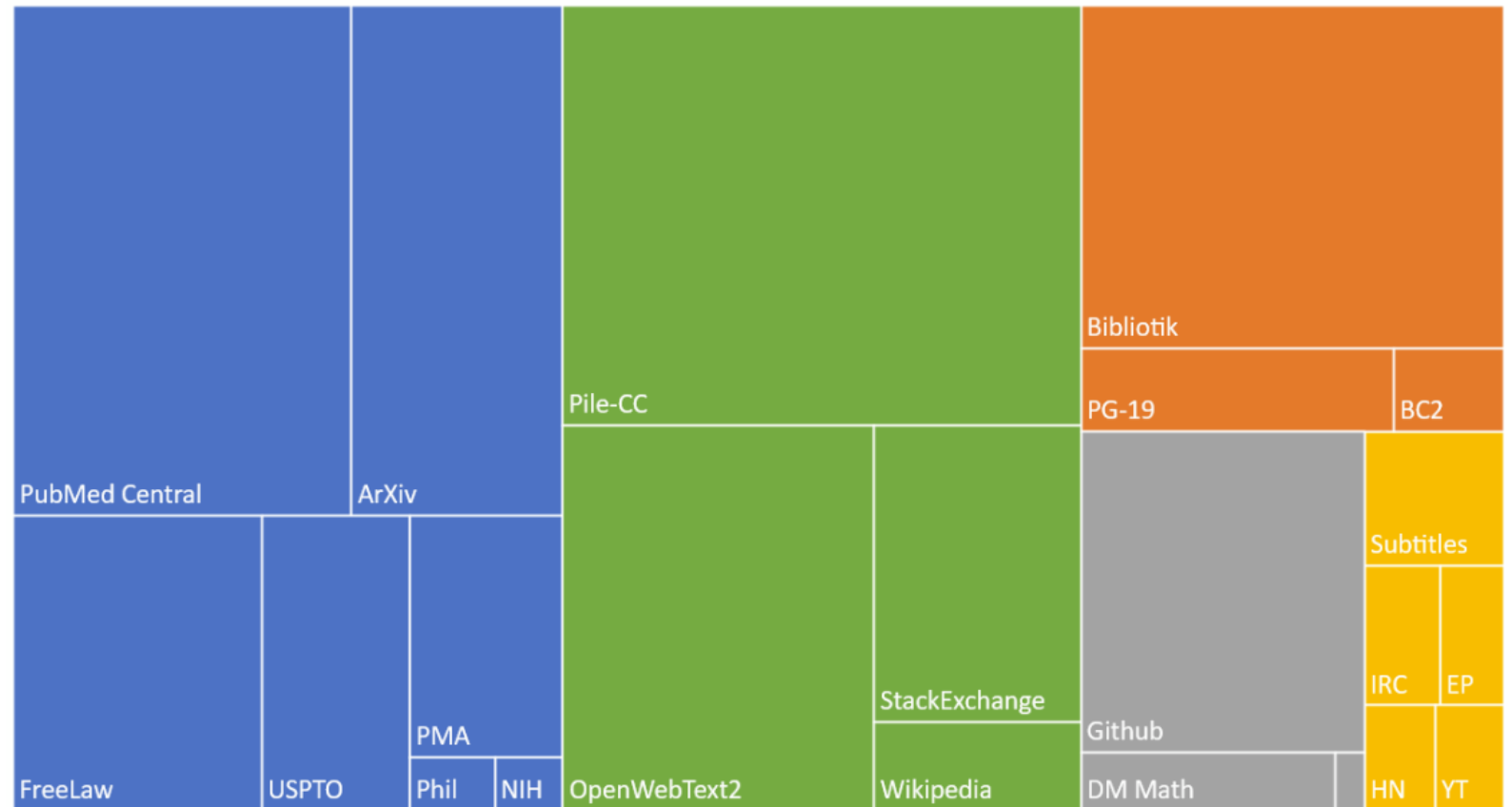
# Training Data for LLMs

## The Pile:

- An open source dataset for training language models
- Comprised of 22 smaller datasets
- Favors high quality text
- 825 Gb  $\approx$  1.2 trillion tokens

Composition of the Pile by Category

■ Academic ■ Internet ■ Prose ■ Dialogue ■ Misc



# Pre-Training vs. Fine-Tuning

## Definitions

### Pre-training

- randomly initialize the parameters, then...
- option A: unsupervised training on very large set of unlabeled instances
- option B: supervised training on a very large set of labeled examples

### Fine-tuning

- initialize parameters to values from pre-training
- (optionally), add a prediction head with a small number of randomly initialized parameters
- train on a specific task of interest by backprop

## Example: Vision Models

### Pre-training

- Example A: unsupervised autoencoder training on very large set of unlabeled images (e.g. MNIST digits)
- Example B: supervised training on a very large image classification dataset (e.g. ImageNet w/21k classes and 14M images)

### Fine-tuning

- object detection, training on 200k labeled images from COCO
- semantic segmentation, training on 20k labeled images from ADE20k

## Example: Language Models

### Pre-training

- unsupervised pre-training by maximizing likelihood of a large set of unlabeled sentences such as...
- The Pile (800 Gb of text)
- Dolma (3 trillion tokens)

### Fine-tuning

- MMLU benchmark: a few training examples from 57 different tasks ranging from elementary mathematics to genetics to law
- code generation, training on ~400 training examples from MBPP

# **POST TRAINING**

# RLHF

- Modern LLMs are not just fine-tuned, they are a variety of posting training stages that occur
- One of the most important is that of alignment, which can be done with algorithms like RLHF and DPO
- RLHF is a form of fine-tuning that uses *reinforcement learning* where the reward function is learned from human preferences

## Step 1

**Collect demonstration data, and train a supervised policy.**

A prompt is sampled from our prompt dataset.

Explain the moon landing to a 6 year old

A labeler demonstrates the desired output behavior.

Some people went to the moon...

This data is used to fine-tune GPT-3 with supervised learning.

SFT

## Step 2

**Collect comparison data, and train a reward model.**

A prompt and several model outputs are sampled.

Explain the moon landing to a 6 year old

A Explain gravity...  
B Explain war...  
C Moon is natural satellite of...  
D People went to the moon...

A labeler ranks the outputs from best to worst.

D > C > A = B

This data is used to train our reward model.

RM

## Step 3

**Optimize a policy against the reward model using reinforcement learning.**

A new prompt is sampled from the dataset.

Write a story about frogs

The policy generates an output.

PPO

Once upon a time...

The reward model calculates a reward for the output.

RM

The reward is used to update the policy using PPO.

$r_k$