



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

Matt Gormley & Henry Chai

Lecture 19

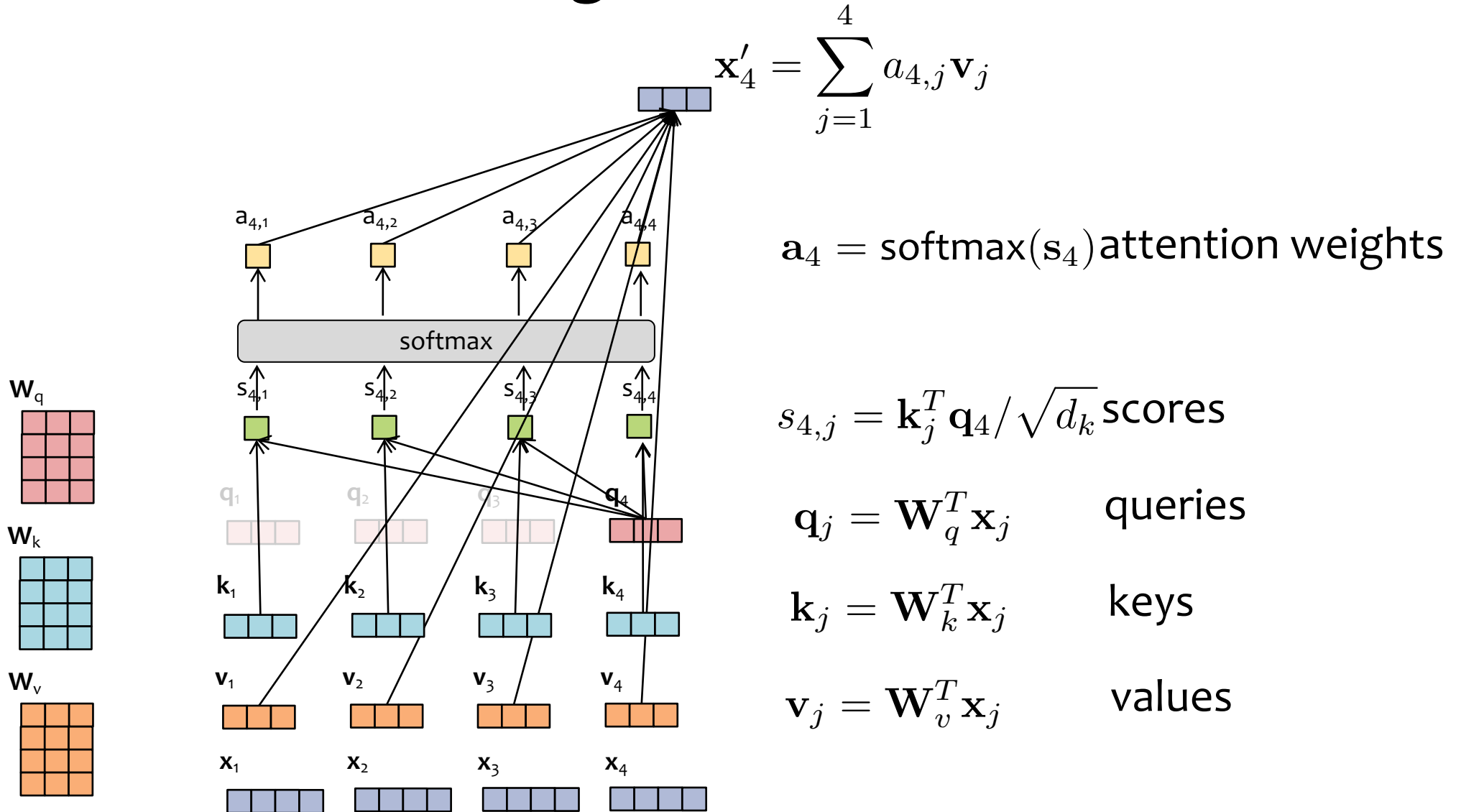
Mar. 27, 2024

# Reminders

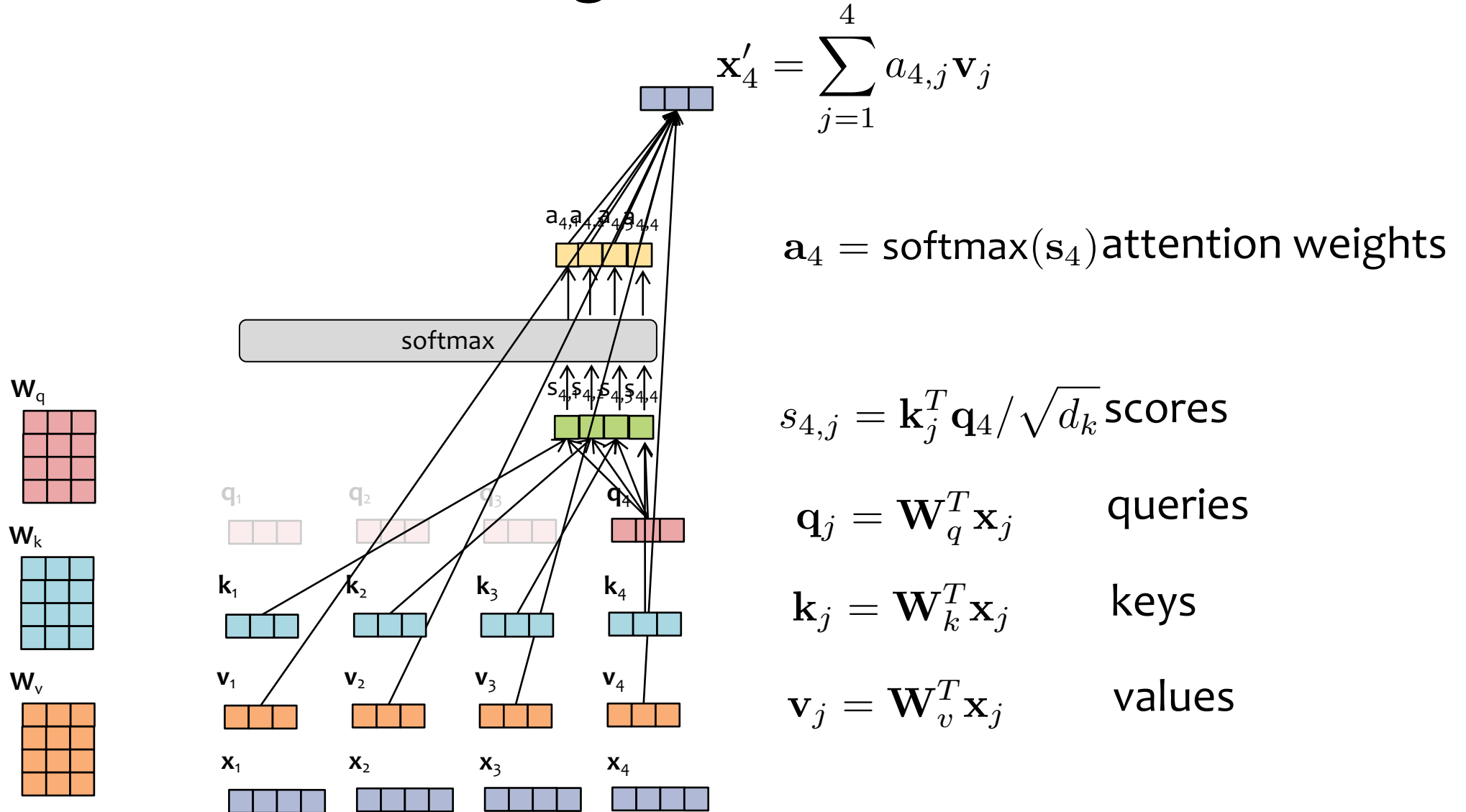
- **Exam 2: Thu, Nov 7, 6:45 pm - 8:45 pm**
- **Homework 7: Deep Learning & LLMs**
  - **Out: Thu, Nov 7**
  - **Due: Sun, Nov 17, 11:59pm**

# **IMPLEMENTING A TRANSFORMER LM**

# Matrix Version of Single-Headed Attention

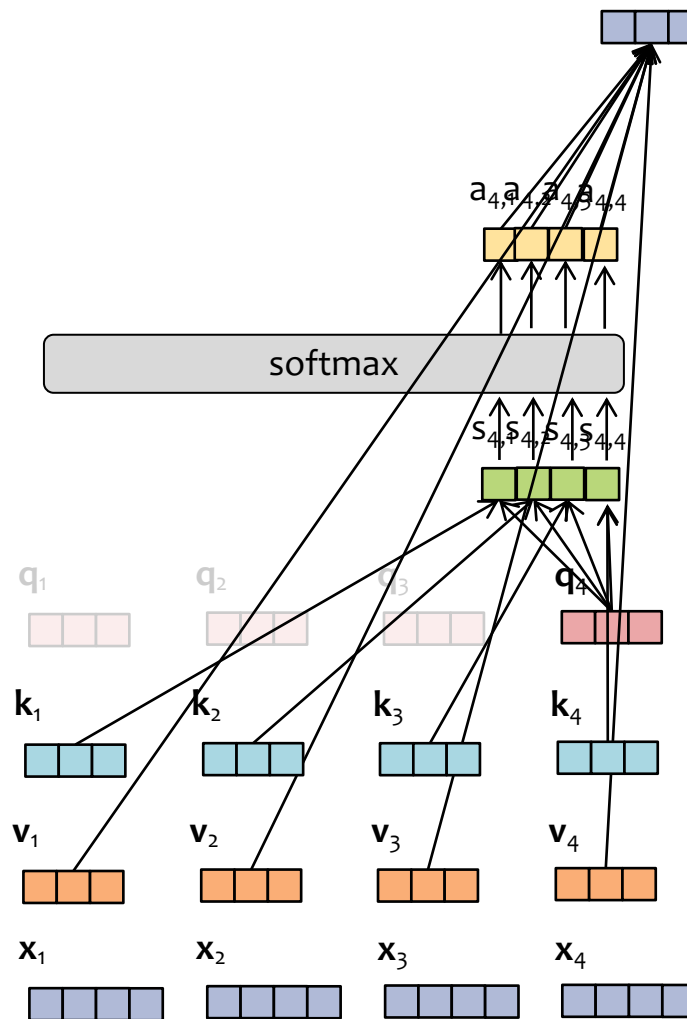
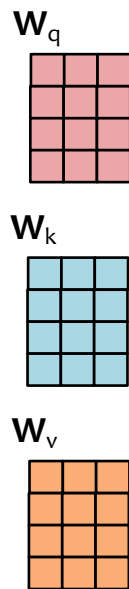


# Matrix Version of Single-Headed Attention



# 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



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

$$A = [a_1, \dots, a_4]^T = \text{softmax}(S)$$

$$S = [s_1, \dots, s_4]^T = QK^T / \sqrt{d_k}$$

$$Q = [q_1, \dots, q_4]^T = XW_q$$

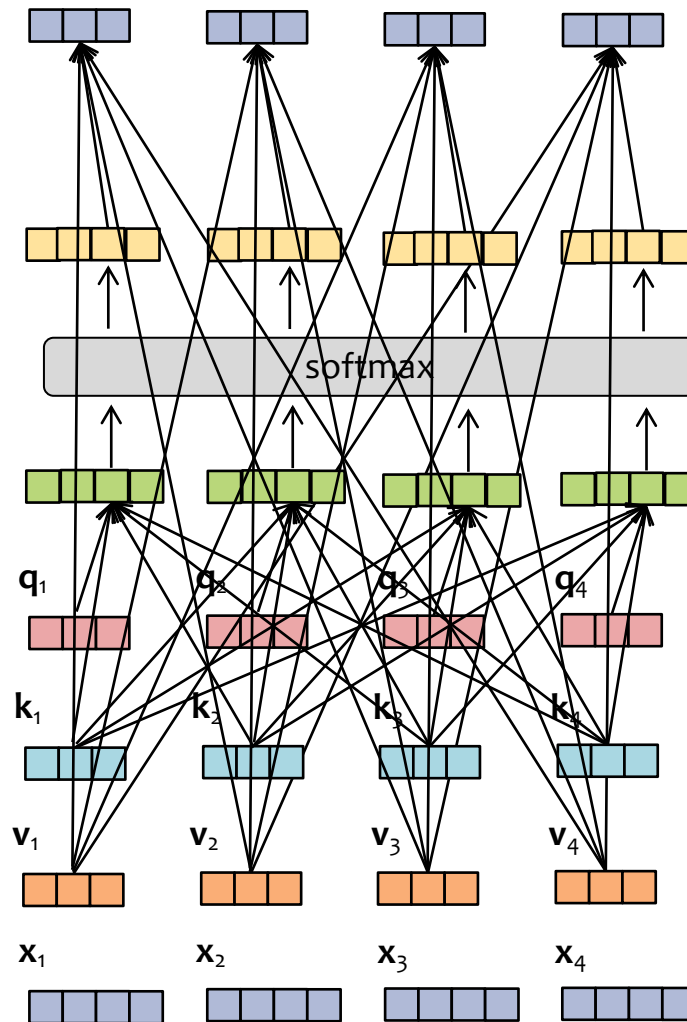
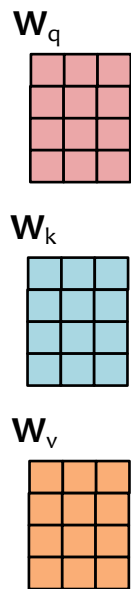
$$K = [k_1, \dots, k_4]^T = XW_k$$

$$V = [v_1, \dots, v_4]^T = XW_v$$

$$X = [x_1, \dots, x_4]^T$$

# 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



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

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

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

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

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

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

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

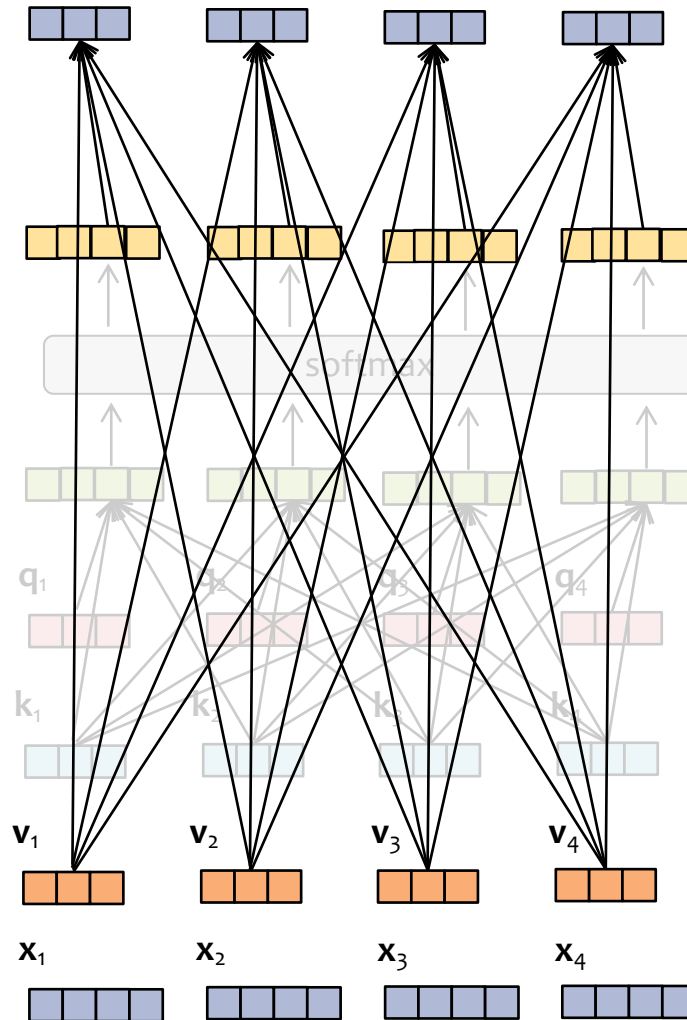
# Matrix Version of Single-Headed Attention

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!



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

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

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

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

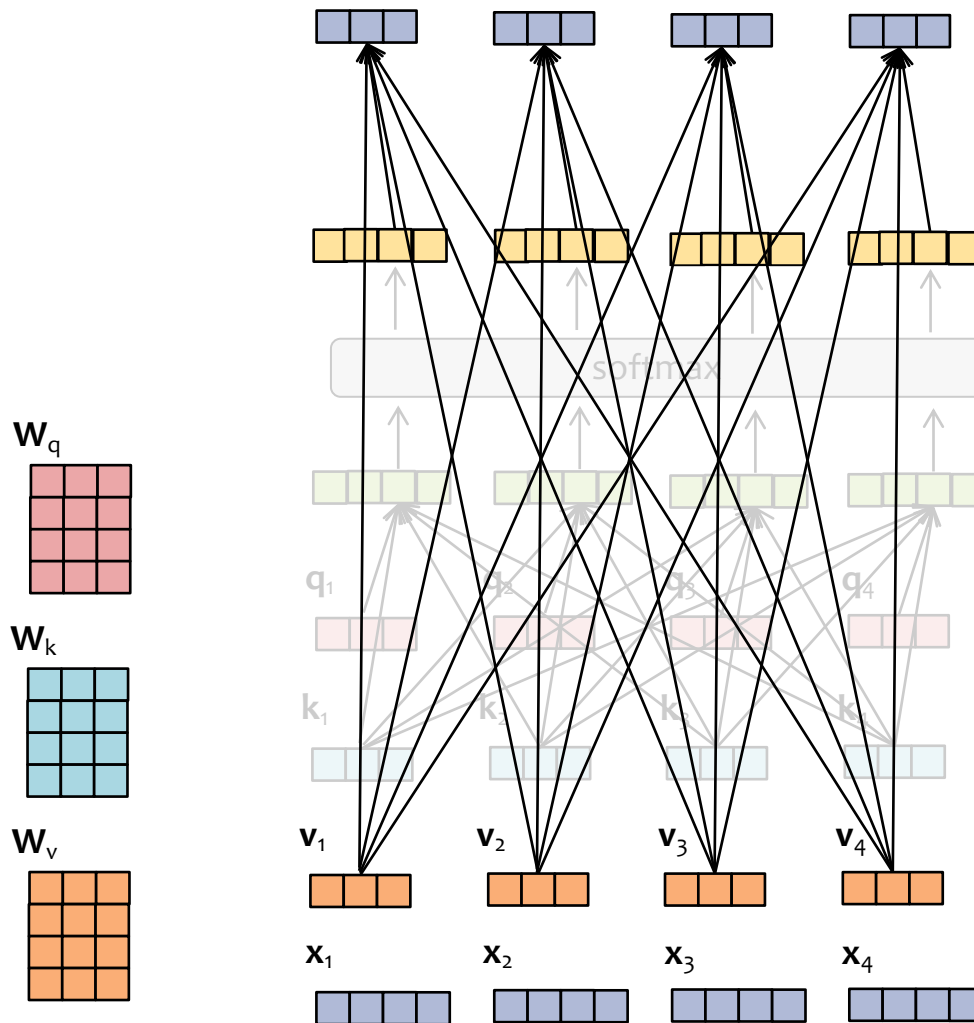
$$\mathbf{K} = [\mathbf{k}_1, \dots, \mathbf{k}_4]^T = \mathbf{X}\mathbf{W}_k$$

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

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



# Matrix Version of Single-Headed Attention



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

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

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

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

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

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

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

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

- A. column-wise
- B. row-wise

**Answer:**

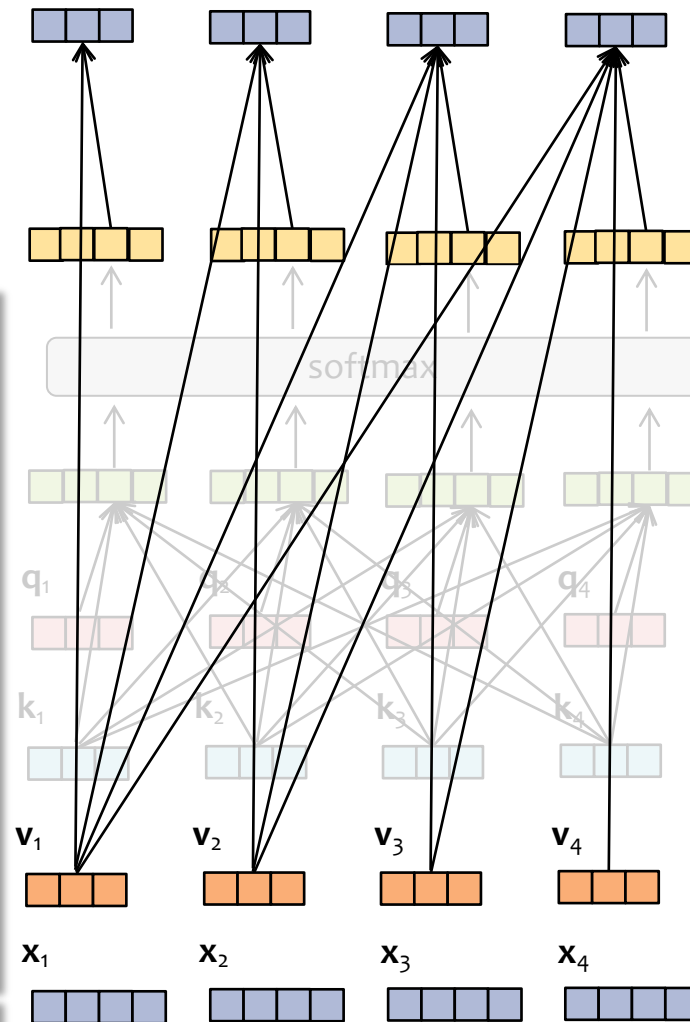
# Matrix Version of Single-Headed (Causal) Attention

**Insight:** if some element in the input to the softmax is  $-\infty$ , 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:**



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

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

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

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

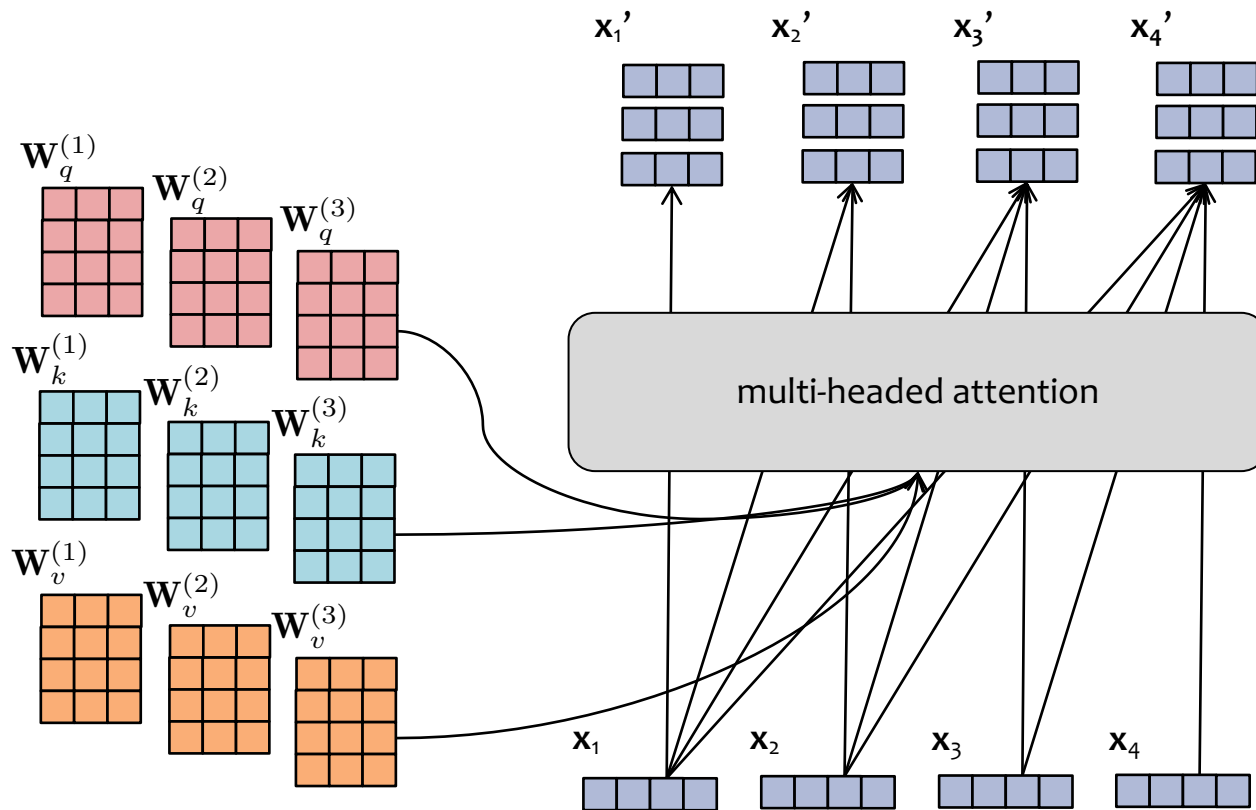
$$\mathbf{K} = \mathbf{X}\mathbf{W}_k$$

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

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

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

# Matrix Version of Multi-Headed (Causal) Attention



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

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

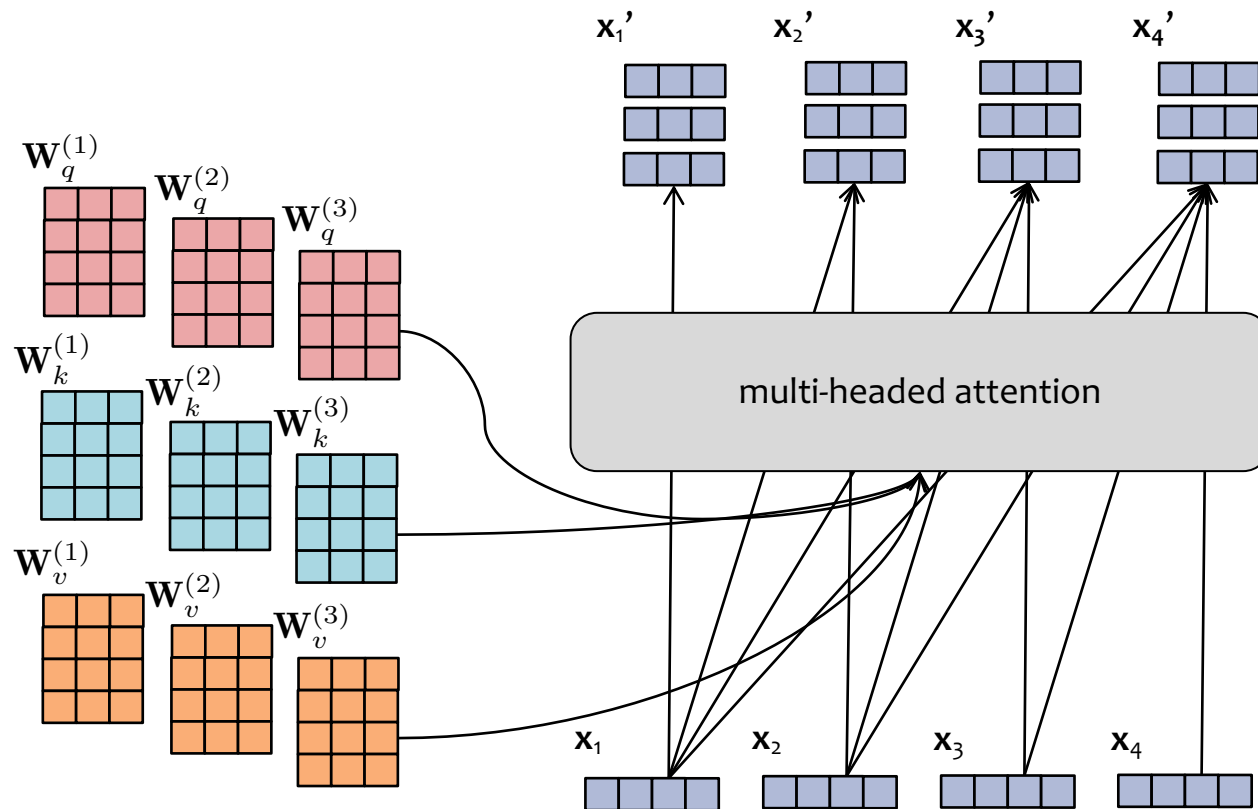
$$\mathbf{Q}^{(i)} = \mathbf{X} \mathbf{W}_q^{(i)}$$

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

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

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

# Matrix Version of Multi-Headed (Causal) Attention



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

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

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

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

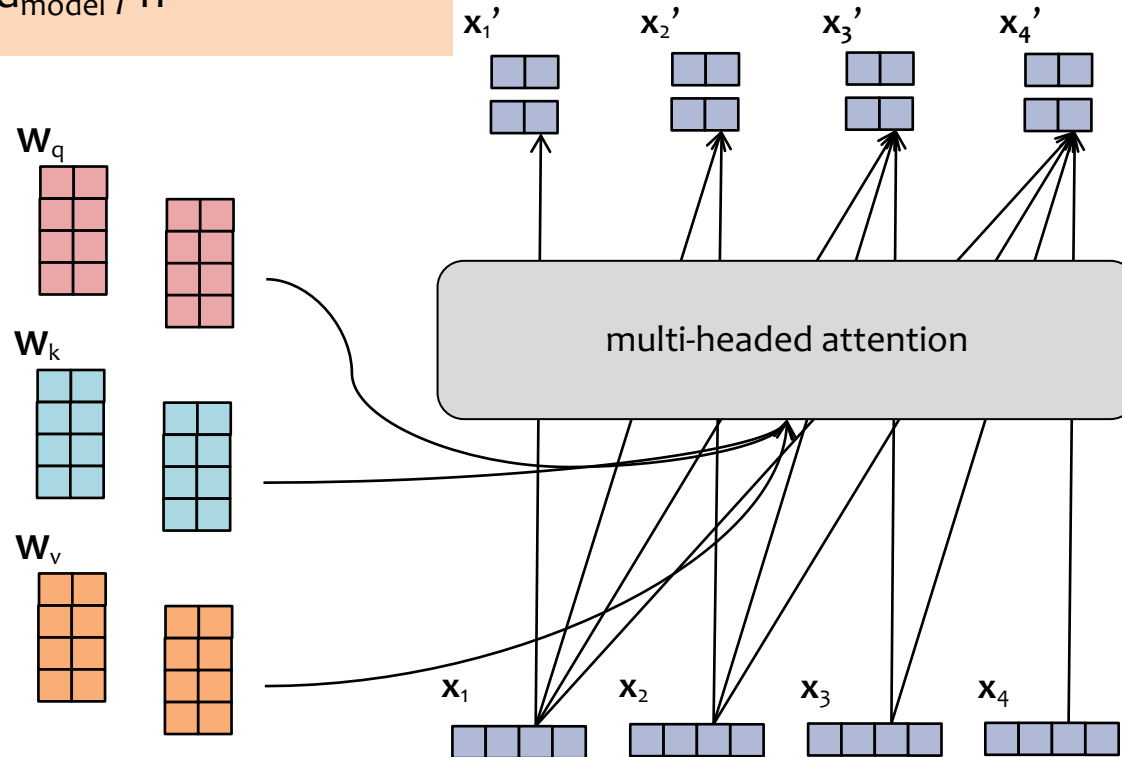
$$\mathbf{V}^{(i)} = \mathbf{X} \mathbf{W}_v^{(i)}$$

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

## 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:

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



# Construction of Multi-Headed (Causal) Attention

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

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

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

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

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

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

# **PRACTICALITIES OF TRANSFORMER LMS**

# In-Class Poll

## Question:

Suppose we have the following input embeddings and attention weights:

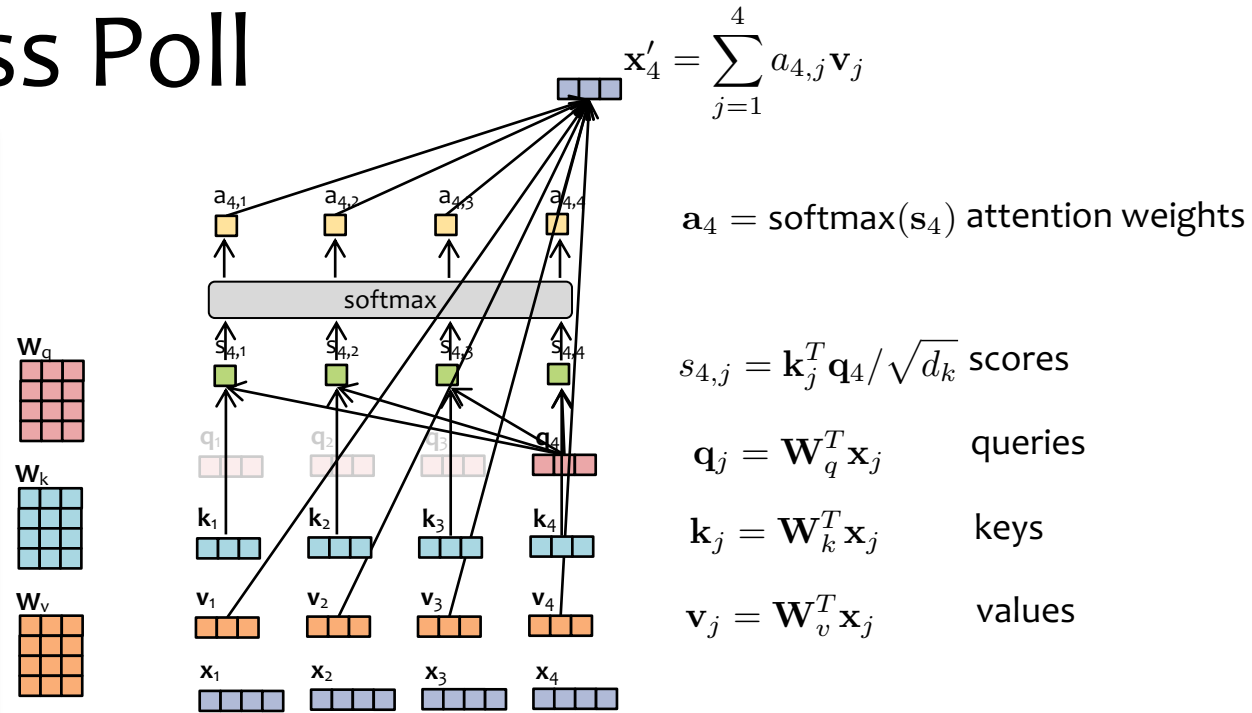
- $x_1 = [1, 0, 0, 0]$   $a_{4,1} = 0.1$
- $x_2 = [0, 1, 0, 0]$   $a_{4,2} = 0.2$
- $x_3 = [0, 0, 2, 0]$   $a_{4,3} = 0.6$
- $x_4 = [0, 0, 0, 1]$   $a_{4,4} = 0.1$

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

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

- $x_2 = [0, 0, 2, 0]$
- $x_3 = [0, 1, 0, 0]$

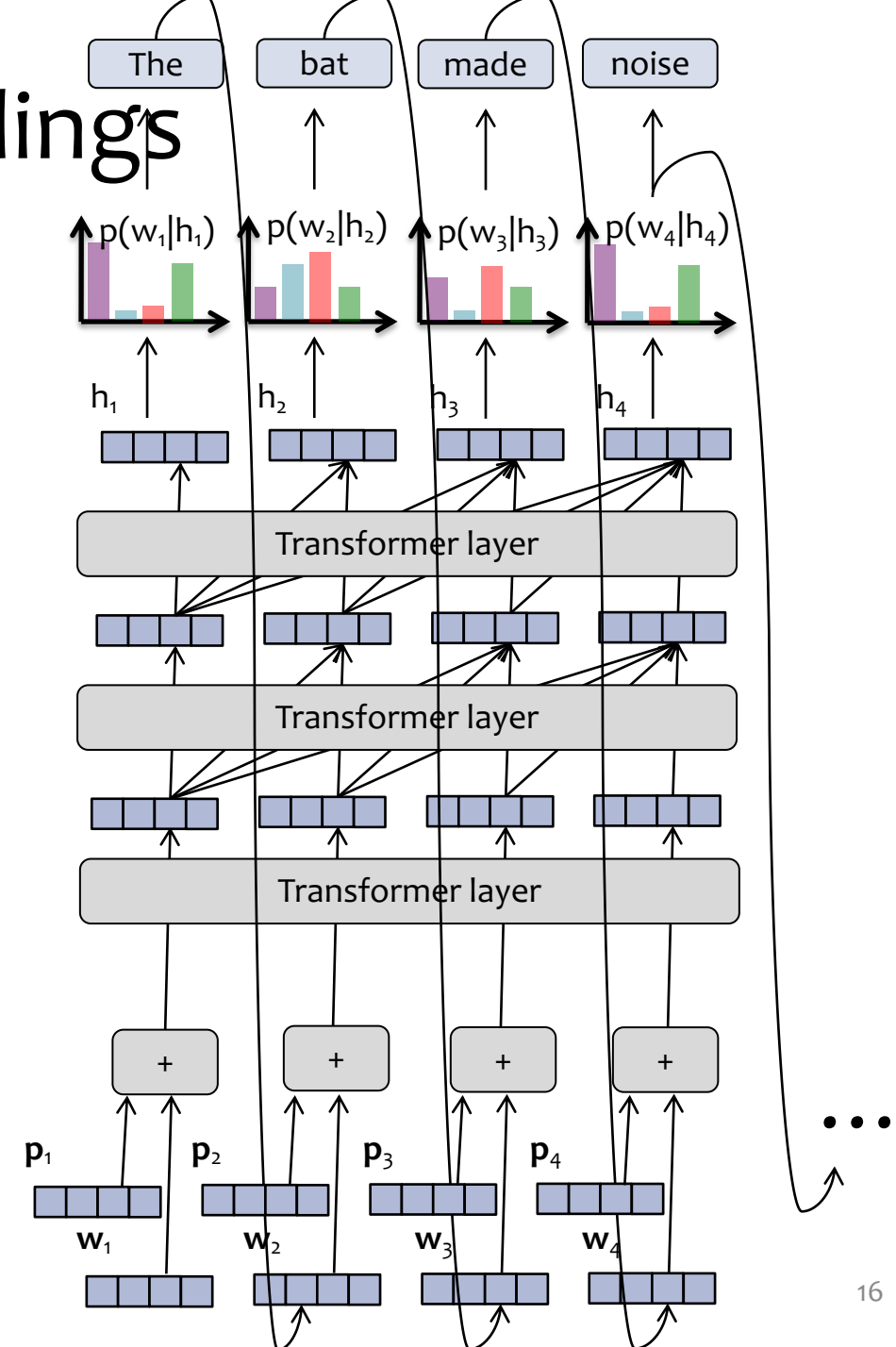
What is the new value of  $x_4'$ ?



## 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)





# 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

$i$	$w_1$	$w_2$	$w_3$	$w_4$	$w_5$	$w_6$	$w_7$	$w_8$	$w_9$	$w_{10}$	$w_{11}$	$w_{12}$
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	<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

$i$	$w_1$	$w_2$	$w_3$	$w_4$	$w_5$	$w_6$	$w_7$	$w_8$	$w_9$	$w_{10}$
1										
2										
3										
4										
5										
6										
7										
8										

**Embeddings:**

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

# TOKENIZATION

# Tokenization

## Word-based Tokenizer:

Input: “Henry is giving a lecture on transformers”

Output: [“henry”, “is”, “giving”, “a”, “lecture”, “on”, “transformers”]

## Pros/Cons:

- Can have difficulty trading off between vocabulary size and computational tractability
- Similar words e.g., “transformers” and “transformer” can get mapped to completely disparate representations
- Typos will typically be out-of-vocabulary (OOV)

# Tokenization

## Word-based Tokenizer:

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

Output: [“henry”, “is”, <OOV>, “a”, <OOV>, “on”, “transformers”]

## Pros/Cons:

- Can have difficulty trading off between vocabulary size and computational tractability
- Similar words e.g., “transformers” and “transformer” can get mapped to completely disparate representations
- Typos will typically be out-of-vocabulary (OOV)



# Tokenization

## Character-based Tokenizer:

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

Output: [“h”, “e”, “n”, “r”, “y”, “i”, “s”, “g”, “i”, “v”, “i”, “n”, “ ’ ”, ... ]

## Pros/Cons:

- Much smaller vocabularies but a lot of semantic meaning is lost...
- Sequences will be much longer than word-based tokenization, potentially causing computational issues
- Can do well on logographic languages e.g., Kanji 漢字

# Tokenization

## Subword-based Tokenizer:

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

Output: [“henry”, “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

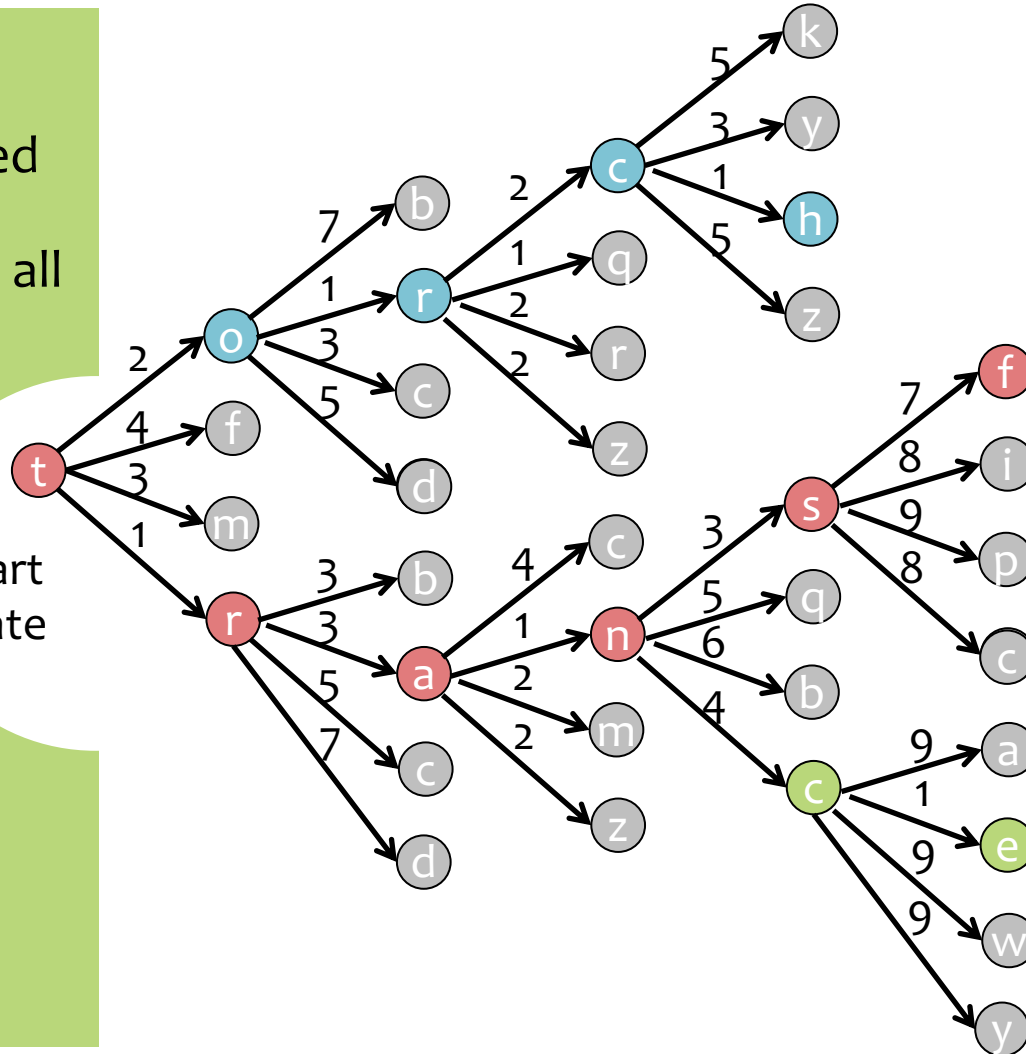
# **GREEDY DECODING FOR A LANGUAGE MODEL**

# Greedy Decoding for a Language Model

## Setup:

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

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 find the highest probably (lowest negative log probability) path from root to a leaf

## Greedy Search:

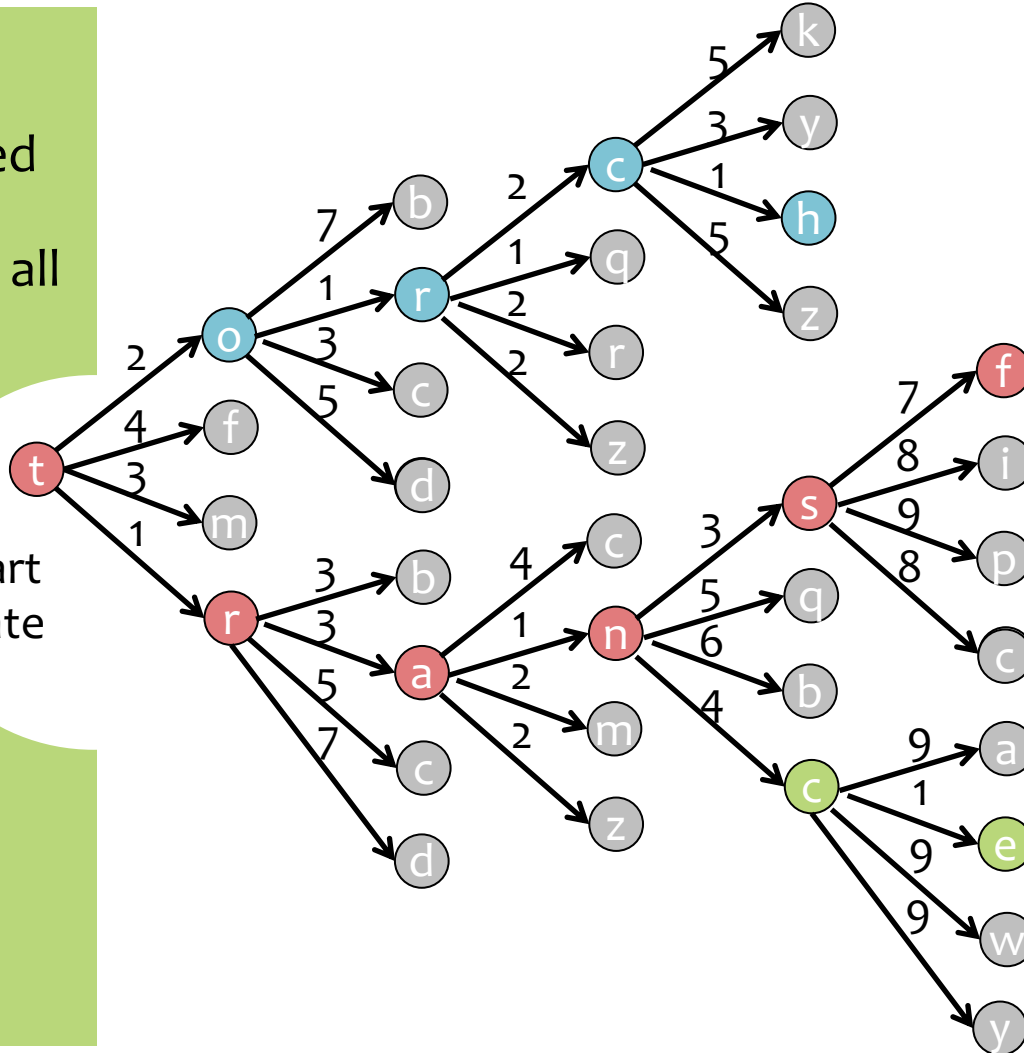
- At each node, selects the edge with lowest negative log probability
- **Heuristic** method of search (i.e. does not necessarily find the best path)
- Computation time: **linear** in max path length

# 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

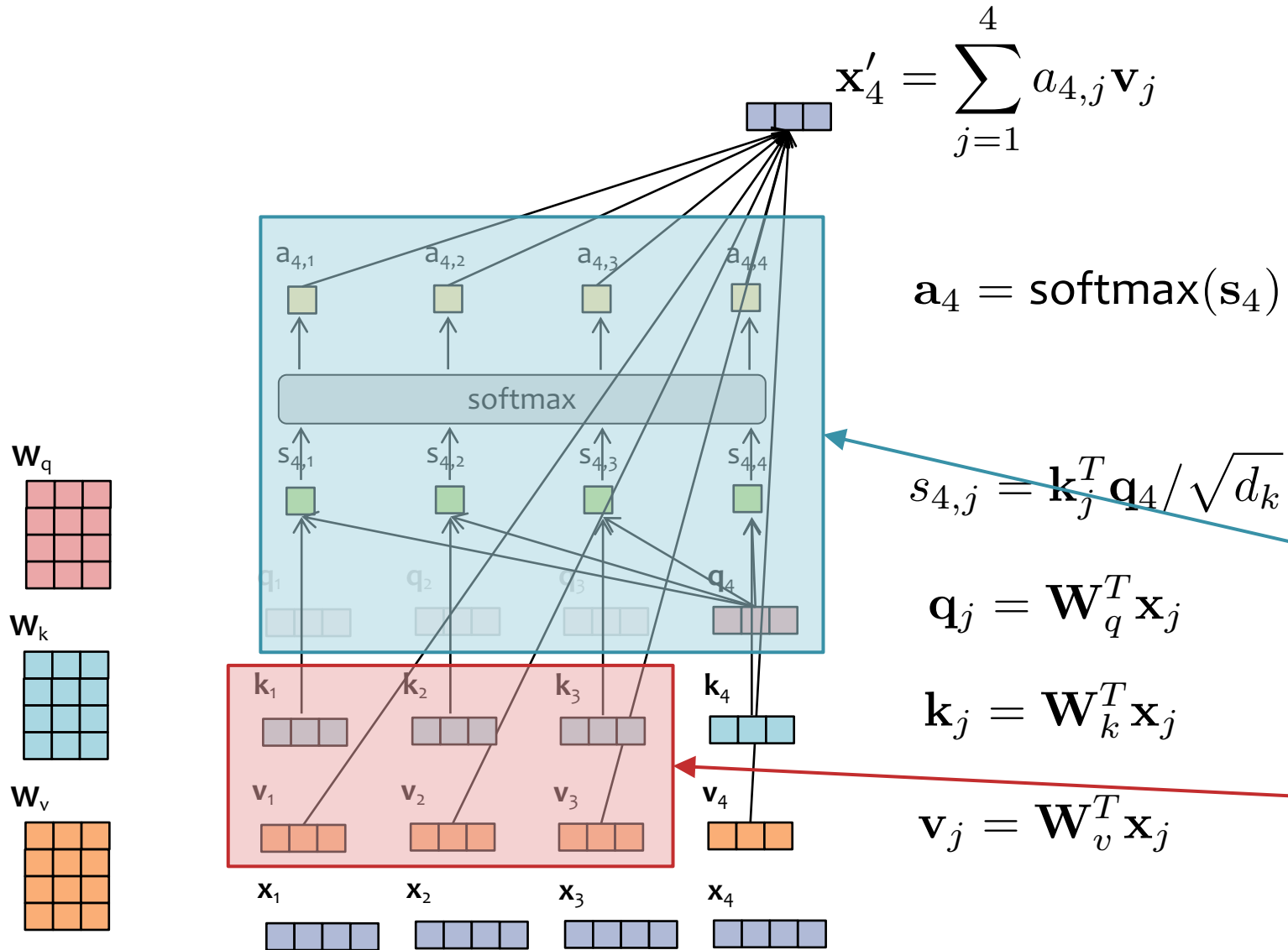
## Goal:

- 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



- 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 (i.e. we can let them fall out of the cache)

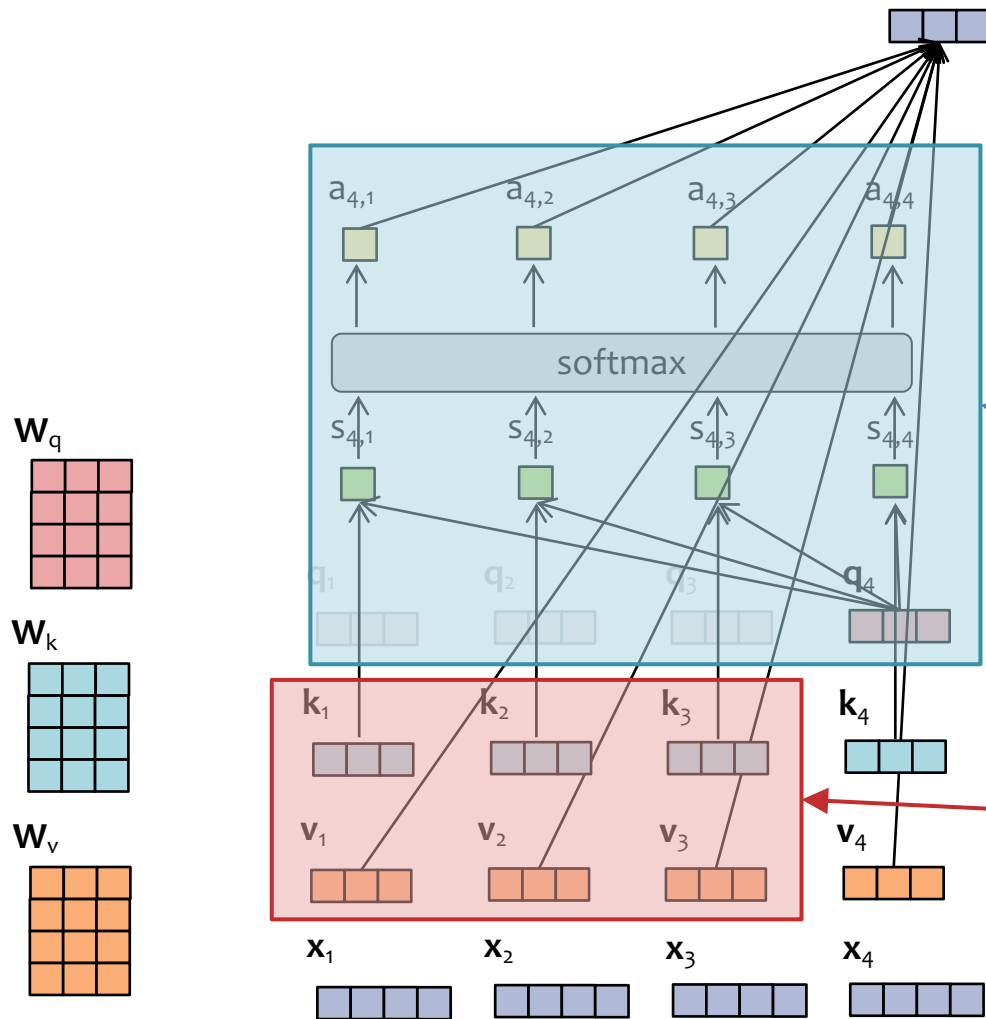
Discarded after this timestep

Computed for previous time-steps and reused for this timestep

# Key-Value Cache

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

- 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 (i.e. we can let them fall out of the cache)



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

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

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

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

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

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

Discarded after this timestep

Computed for previous time-steps and reused for this timestep

# 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)

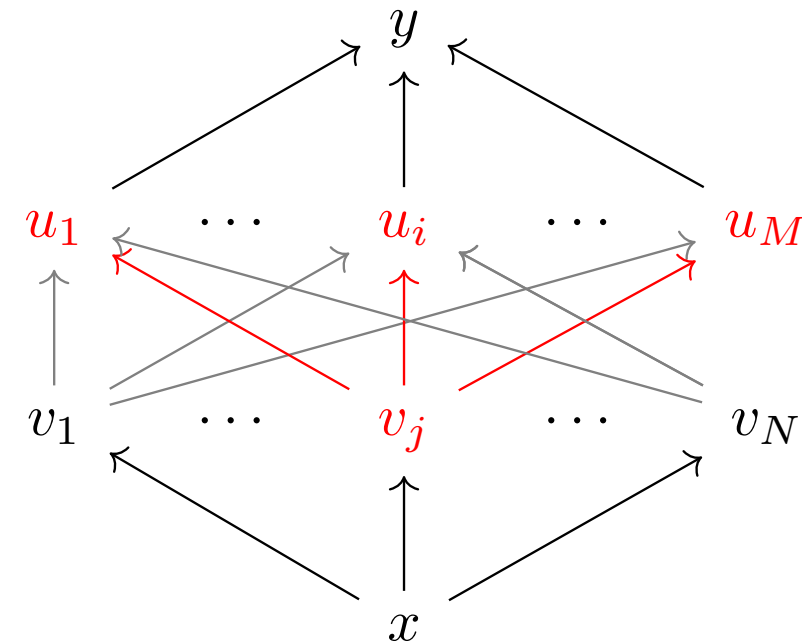
### Forward Computation

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  $\mathbf{u} = \mathbf{g}(\mathbf{v})$  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

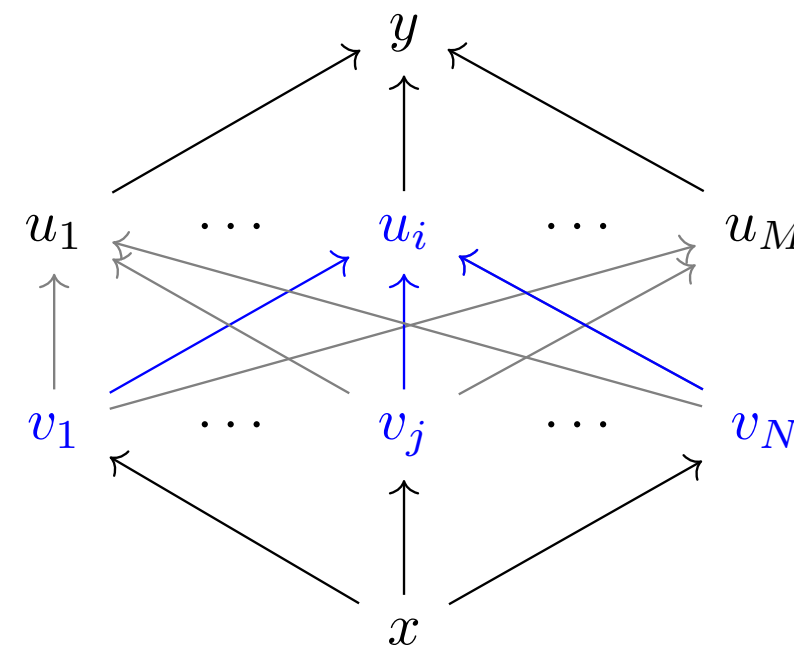
## Automatic Differentiation – Reverse Mode (aka. Backpropagation)

### 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

*Why is the backpropagation algorithm efficient?*

1. Reuses **computation from the forward pass** in the backward pass
2. Reuses **partial derivatives** throughout the backward pass (*but only if the algorithm reuses shared computation in the forward pass*)

(Key idea: partial derivatives in the backward pass should be thought of as variables stored for reuse)

# Gradients

1. Given training data

$$\{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^N$$

2. Choose each of the

– Decision function

$$\hat{\mathbf{y}} = f_{\boldsymbol{\theta}}(\mathbf{x}_i)$$

– Loss function

$$\ell(\hat{\mathbf{y}}, \mathbf{y}_i) \in \mathbb{R}$$

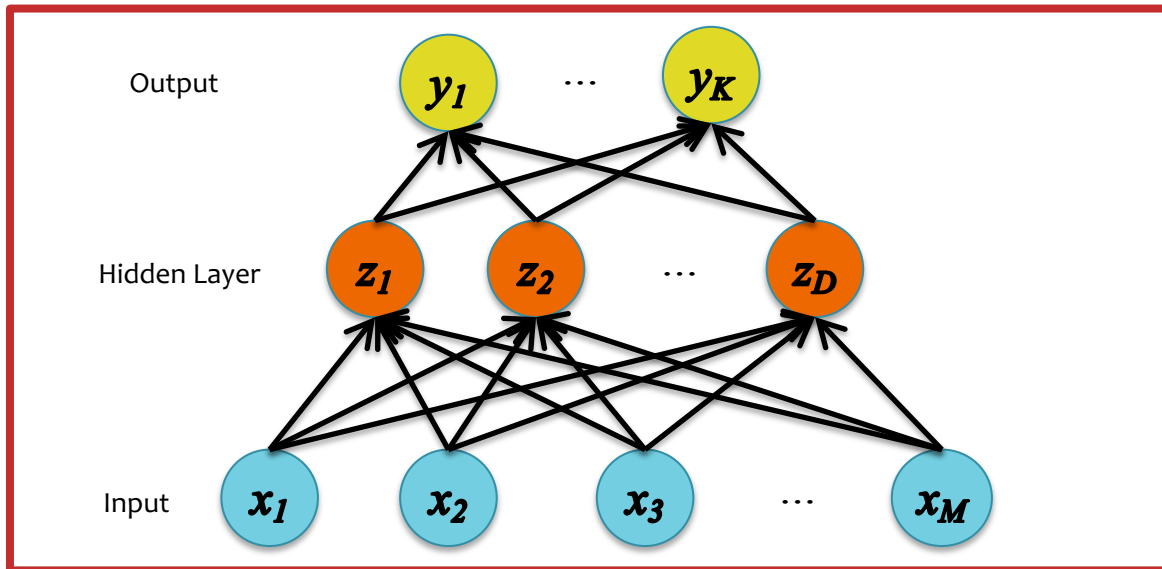
**Backpropagation** can compute this gradient!

And it's a **special case of a more general algorithm** called reverse-mode automatic differentiation that can compute the gradient of any differentiable function efficiently!

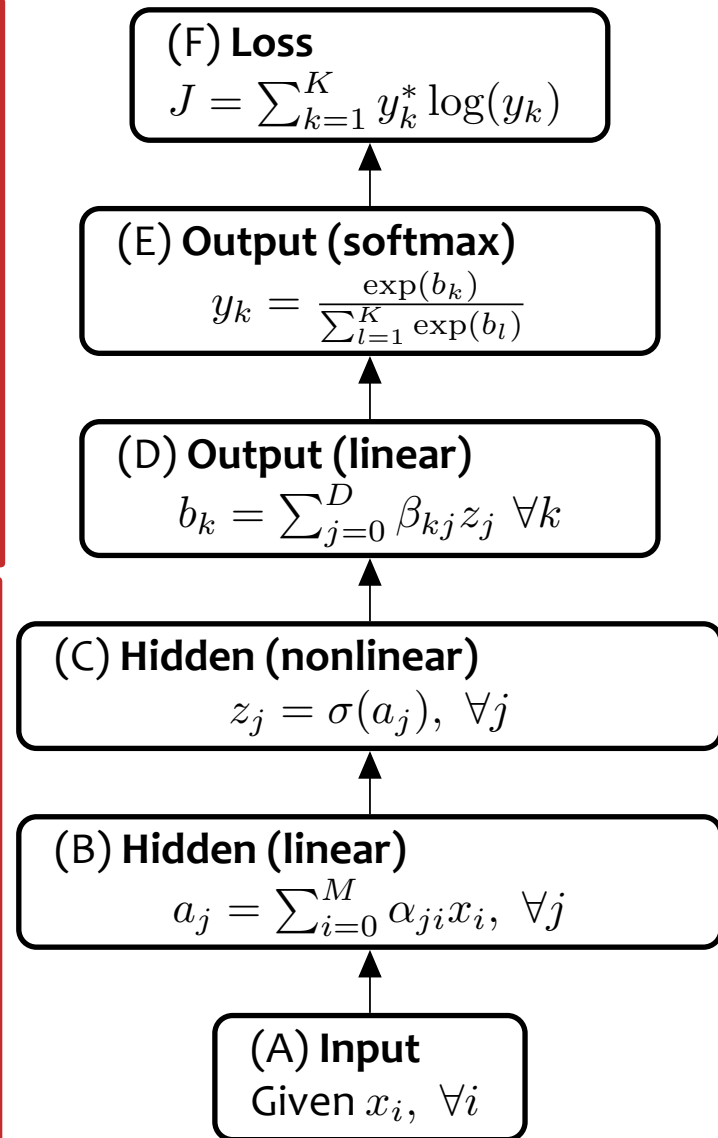
opposite the gradient)


$$\boldsymbol{\theta}^{(t)} \leftarrow \boldsymbol{\theta}^{(t)} - \eta_t \nabla \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_i), \mathbf{y}_i)$$

# Backpropagation: Abstract Picture



Forward	Backward
5. $J = -\mathbf{y}^T \log \hat{\mathbf{y}}$	6. $\mathbf{g}_{\hat{\mathbf{y}}} = -\mathbf{y} \div \hat{\mathbf{y}}$
4. $\hat{\mathbf{y}} = \text{softmax}(\mathbf{b})$	7. $\mathbf{g}_{\mathbf{b}} = \mathbf{g}_{\hat{\mathbf{y}}}^T (\text{diag}(\hat{\mathbf{y}}) - \hat{\mathbf{y}}\hat{\mathbf{y}}^T)$
3. $\mathbf{b} = \beta \mathbf{z}$	8. $\mathbf{g}_{\beta} = \mathbf{g}_{\mathbf{b}}^T \mathbf{z}^T$ $\mathbf{g}_{\mathbf{z}} = \beta^T \mathbf{g}_{\mathbf{b}}^T$
2. $\mathbf{z} = \sigma(\mathbf{a})$	10. $\mathbf{g}_{\mathbf{a}} = \mathbf{g}_{\mathbf{z}} \odot \mathbf{z} \odot (1 - \mathbf{z})$
1. $\mathbf{a} = \alpha \mathbf{x}$	11. $\mathbf{g}_{\alpha} = \mathbf{g}_{\mathbf{a}} \mathbf{x}^T$



# 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:    $\mathbf{g}_{\mathbf{z}} = \beta^T \mathbf{g}_{\mathbf{b}}$ 
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

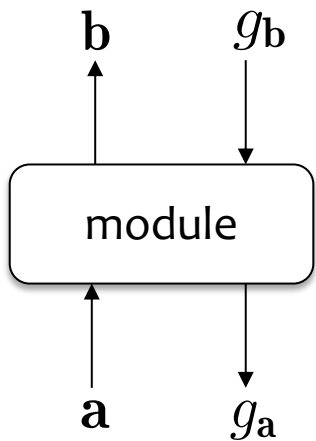
Module-based automatic differentiation (AD / Autodiff) is a technique that has long been used to develop libraries for deep learning

- **Dynamic neural network packages** allow a specification of the computation graph dynamically at runtime
  - PyTorch <http://pytorch.org>
  - Torch <http://torch.ch>
  - DyNet <https://dynet.readthedocs.io>
  - TensorFlow with Eager Execution <https://www.tensorflow.org>
- **Static neural network packages** require a static specification of a computation graph which is subsequently compiled into code
  - TensorFlow with Graph Execution <https://www.tensorflow.org>
  - Aesara (and Theano) <https://aesara.readthedocs.io>
  - *(These libraries are also module-based, but herein by “module-based AD” we mean the dynamic approach)*



# 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  $i \in \{1, \dots, A\}$ .



# Module-based AutoDiff

**Dimensions:** input  $\mathbf{a} \in \mathbb{R}^A$ , output  $\mathbf{b} \in \mathbb{R}^B$ , gradient of output  $\mathbf{g}_a \triangleq \nabla_{\mathbf{a}} J \in \mathbb{R}^A$ , and gradient of input  $\mathbf{g}_b \triangleq \nabla_{\mathbf{b}} J \in \mathbb{R}^B$ .

**Sigmoid Module** The sigmoid layer has only one input vector  $\mathbf{a}$ . Below  $\sigma$  is the sigmoid applied element-wise, and  $\odot$  is element-wise multiplication s.t.  $\mathbf{u} \odot \mathbf{v} = [u_1 v_1, \dots, u_M v_M]$ .

```
1: procedure SIGMOIDFORWARD(a)
2:    $\mathbf{b} = \sigma(\mathbf{a})$ 
3:   return  $\mathbf{b}$ 
4: procedure SIGMOIDBACKWARD(a, b,  $\mathbf{g}_b$ )
5:    $\mathbf{g}_a = \mathbf{g}_b \odot \mathbf{b} \odot (1 - \mathbf{b})$ 
6:   return  $\mathbf{g}_a$ 
```

**Softmax Module** The softmax layer has only one input vector  $\mathbf{a}$ . For any vector  $\mathbf{v} \in \mathbb{R}^D$ , we have that  $\text{diag}(\mathbf{v})$  returns a  $D \times D$  diagonal matrix whose diagonal entries are  $v_1, v_2, \dots, v_D$  and whose non-diagonal entries are zero.

```
1: procedure SOFTMAXFORWARD(a)
2:    $\mathbf{b} = \text{softmax}(\mathbf{a})$ 
3:   return  $\mathbf{b}$ 
4: procedure SOFTMAXBACKWARD(a, b,  $\mathbf{g}_b$ )
5:    $\mathbf{g}_a = \mathbf{g}_b^T (\text{diag}(\mathbf{b}) - \mathbf{b}\mathbf{b}^T)$ 
6:   return  $\mathbf{g}_a$ 
```

**Linear Module** The linear layer has two inputs: a vector  $\mathbf{a}$  and parameters  $\omega \in \mathbb{R}^{B \times A}$ . The output  $\mathbf{b}$  is not used by LINEARBACKWARD, but we pass it in for consistency of form.

```
1: procedure LINEARFORWARD(a,  $\omega$ )
2:    $\mathbf{b} = \omega \mathbf{a}$ 
3:   return  $\mathbf{b}$ 
4: procedure LINEARBACKWARD(a,  $\omega$ ,  $\mathbf{b}$ ,  $\mathbf{g}_b$ )
5:    $\mathbf{g}_\omega = \mathbf{g}_b \mathbf{a}^T$ 
6:    $\mathbf{g}_a = \omega^T \mathbf{g}_b$ 
7:   return  $\mathbf{g}_\omega, \mathbf{g}_a$ 
```

**Cross-Entropy Module** The cross-entropy layer has two inputs: a gold one-hot vector  $\mathbf{a}$  and a predicted probability distribution  $\hat{\mathbf{a}}$ . Its output  $b \in \mathbb{R}$  is a scalar. Below  $\div$  is element-wise division. The output  $b$  is not used by CROSSENTROPYBACKWARD, but we pass it in for consistency of form.

```
1: procedure CROSSENTROPYFORWARD(a,  $\hat{\mathbf{a}}$ )
2:    $b = -\mathbf{a}^T \log \hat{\mathbf{a}}$ 
3:   return  $b$ 
4: procedure CROSSENTROPYBACKWARD(a,  $\hat{\mathbf{a}}$ ,  $b$ ,  $\mathbf{g}_b$ )
5:    $\mathbf{g}_{\hat{\mathbf{a}}} = -\mathbf{g}_b (\mathbf{a} \div \hat{\mathbf{a}})$ 
6:   return  $\mathbf{g}_a$ 
```

# Module-based AutoDiff

## Algorithm 1 Forward Computation

```
1: procedure NNFORWARD(Training example  $(x, y)$ , Parameters  $\alpha, \beta$ )
2:    $\mathbf{a} = \text{LINEARFORWARD}(x, \alpha)$ 
3:    $\mathbf{z} = \text{SIGMOIDFORWARD}(\mathbf{a})$ 
4:    $\mathbf{b} = \text{LINEARFORWARD}(\mathbf{z}, \beta)$ 
5:    $\hat{y} = \text{SOFTMAXFORWARD}(\mathbf{b})$ 
6:    $J = \text{CROSSENTROPYFORWARD}(y, \hat{y})$ 
7:    $\mathbf{o} = \text{object}(x, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{y}, J)$ 
8:   return intermediate quantities  $\mathbf{o}$ 
```

## Algorithm 2 Backpropagation

```
1: procedure NNBACKWARD(Training example  $(x, y)$ , Parameters  $\alpha, \beta$ , Intermediates  $\mathbf{o}$ )
2:   Place intermediate quantities  $x, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{y}, J$  in  $\mathbf{o}$  in scope
3:    $g_J = \frac{dJ}{dJ} = 1$  ▷ Base case
4:    $\mathbf{g}_{\hat{y}} = \text{CROSSENTROPYBACKWARD}(y, \hat{y}, J, g_J)$ 
5:    $\mathbf{g}_{\mathbf{b}} = \text{SOFTMAXBACKWARD}(\mathbf{b}, \hat{y}, \mathbf{g}_{\hat{y}})$ 
6:    $\mathbf{g}_{\beta}, \mathbf{g}_{\mathbf{z}} = \text{LINEARBACKWARD}(\mathbf{z}, \mathbf{b}, \mathbf{g}_{\mathbf{b}})$ 
7:    $\mathbf{g}_{\mathbf{a}} = \text{SIGMOIDBACKWARD}(\mathbf{a}, \mathbf{z}, \mathbf{g}_{\mathbf{z}})$ 
8:    $\mathbf{g}_{\alpha}, \mathbf{g}_x = \text{LINEARBACKWARD}(x, \mathbf{a}, \mathbf{g}_{\mathbf{a}})$  ▷ We discard  $\mathbf{g}_x$ 
9:   return parameter gradients  $\mathbf{g}_{\alpha}, \mathbf{g}_{\beta}$ 
```

## Advantages of Module-based AutoDiff

1. Easy to reuse / adapt for other models
2. Encapsulated layers are easier to optimize (e.g. implement in C++ or CUDA)
3. Easier to find bugs because we can run a finite-difference check on each layer separately

# 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          $\mathbf{b} = \sigma(\mathbf{a})$ 
4         return  $\mathbf{b}$ 
5     method backward(a, b,  $\mathbf{g}_b$ )
6          $\mathbf{g}_a = \mathbf{g}_b \odot \mathbf{b} \odot (1 - \mathbf{b})$ 
7         return  $\mathbf{g}_a$ 
```

```
1 class Softmax(Module)
2     method forward(a)
3          $\mathbf{b} = \text{softmax}(\mathbf{a})$ 
4         return  $\mathbf{b}$ 
5     method backward(a, b,  $\mathbf{g}_b$ )
6          $\mathbf{g}_a = \mathbf{g}_b^T (\text{diag}(\mathbf{b}) - \mathbf{b}\mathbf{b}^T)$ 
7         return  $\mathbf{g}_a$ 
```

```
1 class Linear(Module)
2     method forward(a,  $\omega$ )
3          $\mathbf{b} = \omega\mathbf{a}$ 
4         return  $\mathbf{b}$ 
5     method backward(a,  $\omega$ , b,  $\mathbf{g}_b$ )
6          $\mathbf{g}_\omega = \mathbf{g}_b\mathbf{a}^T$ 
7          $\mathbf{g}_a = \omega^T \mathbf{g}_b$ 
8         return  $\mathbf{g}_\omega, \mathbf{g}_a$ 
```

```
1 class CrossEntropy(Module)
2     method forward(a,  $\hat{\mathbf{a}}$ )
3          $b = -\mathbf{a}^T \log \hat{\mathbf{a}}$ 
4         return  $\mathbf{b}$ 
5     method backward(a,  $\hat{\mathbf{a}}$ , b,  $\mathbf{g}_b$ )
6          $\mathbf{g}_{\hat{\mathbf{a}}} = -\mathbf{g}_b(\mathbf{a} \div \hat{\mathbf{a}})$ 
7         return  $\mathbf{g}_a$ 
```

# 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  $\boldsymbol{\alpha}$ , Tensor  $\boldsymbol{\beta}$ )
11         $\mathbf{a}$  = lin1_layer.apply_fwd( $\mathbf{x}$ ,  $\boldsymbol{\alpha}$ )
12         $\mathbf{z}$  = sig_layer.apply_fwd( $\mathbf{a}$ )
13         $\mathbf{b}$  = lin2_layer.apply_fwd( $\mathbf{z}$ ,  $\boldsymbol{\beta}$ )
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  $\boldsymbol{\alpha}$ , Tensor  $\boldsymbol{\beta}$ )
19        tape_bwd()
20        return lin1_layer.in_gradients[1], lin2_layer.in_gradients[1]
```

# 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
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
19        tape_bwd()
20        return lin1_layer.in_gradients[1], lin2_la
```

```
1 global tape = stack()
2
3 class Module:
4
5     method init()
6         out_tensor = null
7         out_gradient = 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 x, Tensor y, Tensor
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
19        tape_bwd()
20        return lin1_layer.in_gradients[1], lin2_la
```

```
1 global tape = stack()
2
3 class Module:
4
5     method init()
6         out_tensor = null
7         out_gradient = 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     loss_fn = nn.CrossEntropyLoss()
20     optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
21
22     # Compute prediction error
23     pred = model(X)
24     loss = loss_fn(pred, y)
25
26     # Backpropagation
27     optimizer.zero_grad()
28     loss.backward()
29     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
```

# 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

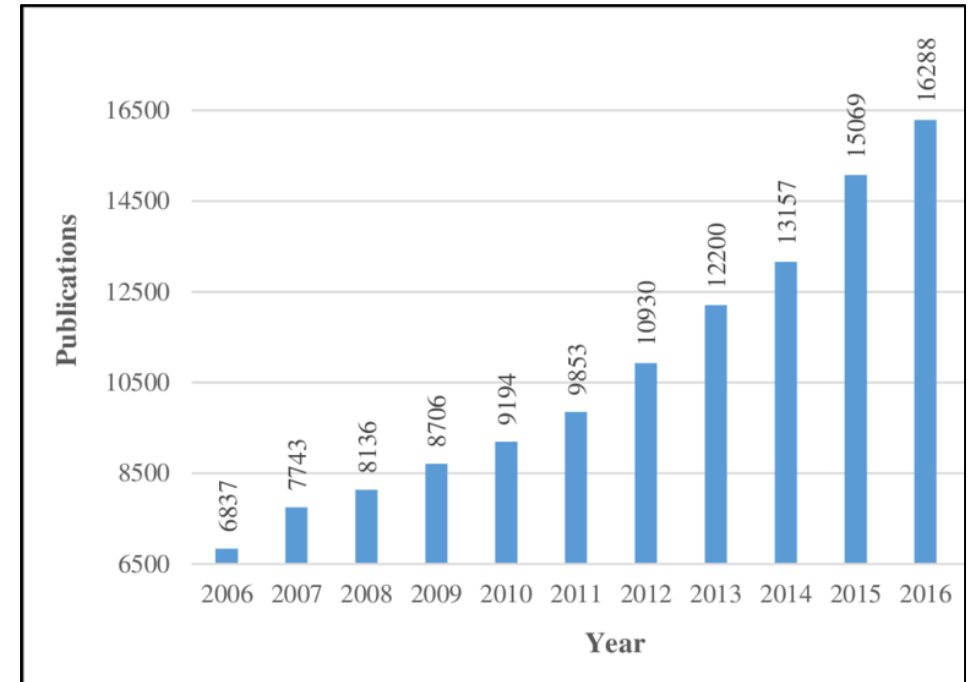
## Language Modeling

- 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

# **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 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*

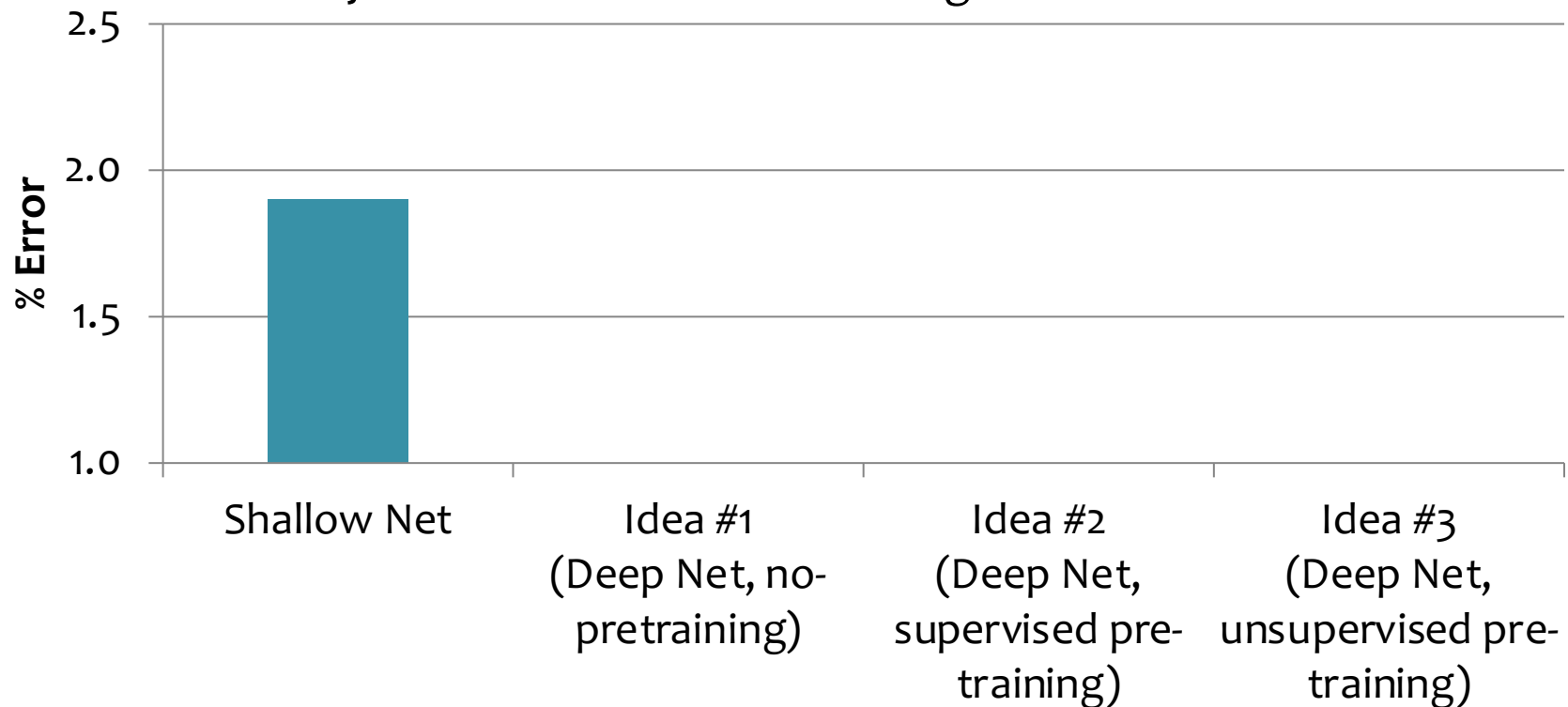
- unsupervised pre-training by maximizing likelihood of a large set of unlabeled sentences such as...
- The Pile (800 Gb of text)
- Dolma (3 trillion tokens)

### *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

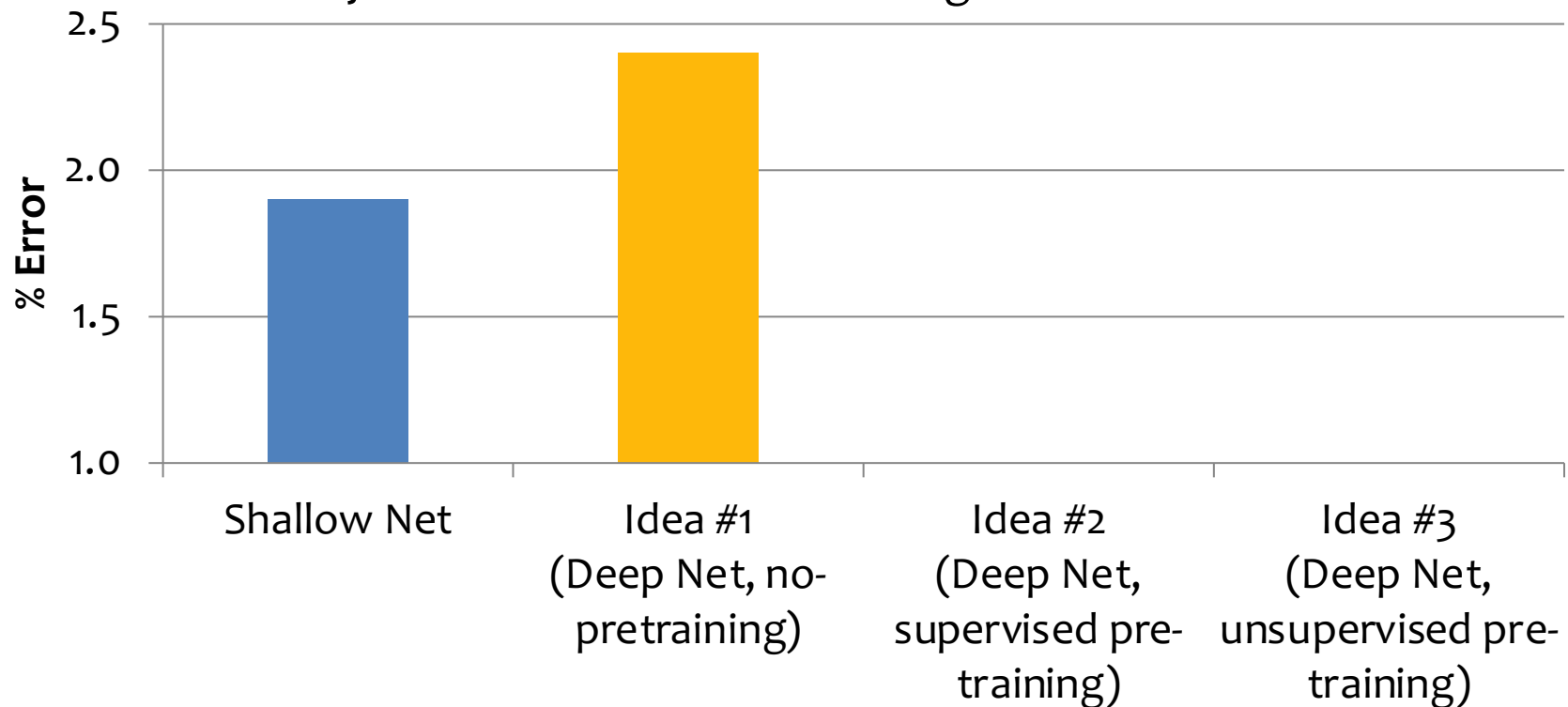
# Pre-Training and Fine-Tuning on MNIST

- 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

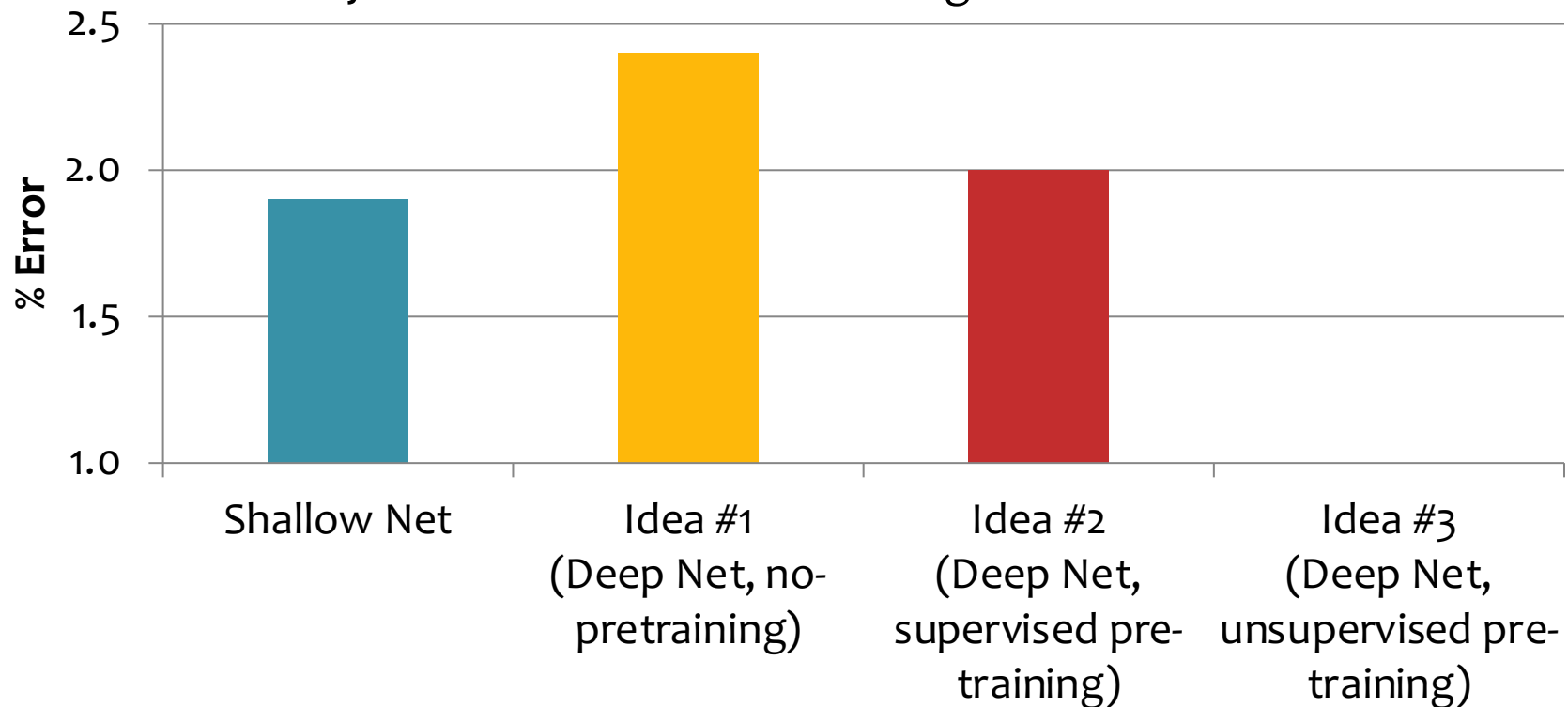
- Results from Bengio et al. (2006) on MNIST digit classification task
- Percent error (lower is better)
- Some methods first do pre-training
- Every method includes fine-tuning on labeled data





# Pre-Training and Fine-Tuning on MNIST

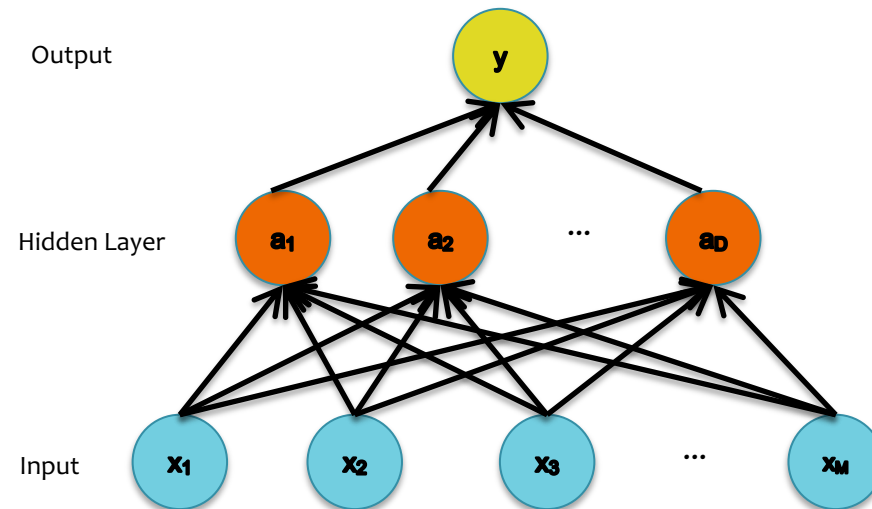
- Results from Bengio et al. (2006) on MNIST digit classification task
- Percent error (lower is better)
- Some methods first do pre-training
- Every method includes fine-tuning on labeled data



# Unsupervised Autoencoder Pre-Training for Vision

## Unsupervised pre-training of the first layer:

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

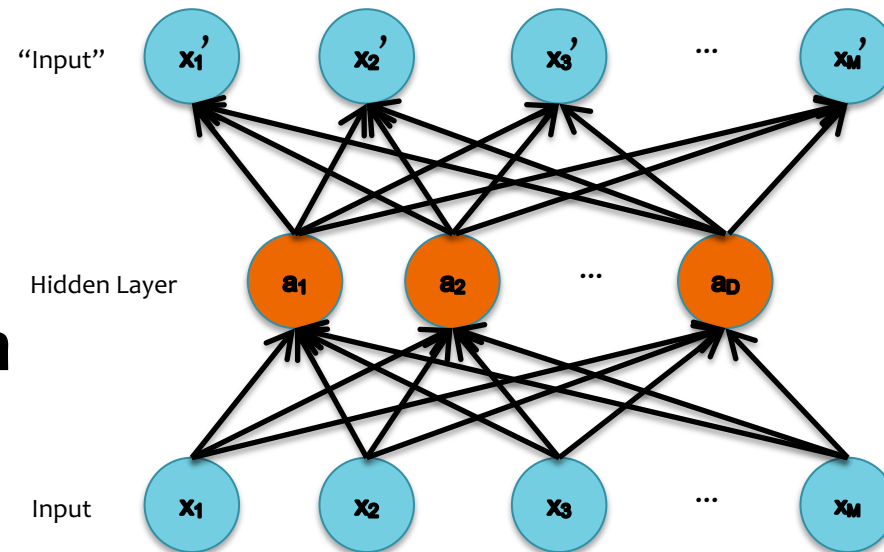


# Unsupervised Autoencoder Pre-Training for Vision

## Unsupervised pre-training of the first layer:

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

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



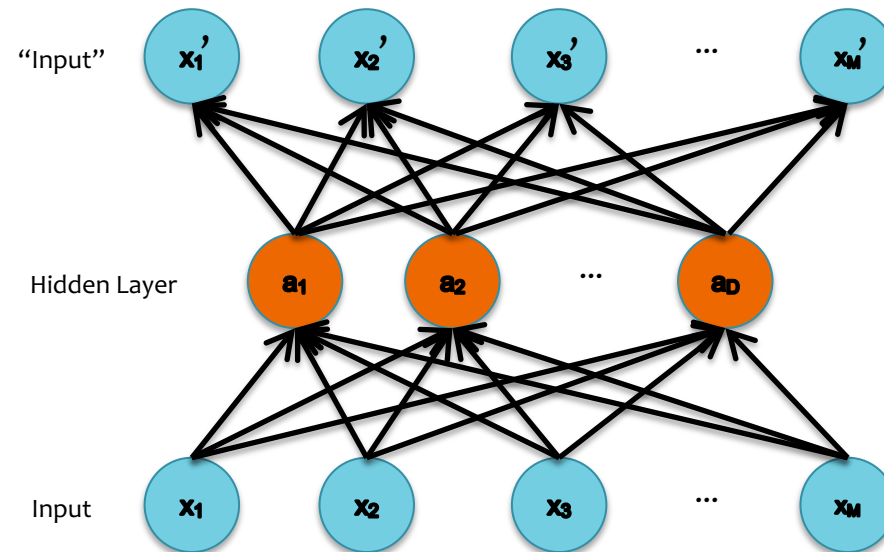
# Unsupervised Autoencoder Pre-Training for Vision

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

- $x'$  is the *reconstruction* of  $x$
- 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.

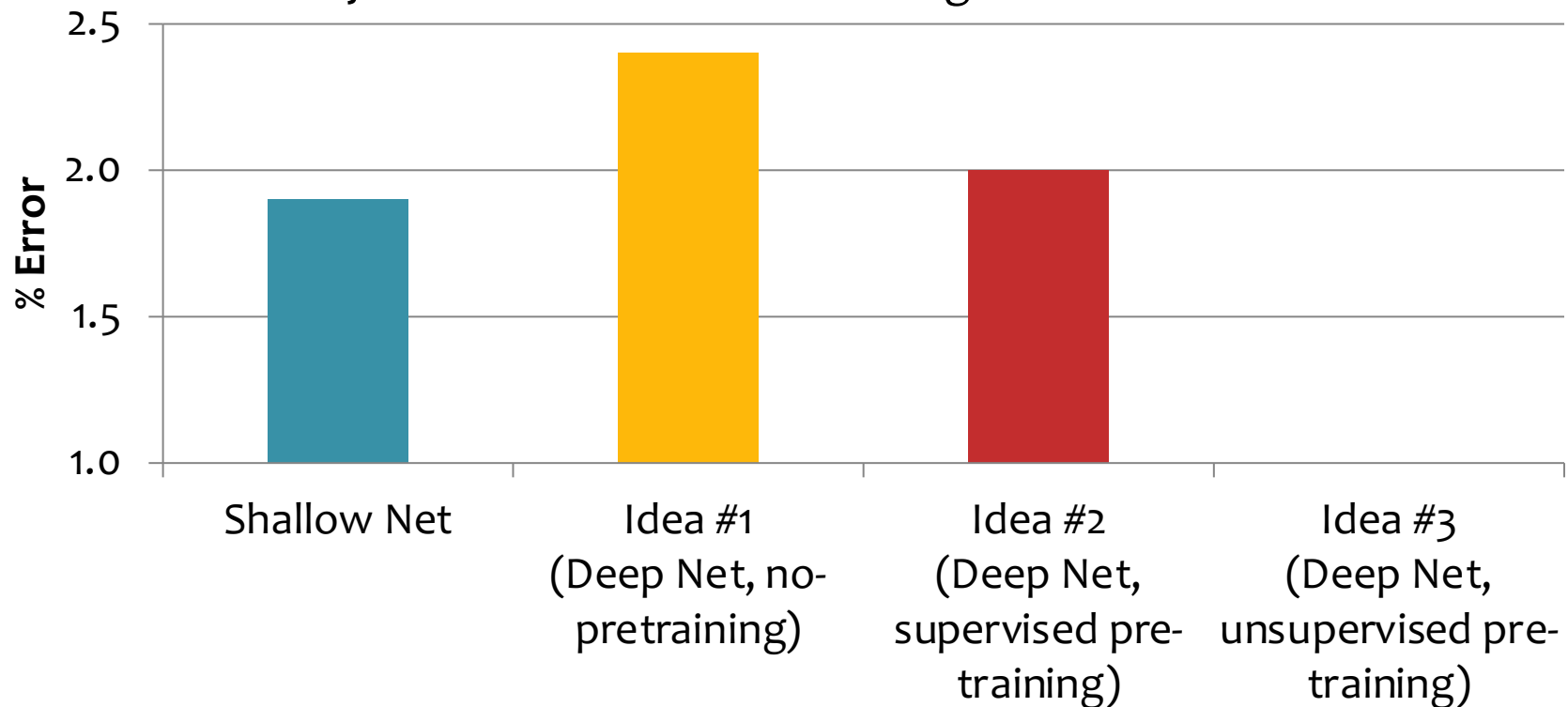
DECODER:  $x' = h(W'z)$

ENCODER:  $z = h(Wx)$



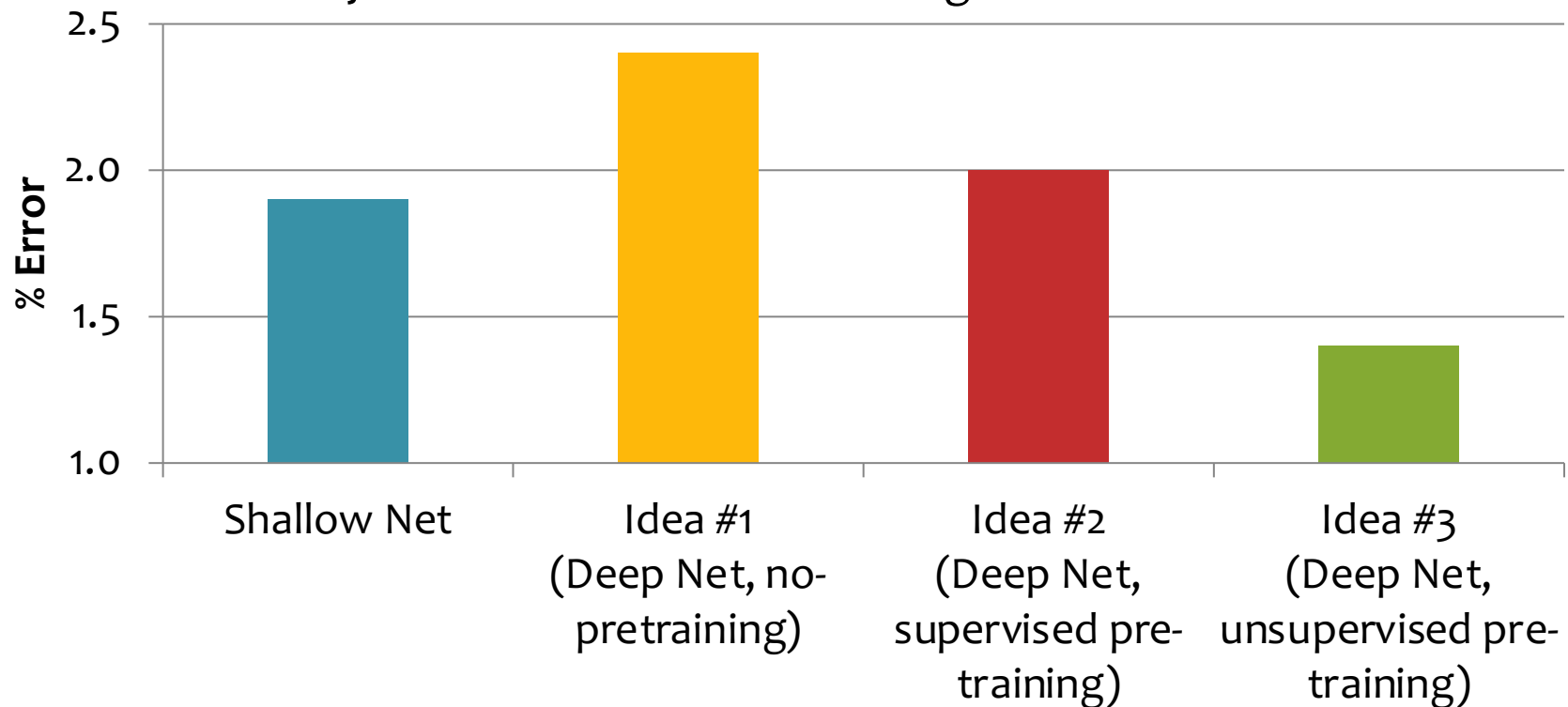
# Pre-Training and Fine-Tuning on MNIST

- 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



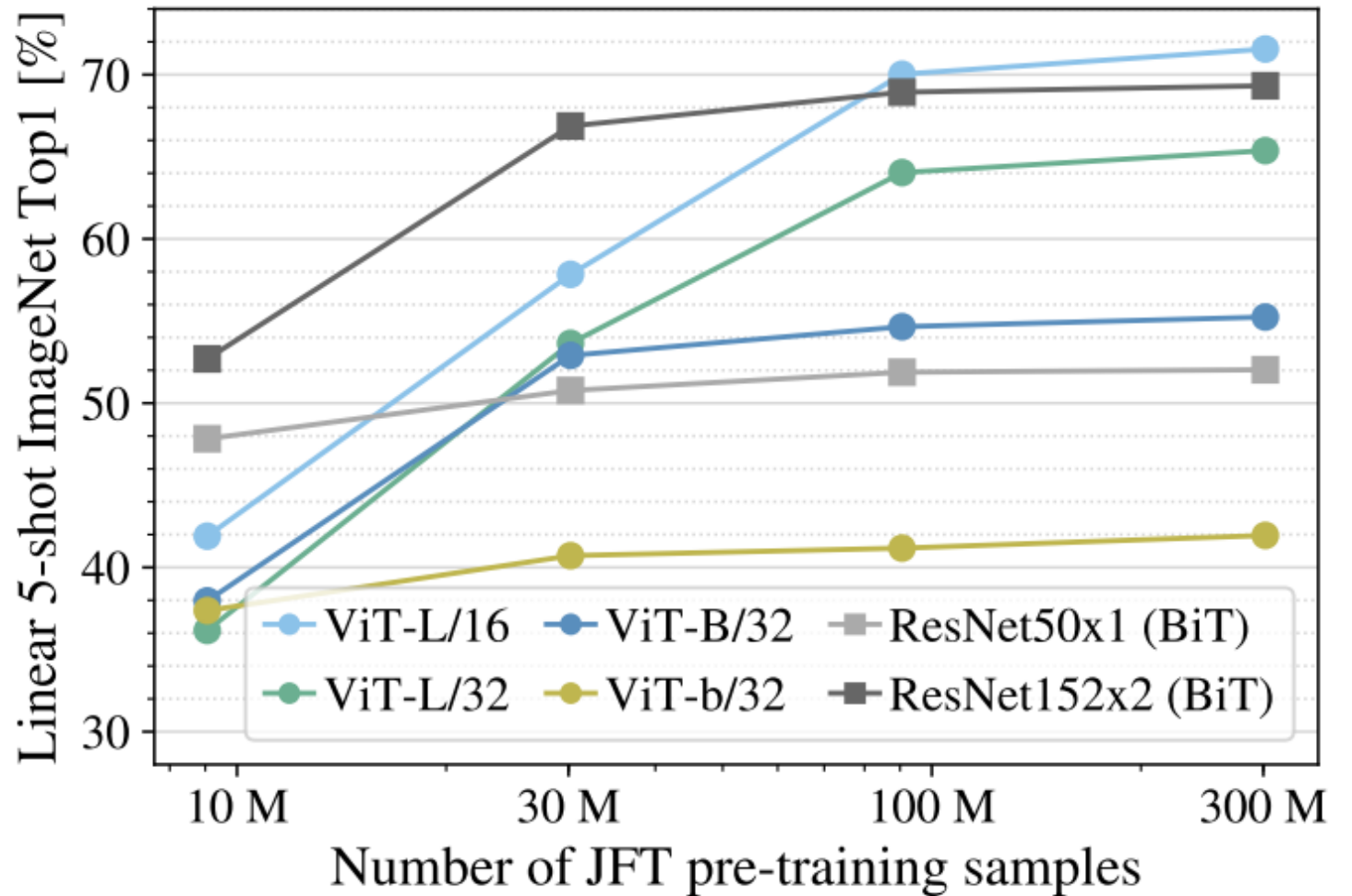
# Pre-Training and Fine-Tuning on MNIST

- 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



# 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*

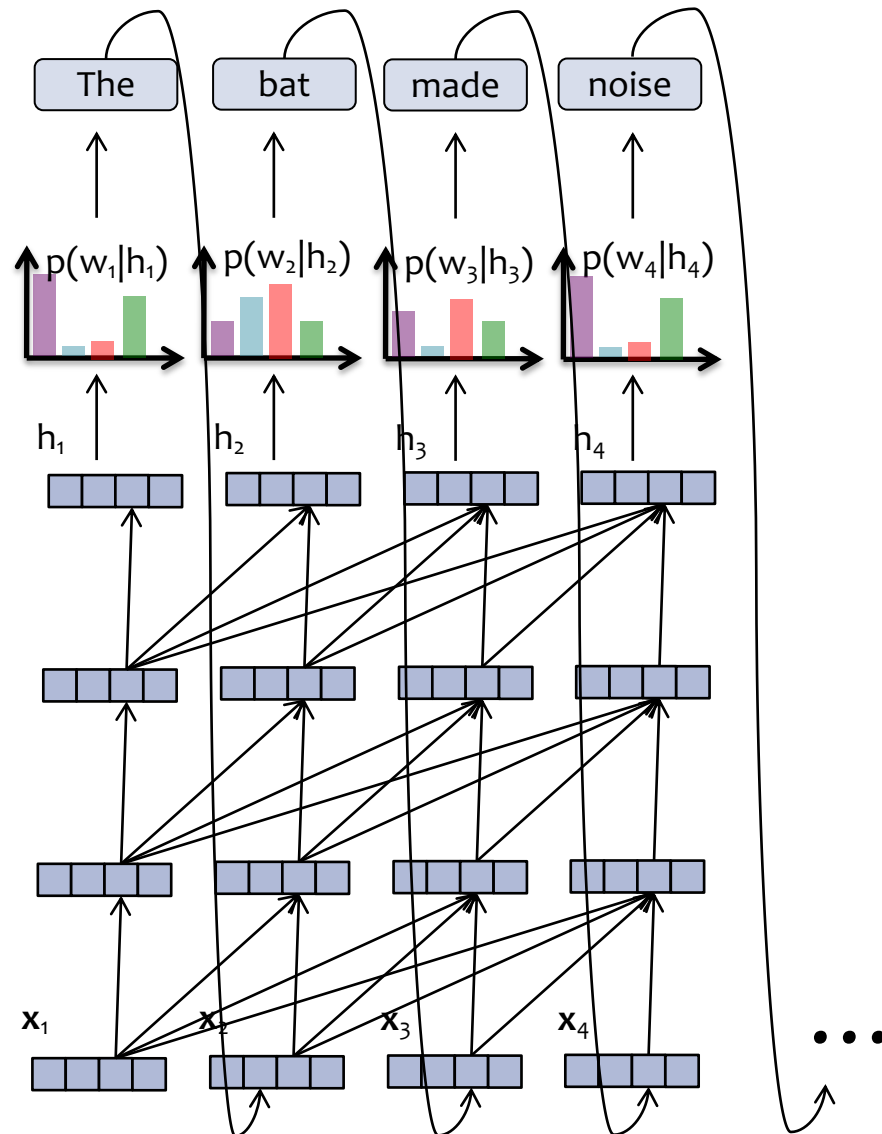
- unsupervised pre-training by maximizing likelihood of a large set of unlabeled sentences such as...
- The Pile (800 Gb of text)
- Dolma (3 trillion tokens)

### *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



# 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

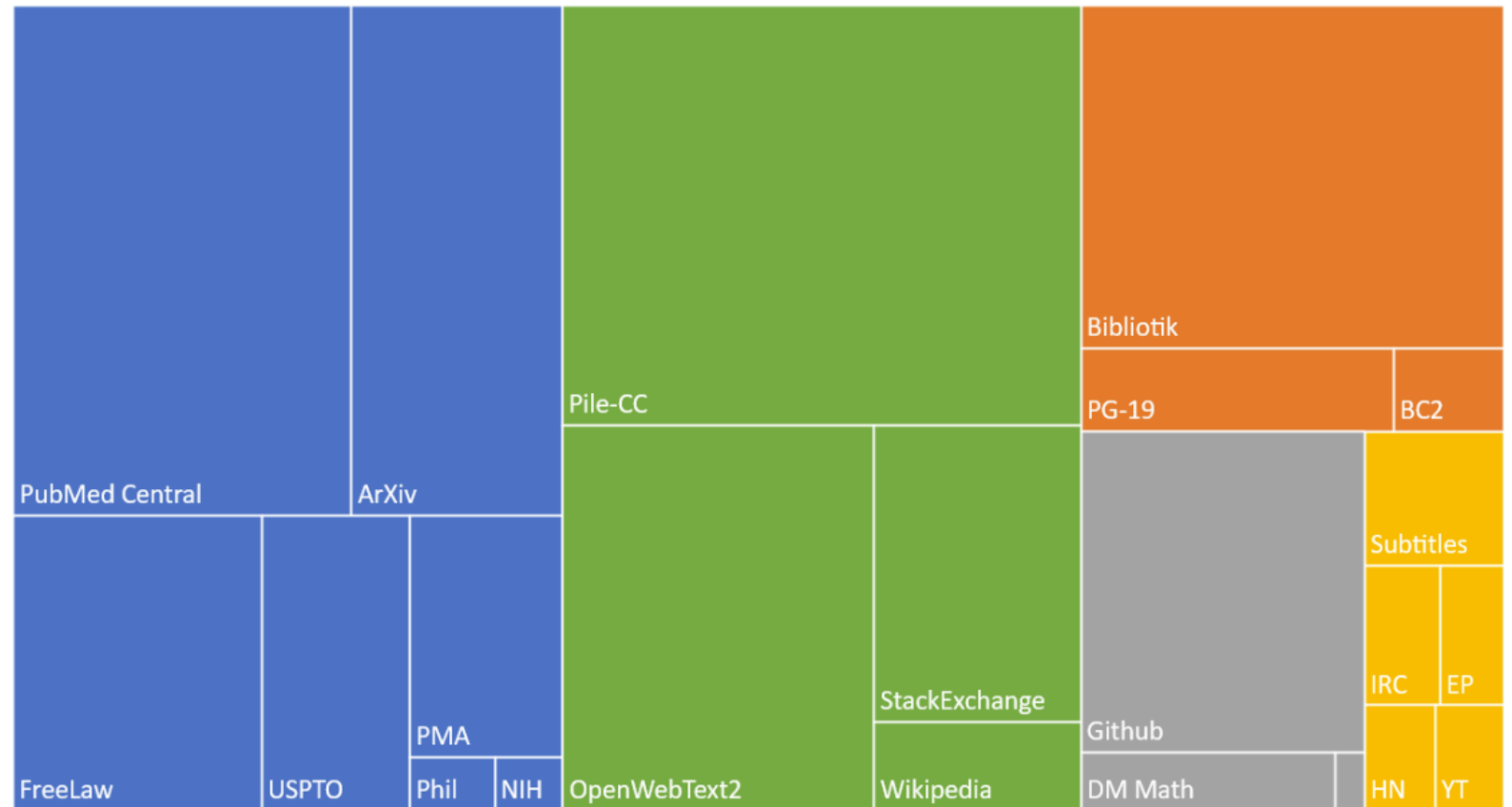
# Training Data for LLMs

## 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



# PROMPTING

# Zero-shot vs. Few-Shot

- **Definition:** in **zero-shot learning** we assume that training data does not contain any examples of the labels that appear in the test data
- **Definition:** in **few-shot learning** we assume that training data contains a handful (maybe two, three, or four) examples of each label

# Prompting

- Language models are trained to maximize the likelihood of sequences in the training data
- Most condition on the previous tokens to generate the next tokens

**Definition:** An *autoregressive language model* defines a probability distribution over sequences  $\mathbf{x}_{1:T}$  of the form:

$$p(\mathbf{x}_{1:T}) = \prod_{t=1}^T p(x_t \mid x_1, \dots, x_{t-1})$$

- Key idea behind prompting: provide a prefix string to the such that its likely completion is the answer you want

# Prompting Examples

## GPT-3

Prompt  
(i.e. context)

The City  
BY C. P. CAVAFY  
TRANSLATED BY EDMUND KEELEY  
[Poem text omitted]

SOME TREES  
John Ashbery  
[Poem text omitted]

Shadows on the Way  
Wallace Stevens

Output  
(i.e. generated poem)

I must have shadows on the way  
If I am to walk I must have  
Each step taken slowly and alone  
To have it ready made

And I must think in lines of grey  
To have dim thoughts to be my guide  
Must look on blue and green  
And never let my eye forget  
That color is my friend  
And purple must surround me too

The yellow of the sun is no more  
Intrusive than the bluish snow  
That falls on all of us. I must have  
Grey thoughts and blue thoughts walk with me  
If I am to go away at all.

## Conditional Sampling:

To draw a sample, feed in a prompt (i.e. context) and then sample from the model conditioned on that prompt

translation

# Prompting Examples

Llama-2-70B

Prompt

El último dueño de esta propiedad había sido un hombre soltero, que alcanzó una muy avanzada edad, y que durante gran parte de su existencia tuvo en su hermana una fiel compañera y ama de casa.

English translation:

Model Output

The last owner of this property was a bachelor man who lived to a very advanced age, and who had his sister as a faithful companion and housekeeper for most of his life.

**Conditional Sampling:**

To draw a sample, feed in a prompt (i.e. context) and then sample from the model conditioned on that prompt

English source from Sense & Sensibility

The late owner of this estate was a single man, who lived to a very advanced age, and who for many years of his life, had a constant companion and housekeeper in his sister.



# Prompting Examples

## Llama-2-70B

Prompt

Rhodochiton is a genus of flowering plants within the family Plantaginaceae, native to southern Mexico and neighbouring Guatemala. They climb by means of twining leaf stalks. One of the three species, "Rhodochiton atrosanguineus", the purple bell vine, is grown as an ornamental plant. All three species are sometimes included in "Lophospermum".

Question: You can find the purple bell vine in more than one country. True, False, or Neither?

Model Output

Answer: True.

Explanation: The purple bell vine (Rhodochiton atrosanguineus) is native to southern Mexico and neighbouring Guatemala, which means it can be found in more than one country.

Correct Answer

True.

# Prompting Examples

## Llama-2-70B

Prompt

Story: Jason listened to the weather and heard it was going to be sunny. He thought the kids might like to go swimming. He gathered up the swimsuits, towels and sunscreen. Jason and the kids got into the truck and drove to the beach. They spent the next 2 hours playing and splashing in the surf.

One-sentence Summary:

Model Output

Jason took the kids swimming at the beach after hearing the weather forecast, gathering necessary items and driving there.

# What else can a large LM (attempt to) do?

Using the idea of **prompts**, we can apply LMs to a **variety of different problems** in natural language processing.

In the **zero-shot setting**, we simply feed the context to the model and observe how it completes the sequence. (i.e. there is no additional training)

## Answer fact-based questions:

---

Context → Organisms require energy in order to do what?

---

Correct Answer → mature and develop.  
Incorrect Answer → rest soundly.  
Incorrect Answer → absorb light.  
Incorrect Answer → take in nutrients.

---

## Complete sentences logically:

---

Context → My body cast a shadow over the grass because

---

Correct Answer → the sun was rising.  
Incorrect Answer → the grass was cut.

---

## Complete analogies:

---

Context → lull is to trust as

---

Correct Answer → cajole is to compliance  
Incorrect Answer → balk is to fortitude  
Incorrect Answer → betray is to loyalty  
Incorrect Answer → hinder is to destination  
Incorrect Answer → soothe is to passion

---

## Reading comprehension:

---

Context → anli 1: anli 1: Fulton James MacGregor MSP is a Scottish politician who is a Scottish National Party (SNP) Member of Scottish Parliament for the constituency of Coatbridge and Chryston. MacGregor is currently Parliamentary Liaison Officer to Shona Robison, Cabinet Secretary for Health & Sport. He also serves on the Justice and Education & Skills committees in the Scottish Parliament.  
Question: Fulton James MacGregor is a Scottish politician who is a Liaison officer to Shona Robison who he swears is his best friend. True, False, or Neither?

---

Correct Answer → Neither  
Incorrect Answer → True  
Incorrect Answer → False

---

# Zero-shot LLMs

- GPT-2 (1.5B parameters) for unsupervised prediction on various tasks
- GPT-2 models  $p(\text{output} \mid \text{input}, \text{task})$ 
  - translation: (*translate to french, english text, french text*)
  - reading comprehension: (*answer the question, document, question, answer*)
- Why does this work?

---

“I’m not the cleverest man in the world, but like they say in French: **Je ne suis pas un imbecile [I’m not a fool].**”

In a now-deleted post from Aug. 16, Soheil Eid, Tory candidate in the riding of Joliette, wrote in French: “**Mentez mentez, il en restera toujours quelque chose,**” which translates as, “**Lie lie and something will always remain.**”

“I hate the word ‘**perfume,**’” Burr says. ‘It’s somewhat better in French: ‘**parfum.**’

If listened carefully at 29:55, a conversation can be heard between two guys in French: “-**Comment on fait pour aller de l’autre coté? -Quel autre coté?**”, which means “- **How do you get to the other side? - What side?**”.

If this sounds like a bit of a stretch, consider this question in French: **As-tu aller au cinéma?**, or **Did you go to the movies?**, which literally translates as Have-you to go to movies/theater?

“**Brevet Sans Garantie Du Gouvernement**”, translated to English: “**Patented without government warranty**”.

---

*Table 1.* Examples of naturally occurring demonstrations of English to French and French to English translation found throughout the WebText training set.

# Zero-shot LLMs

- GPT-2 (1.5B parameters) for unsupervised prediction on various tasks
- GPT-2 models  $p(\text{output} \mid \text{input}, \text{task})$ 
  - translation: (translate to french, english text, french text)
  - reading comprehension: (answer the question, document, question, answer)
- Why does this work?

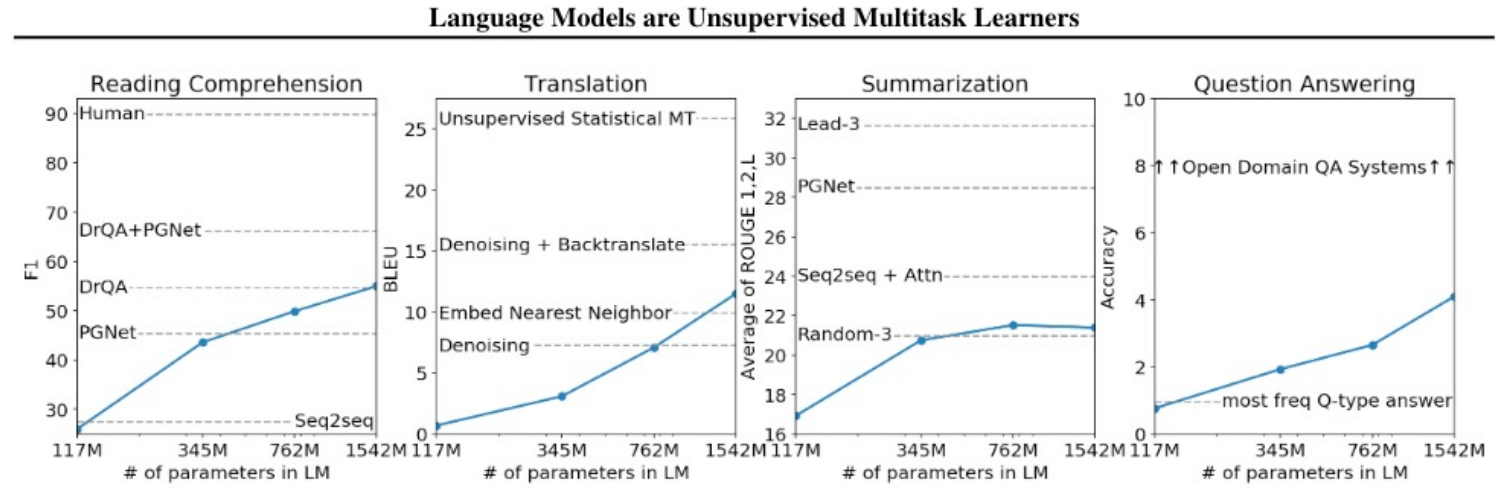


Figure 1. Zero-shot task performance of WebText LMs as a function of model size on many NLP tasks. Reading Comprehension results are on CoQA (Reddy et al., 2018), translation on WMT-14 Fr-En (Artetxe et al., 2017), summarization on CNN and Daily Mail (See et al., 2017), and Question Answering on Natural Questions (Kwiatkowski et al., 2019). Section 3 contains detailed descriptions of each result.

	LAMBADA (PPL)	LAMBADA (ACC)	CBT-CN (ACC)	CBT-NE (ACC)	WikiText2 (PPL)	PTB (PPL)	enwik8 (BPB)	text8 (BPC)	WikiText103 (PPL)	1BW (PPL)
SOTA	99.8	59.23	85.7	82.3	39.14	46.54	0.99	1.08	18.3	<b>21.8</b>
117M	<b>35.13</b>	45.99	<b>87.65</b>	<b>83.4</b>	<b>29.41</b>	65.85	1.16	1.17	37.50	75.20
345M	<b>15.60</b>	55.48	<b>92.35</b>	<b>87.1</b>	<b>22.76</b>	47.33	1.01	<b>1.06</b>	26.37	55.72
762M	<b>10.87</b>	<b>60.12</b>	<b>93.45</b>	<b>88.0</b>	<b>19.93</b>	<b>40.31</b>	<b>0.97</b>	<b>1.02</b>	22.05	44.575
1542M	<b>8.63</b>	<b>63.24</b>	<b>93.30</b>	<b>89.05</b>	<b>18.34</b>	<b>35.76</b>	<b>0.93</b>	<b>0.98</b>	<b>17.48</b>	42.16

Table 3. Zero-shot results on many datasets. No training or fine-tuning was performed for any of these results. PTB and WikiText-2 results are from (Gong et al., 2018). CBT results are from (Bajgar et al., 2016). LAMBADA accuracy result is from (Hoang et al., 2018) and LAMBADA perplexity result is from (Grave et al., 2016). Other results are from (Dai et al., 2019).

# **IN-CONTEXT LEARNING**

# Few-shot Learning with LLMs

Suppose you have...

- a dataset  $D = \{(x_i, y_i)\}_{i=1}^N$  and  $N$  is rather small (i.e. few-shot setting)
- a very large (billions of parameters) pre-trained language model

There are two ways to “learn”

This section!



## Option A: Supervised fine-tuning

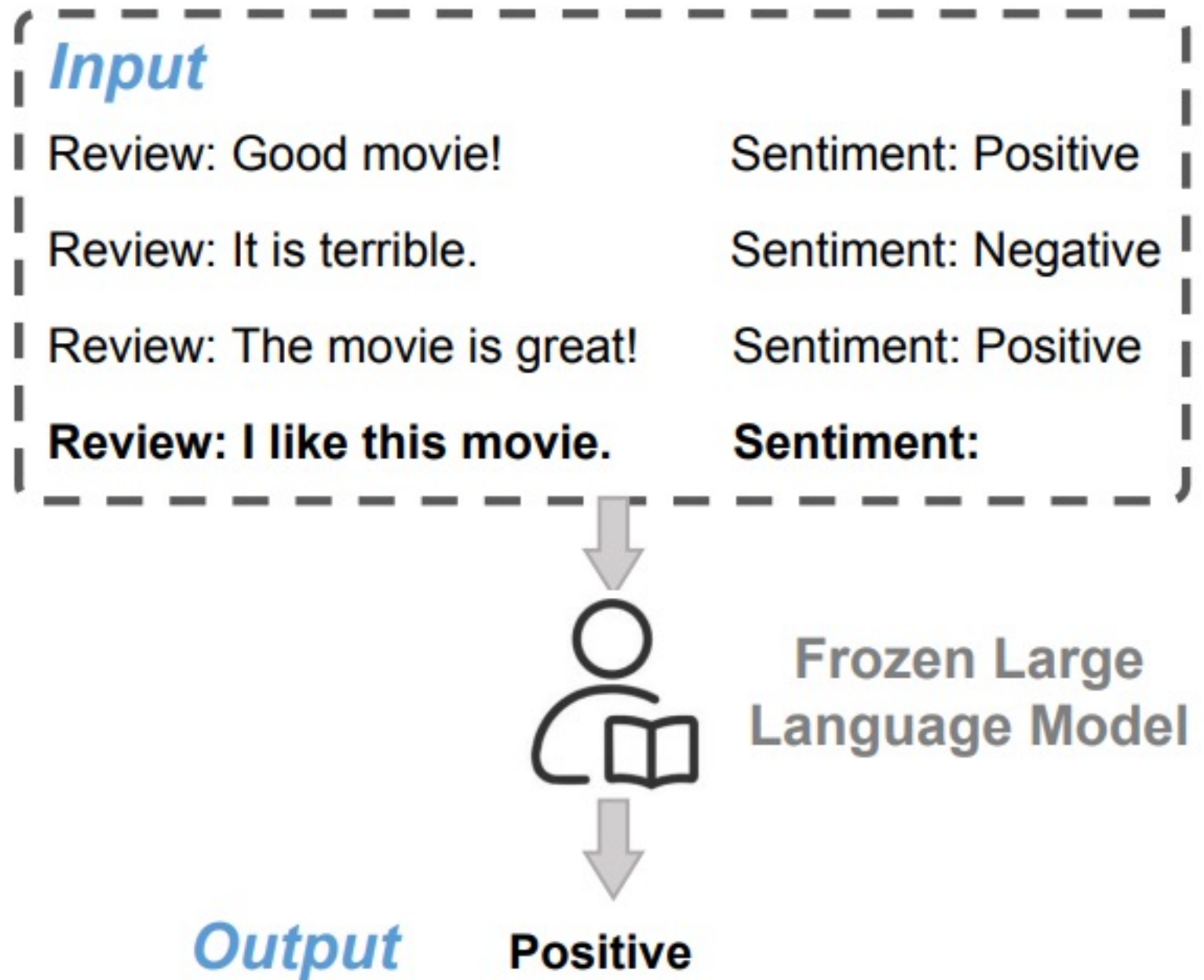
- **Definition:** fine-tune the LLM on the training data using...
  - a standard supervised objective
  - backpropagation to compute gradients
  - your favorite optimizer (e.g. Adam)
- **Pro:** fits into the standard ML recipe
- **Pro:** still works if  $N$  is large
- **Con:** backpropagation requires  $\sim 3x$  the memory and computation time as the forward computation
- **Con:** you might not have access to the model weights at all (e.g. because the model is proprietary)

## Option B: In-context learning

- **Definition:**
  1. feed training examples to the LLM as a prompt
  2. allow the LLM to infer patterns in the training examples during inference (i.e. decoding)
  3. take the output of the LLM following the prompt as its prediction
- **Con:** the prompt may be very long and Transformer LMs require  $O(N^2)$  time/space where  $N$  = length of context
- **Pro:** no backpropagation required and only one pass through the training data
- **Pro:** does not require model weights, only API access

# Few-shot In-context Learning

- Few-shot learning can be done via in-context learning
- Typically, a task description is presented first
- Then a sequence of input/output pairs from a training dataset are presented in sequence





# Few-shot In-context Learning

- Few-shot learning can be done via in-context learning
- Typically, a task description is presented first
- Then a sequence of input/output pairs from a training dataset are presented in sequence

The three settings we explore for in-context learning

## Zero-shot

The model predicts the answer given only a natural language description of the task. No gradient updates are performed.

```
1 Translate English to French: ← task description
2 cheese => ..... ← prompt
```

## One-shot

In addition to the task description, the model sees a single example of the task. No gradient updates are performed.

```
1 Translate English to French: ← task description
2 sea otter => loutre de mer ← example
3 cheese => ..... ← prompt
```

## Few-shot

In addition to the task description, the model sees a few examples of the task. No gradient updates are performed.

```
1 Translate English to French: ← task description
2 sea otter => loutre de mer ← examples
3 peppermint => menthe poivrée ←
4 plush girafe => girafe peluche ←
5 cheese => ..... ← prompt
```

Traditional fine-tuning (not used for GPT-3)

## Fine-tuning

The model is trained via repeated gradient updates using a large corpus of example tasks.

