10-301/10-601 Introduction to Machine Learning

Machine Learning Department
School of Computer Science
Carnegie Mellon University

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

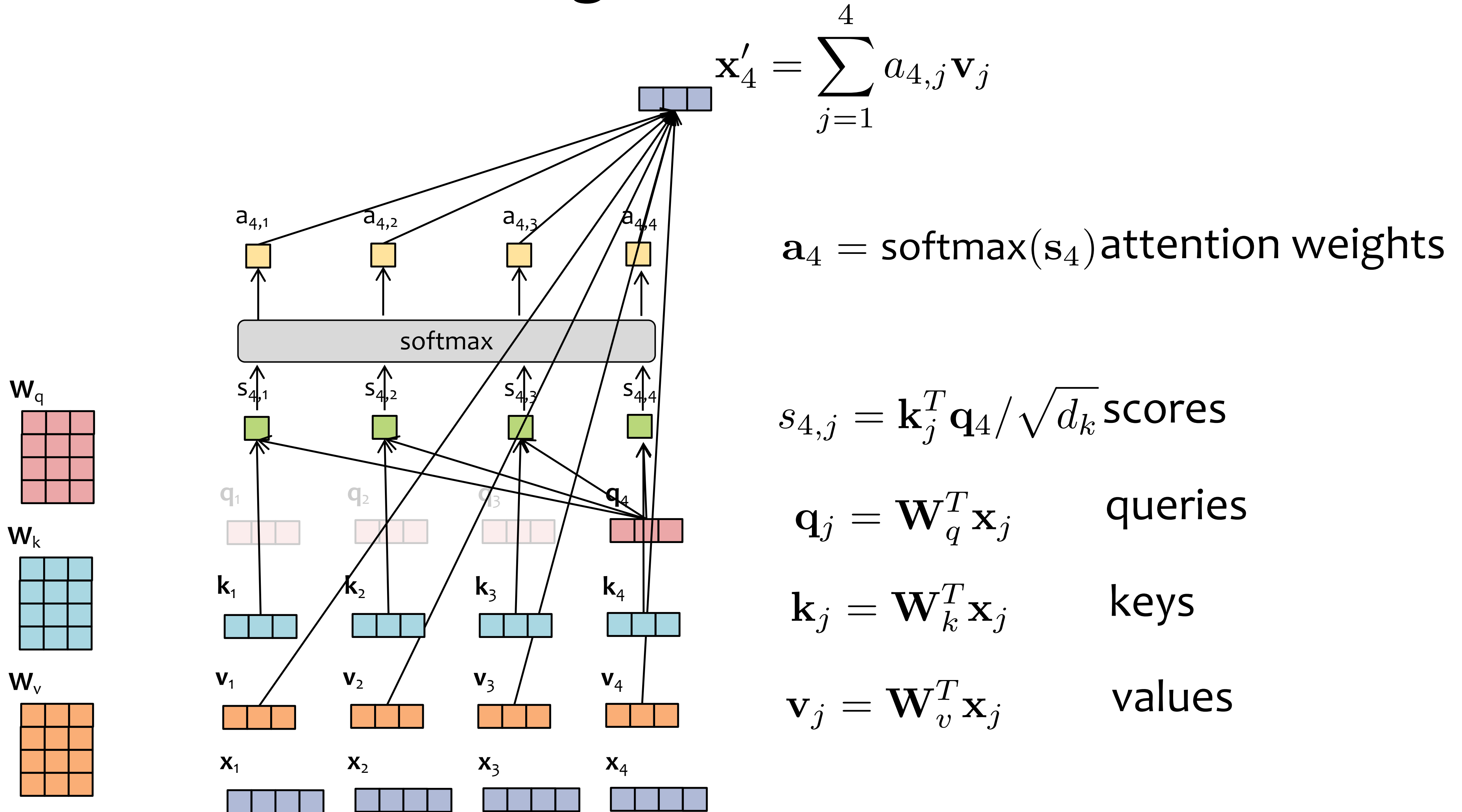
Matt Gormley & Henry Chai
Lecture 19
Mar. 27, 2024

Reminders

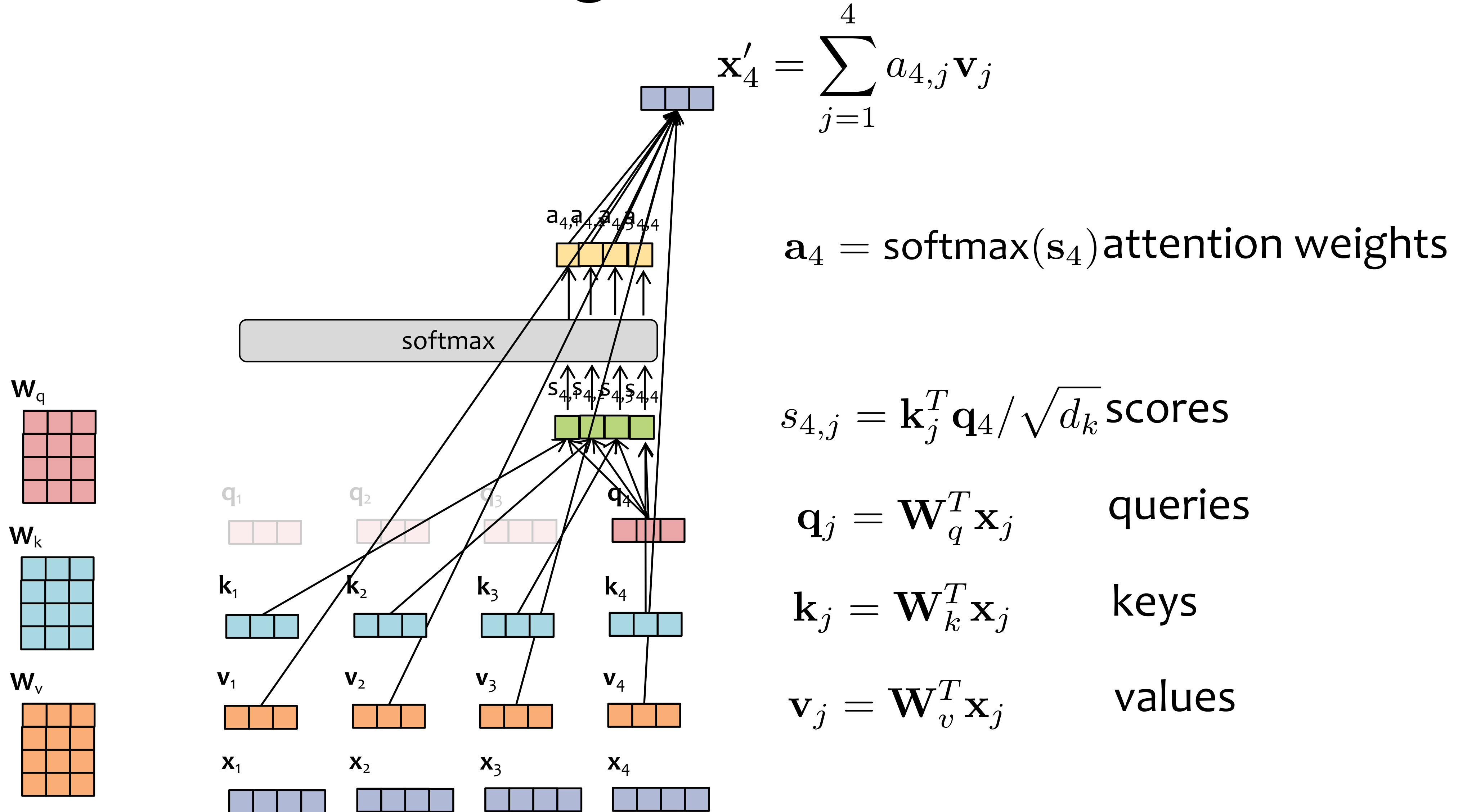
- **Exam 2: Thu, Nov 7, 6:45 pm - 8:45 pm**
- **Homework 7: Deep Learning & LLMs**
  - **Out: Thu, Nov 7**
  - **Due: Sun, Nov 17, 11:59pm**

IMPLEMENTING A TRANSFORMER LM

Matrix Version of Single-Headed Attention

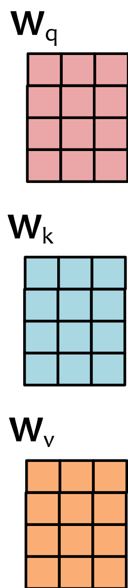


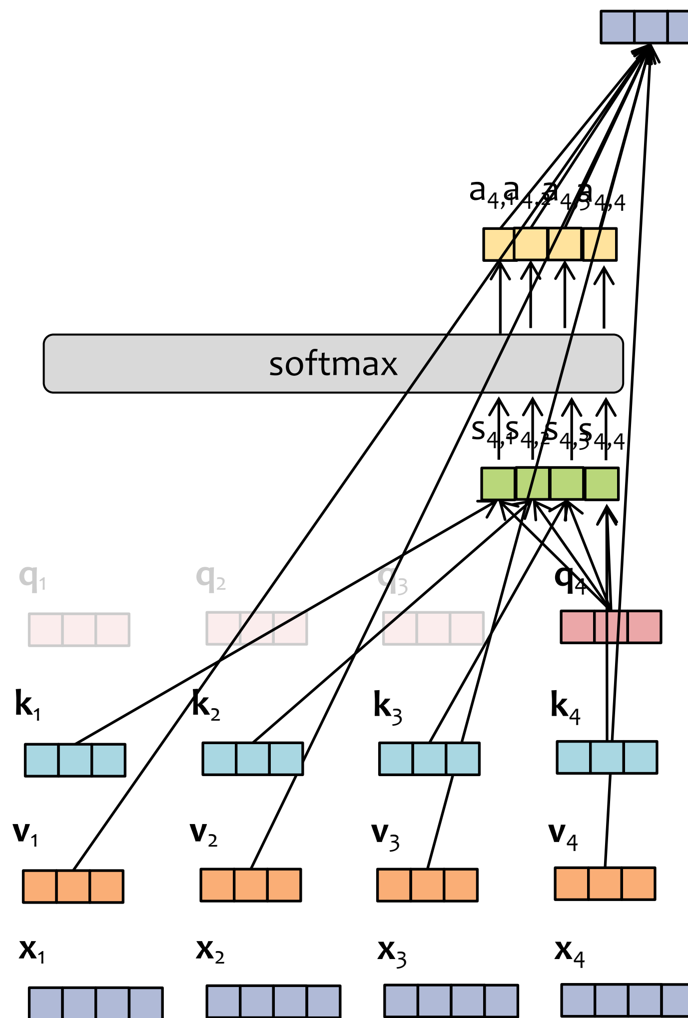
Matrix Version of Single-Headed Attention



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





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

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

$\in \mathbb{R}^{N \times N}$

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

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

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

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

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

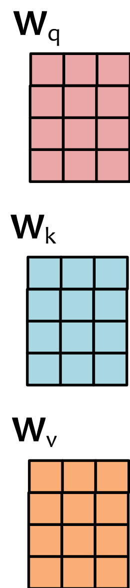
$\in \mathbb{R}^{N \times d_k}$

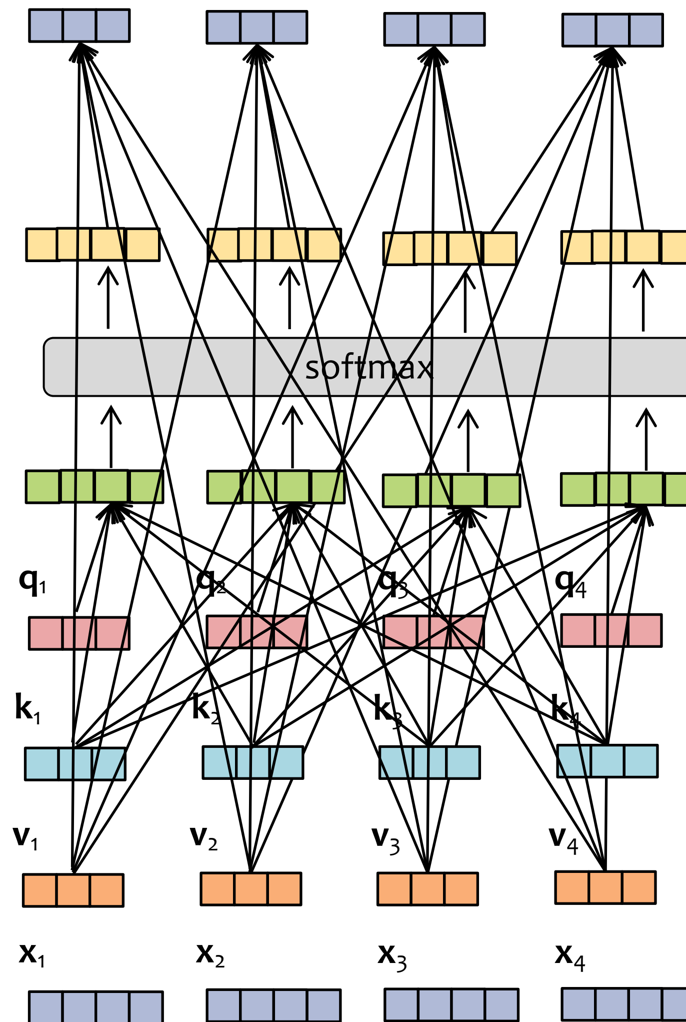
\# rows = N

\# cols = d_{in}

Matrix Version of Single-Headed Attention

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





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

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

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

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

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

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

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

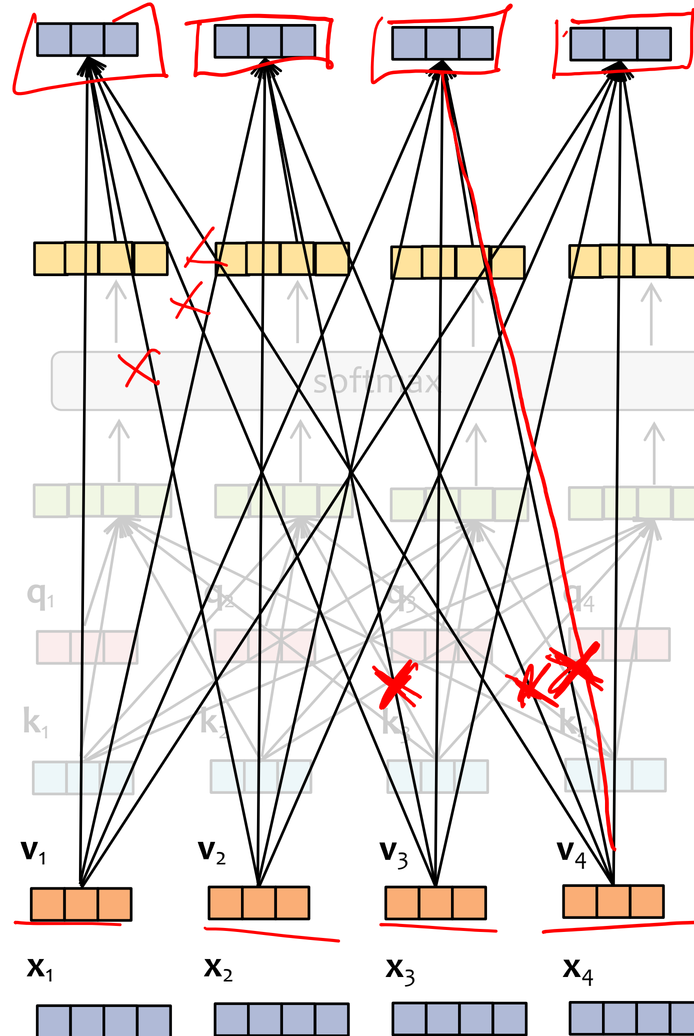
Matrix Version of Single-Headed Attention

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

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

So what is this model?

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



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

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

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

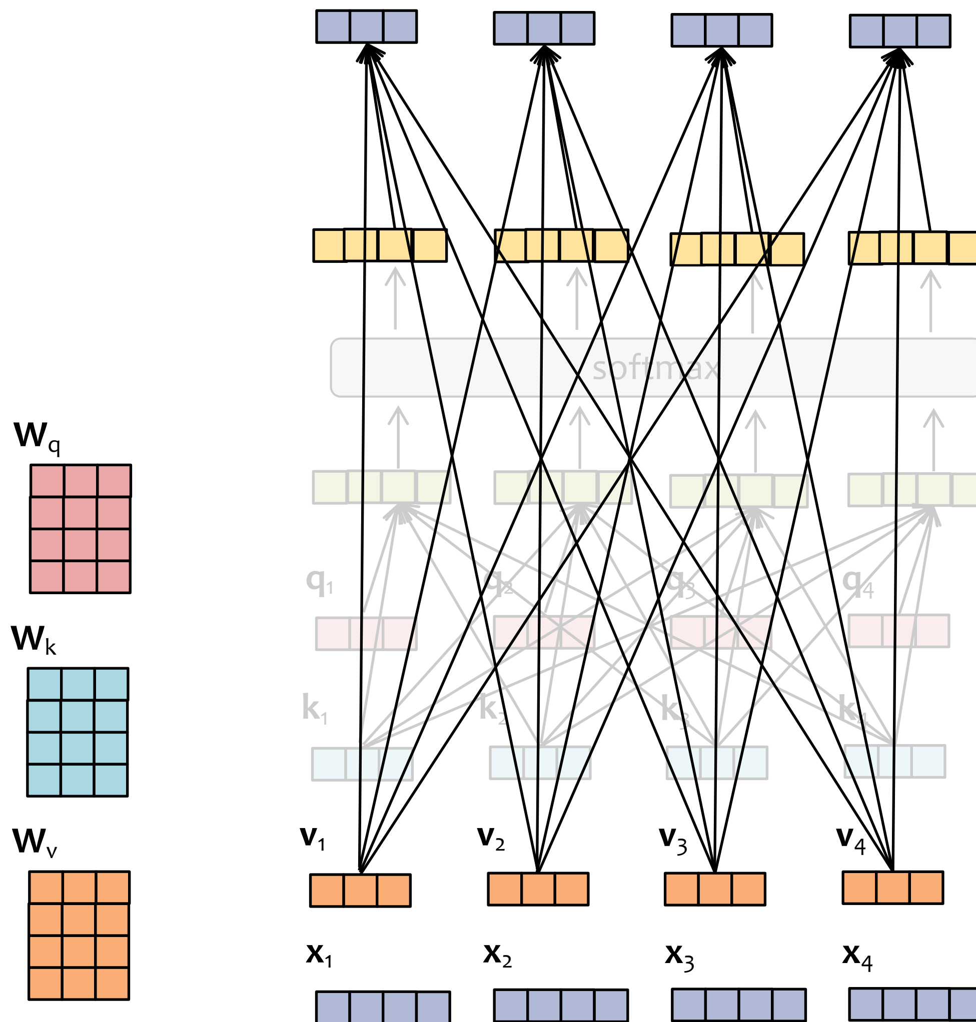
$$\mathbf{Q} = [\mathbf{q}_1, \ldots, \mathbf{q}_4]^T = \mathbf{X}\mathbf{W}_q$$

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

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

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

Matrix Version of Single-Headed Attention



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

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

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

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

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

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

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

Q1

Question: How is the softmax applied?

A. column-wise 44%

B. row-wise 56%

C = toxic

Answer:

Matrix Version of Single-Headed (Causal) Attention

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

Q2

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: 65%

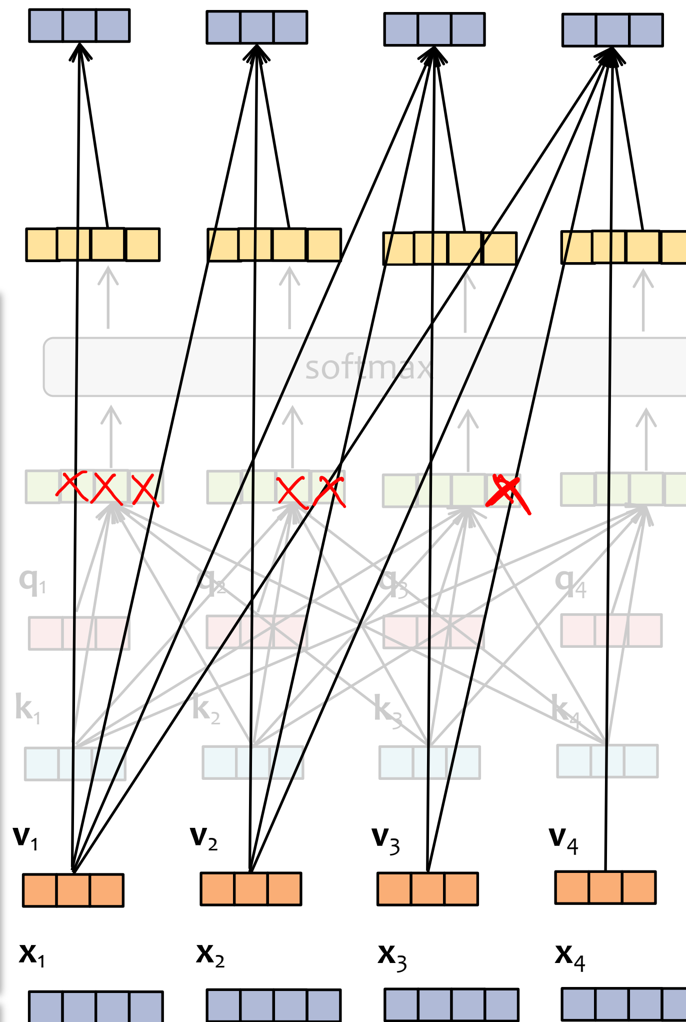
$$\mathbf{M} = \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

C:

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

D = toxic

Answer:



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

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

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

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

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

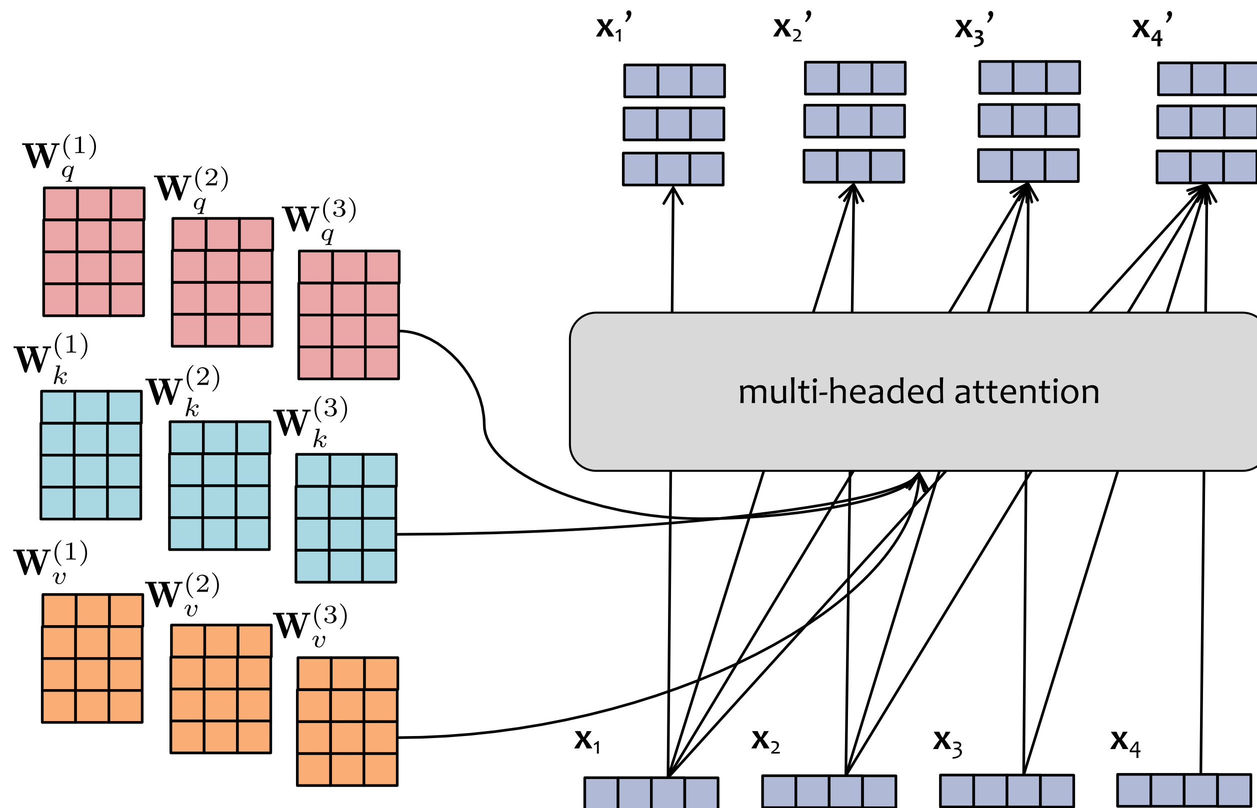
$$\mathbf{V} = \mathbf{X}\mathbf{W}_v$$

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

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

Matrix Version of Multi-Headed (Causal) Attention

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



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

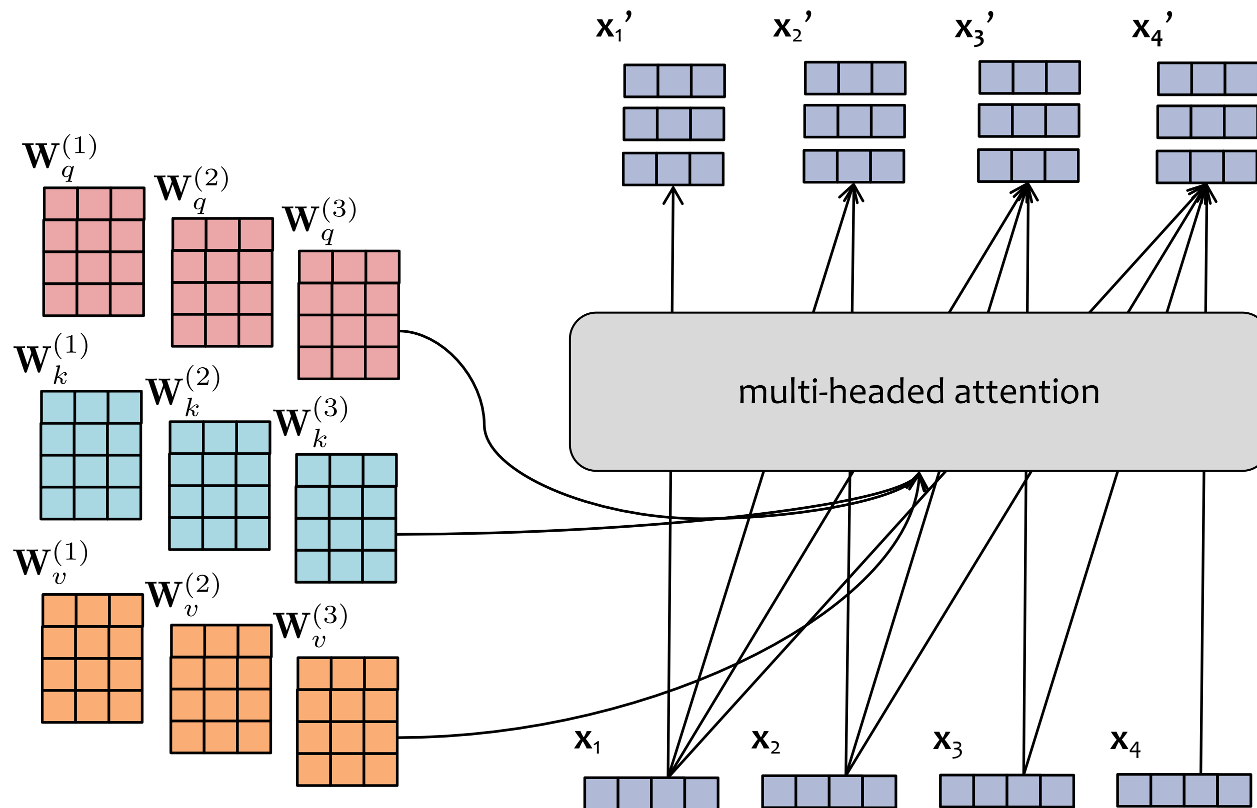
$$\mathbf{Q}^{(i)} = \mathbf{X}\mathbf{W}_q^{(i)}$$

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

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

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

Matrix Version of Multi-Headed (Causal) Attention



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

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

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

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

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

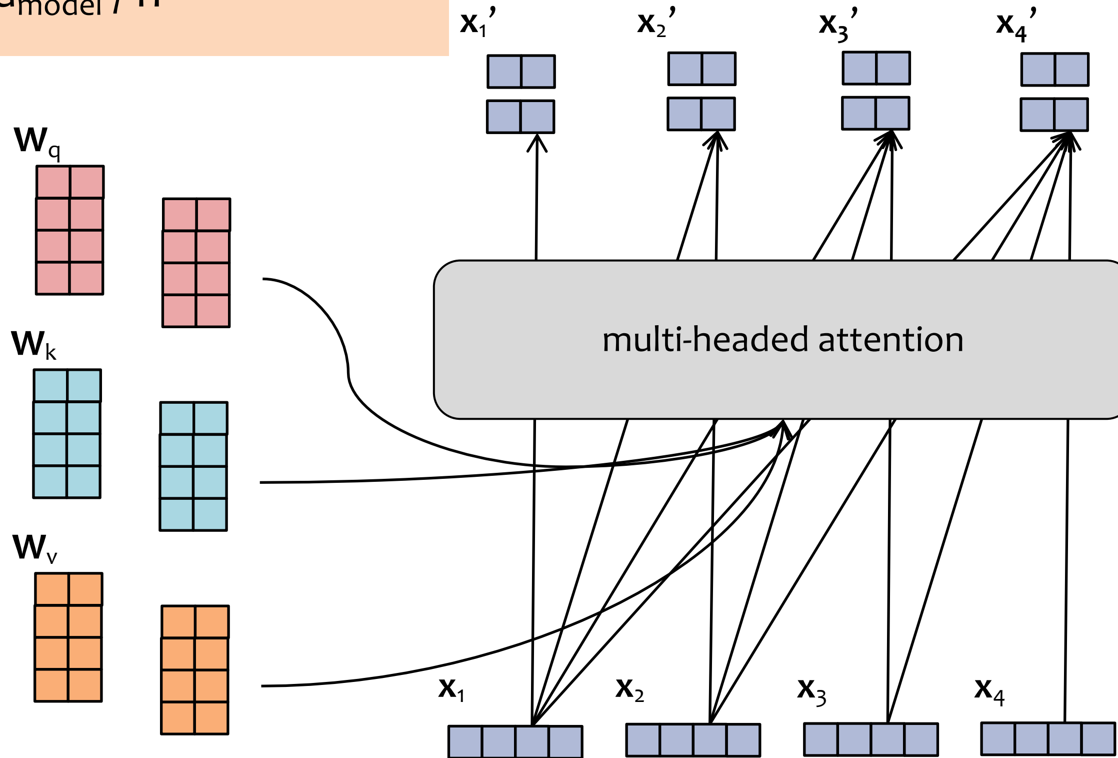
$$\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_4]^T$$

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

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

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

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



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

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

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

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

PRACTICALITIES OF TRANSFORMER LMS

In-Class Poll

Q3

Question:

Suppose we have the following input embeddings and attention weights:

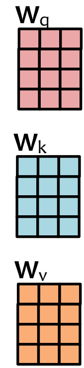
- $x_1 = [1,0,0,0]$ $a_{4,1} = 0.1$
- $x_2 = [0,1,0,0]$ $a_{4,2} = 0.2$
- $x_3 = [0,0,2,0]$ $a_{4,3} = 0.6$
- $x_4 = [0,0,0,1]$ $a_{4,4} = 0.1$

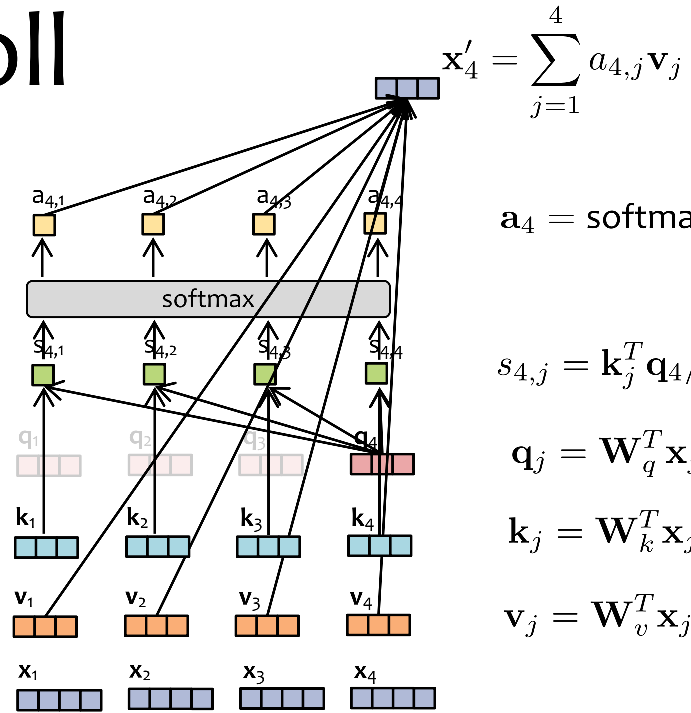
And $W_v = I$. Then we can compute x_4'.

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'?





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

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

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

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

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

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

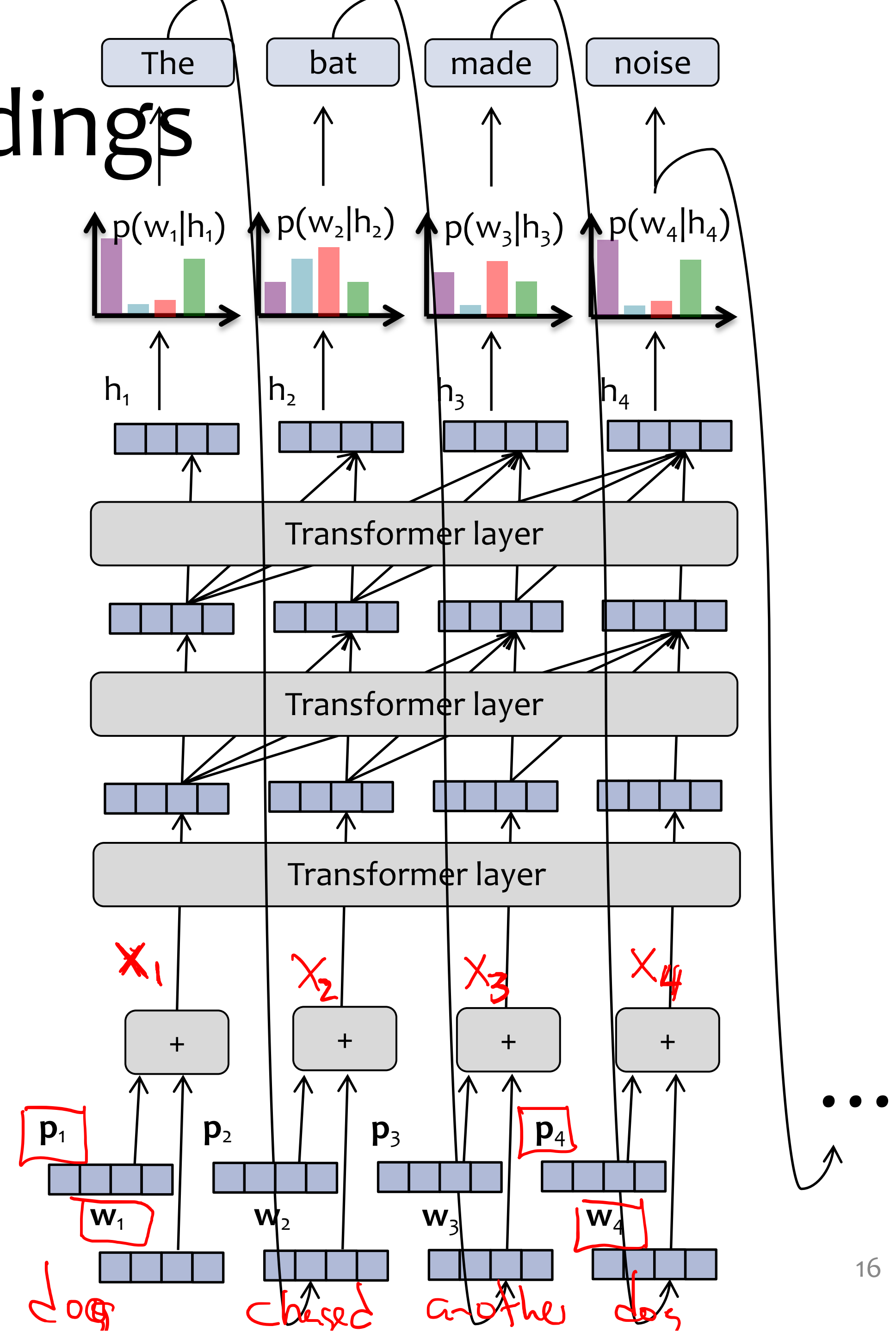
Answer:

exactly the same as before

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)

if $w_1 = w_4$
does not imply $x_1 = x_4$



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_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	<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_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}
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)

Slide adapted from Henry Chai

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”

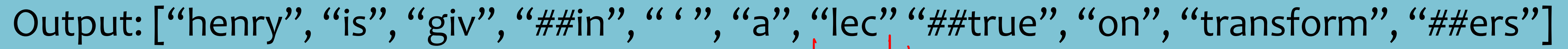
Output: [“h”, “e”, “n”, “r”, “y”, “i”, “s”, “g”, “i”, “v”, “i”, “n”, “ ’ ”, …]

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:



Pros/Cons:

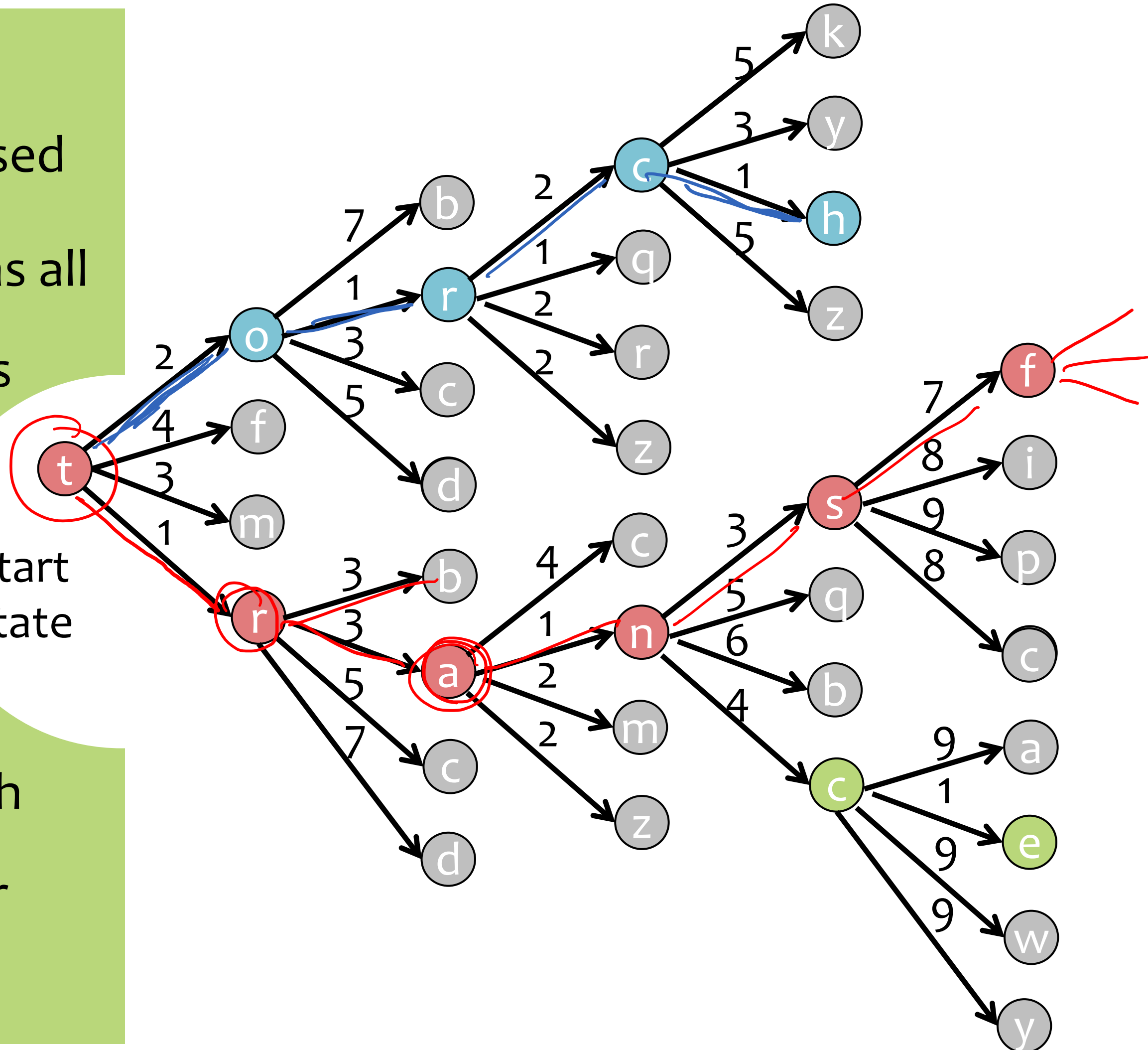
- Split long or rare words into smaller, semantically meaningful components or subwords
- No out-of-vocabulary words – any non-subword token can be constructed from other subwords (always includ all characters as subwords)
- Examples algorithms for learning a subword tokenization:
  - Byte-Pair-Encoding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

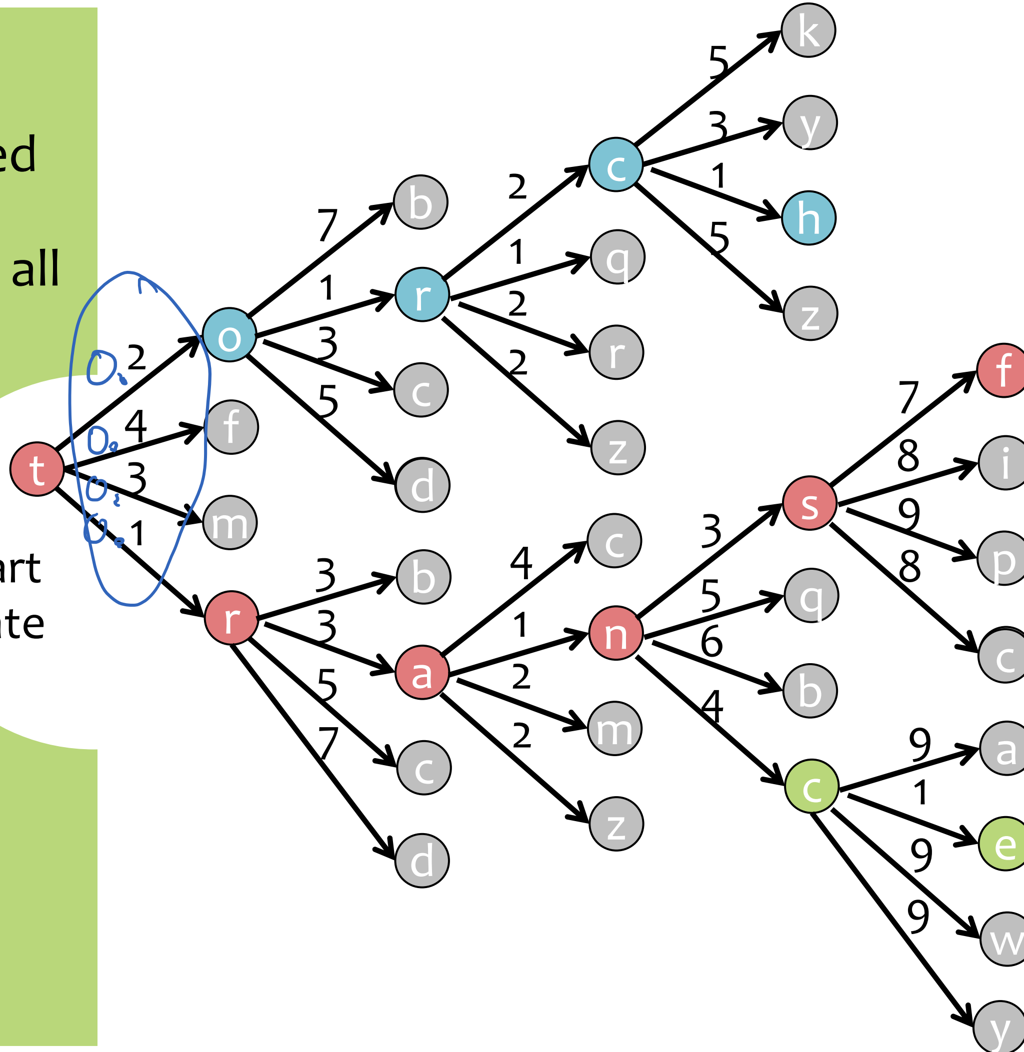
Greedy Search:

- 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
- Here we only show the high probability neighbors for space



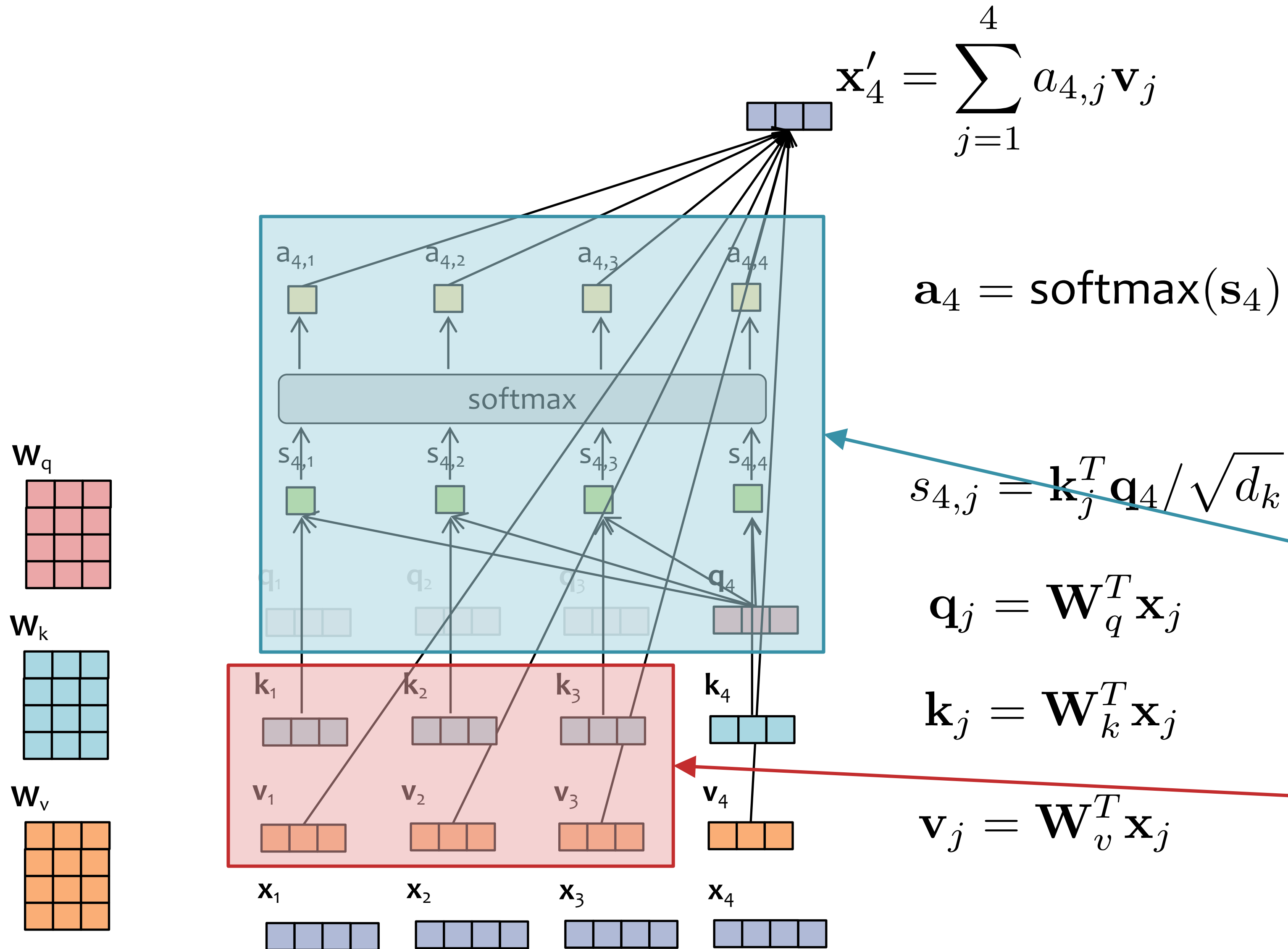
Goal:

- 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



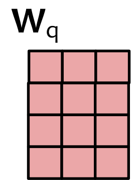
- At each timestep, we reuse all previous keys and values (i.e. we need to cache them)
- But we can get rid of the queries, similarity scores, and attention weights (i.e. we can let them fall out of the cache)

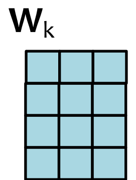
Discarded after this timestep

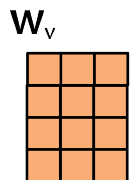
Computed for previous time-steps and reused for this timestep

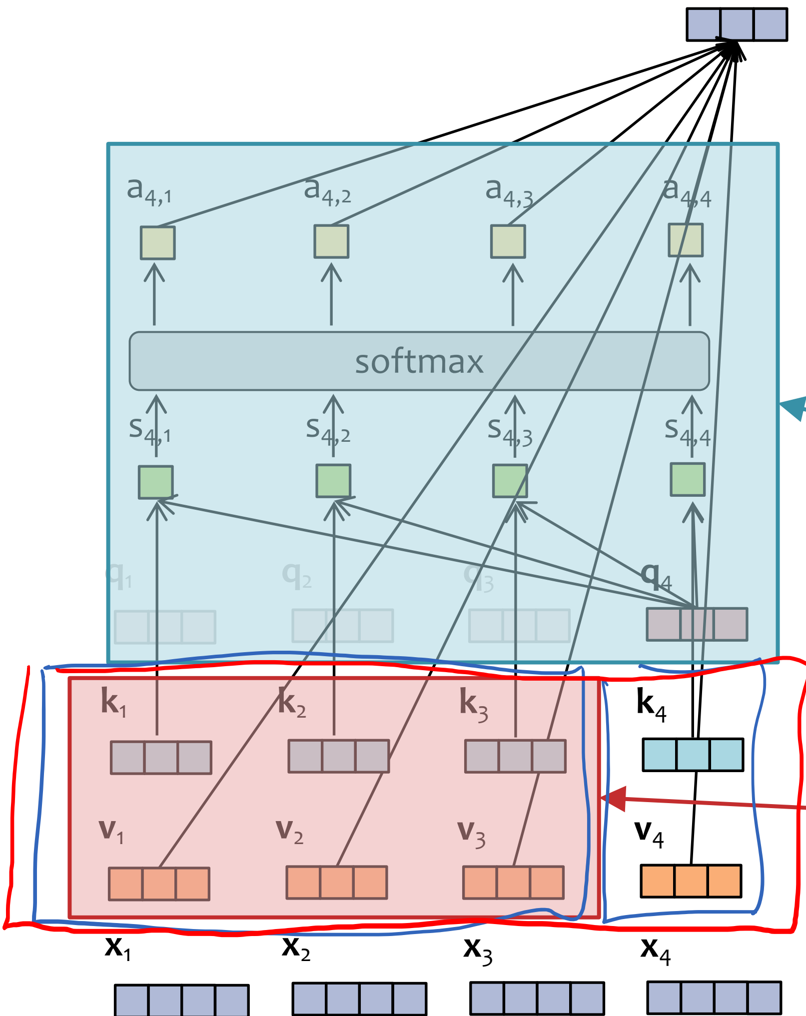
Key-Value Cache

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









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

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

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

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

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

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

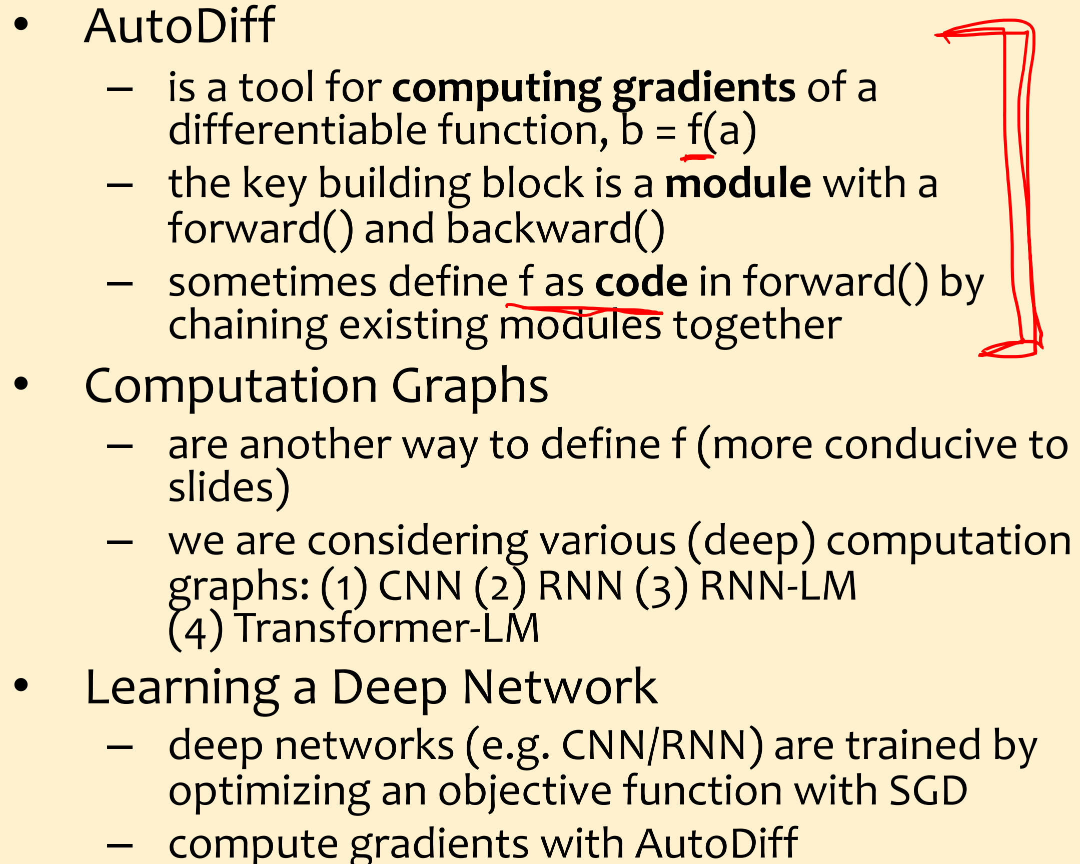
- At each timestep, we reuse all previous keys and values (i.e. we need to cache them)
- But we can get rid of the queries, similarity scores, and attention weights (i.e. we can let them fall out of the cache)

Discarded after this timestep

Computed for previous time-steps and reused for this timestep

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

Backpropagation

Recall...

Automatic Differentiation – Reverse Mode (aka. Backpropagation)

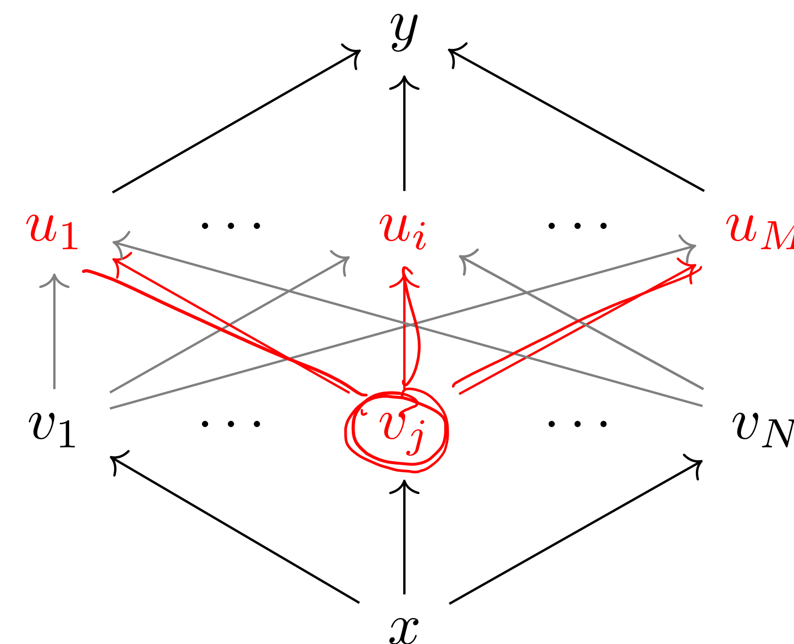
Forward Computation

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 $v_1, \ldots, v_N$
   a. Compute $u_i = g_i(v_1, \ldots, v_N)$
   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 $u_1, \ldots, u_M$ denote all the nodes with v_j as an input
   Assuming that $y = h(\mathbf{u}) = h(u_1, \ldots, u_M)$
   and $\mathbf{u} = g(\mathbf{v})$ or equivalently $u_i = g_i(v_1, \ldots, v_j, \ldots, v_N)$ 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

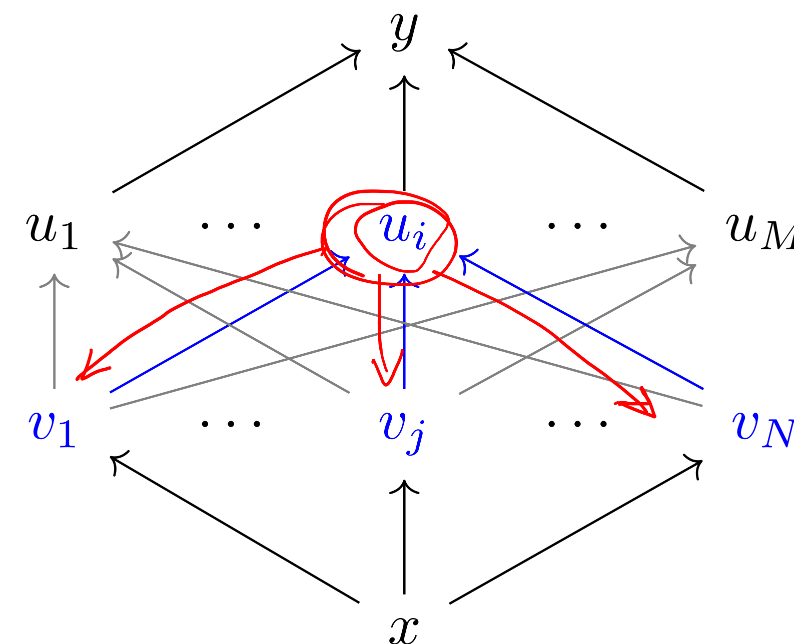
Automatic Differentiation – Reverse Mode (aka. Backpropagation)

<u>Forward Computation</u>

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 $v_1, \ldots, v_N$
   a. Compute $u_i = g_i(v_1, \ldots, v_N)$
   b. Store the result at the node

<u>Backward Computation (Version B)</u>

1. **Initialize** all partial derivatives dy/du_j to 0 and $dy/dy = 1$.
2. Visit each node in **reverse topological order**.
   For variable $u_i = g_i(v_1, \ldots, v_N)$
   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

Recall…

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)

A Recipe for

Recall...

Gradients

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!

1. Given training data: $\{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^N$

2. Choose each of these:

– Decision function

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

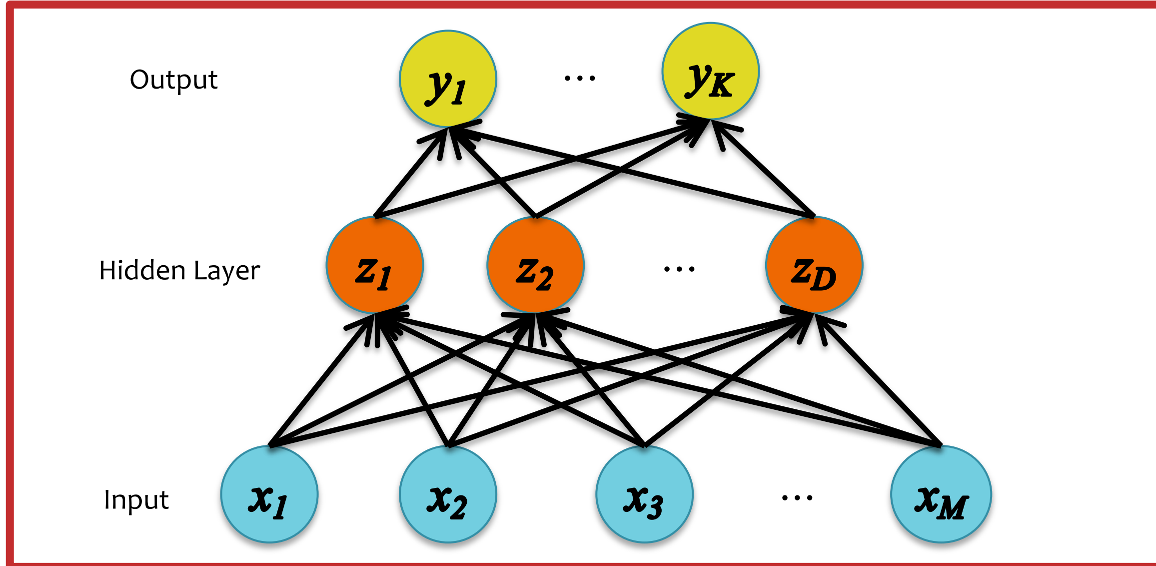
– Loss function

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

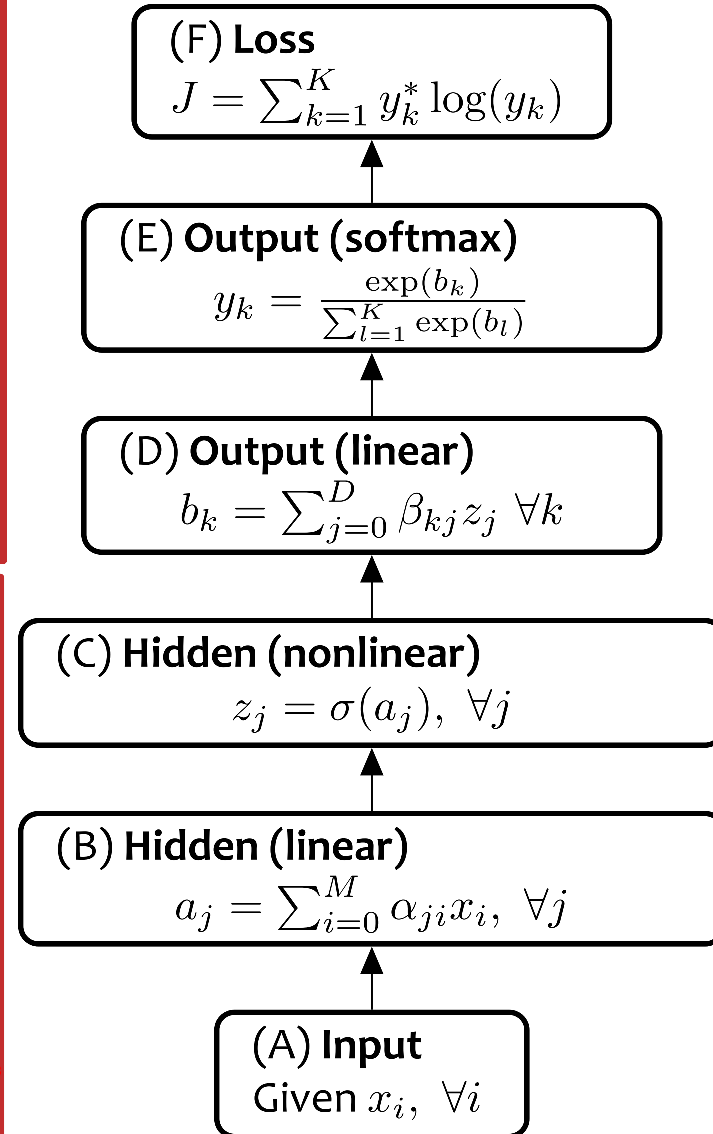
opposite the gradient)

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta_t \nabla \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \boldsymbol{y}_i)$$

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. $\mathbf{g}_{\mathbf{b}} = \mathbf{g}_{\hat{\mathbf{y}}}^T \left(\text{diag}(\hat{\mathbf{y}}) - \hat{\mathbf{y}}\hat{\mathbf{y}}^T\right)$
3. $\mathbf{b} = \boldsymbol{\beta}\mathbf{z}$	8. $\mathbf{g}_{\boldsymbol{\beta}} = \mathbf{g}_{\mathbf{b}}^T \mathbf{z}^T$
	$\mathbf{g}_{\mathbf{z}} = \boldsymbol{\beta}^T \mathbf{g}_{\mathbf{b}}^T$
2. $\mathbf{z} = \sigma(\mathbf{a})$	10. $\mathbf{g}_{\mathbf{a}} = \mathbf{g}_{\mathbf{z}} \odot \mathbf{z} \odot (1 - \mathbf{z})$
1. $\mathbf{a} = \boldsymbol{\alpha}\mathbf{x}$	11. $\mathbf{g}_{\boldsymbol{\alpha}} = \mathbf{g}_{\mathbf{a}} \mathbf{x}^T$



Backpropagation: Procedural Method

Algorithm 1 Forward Computation

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

Algorithm 2 Backpropagation

1: **procedure** NNBACKWARD(Training example $(\mathbf{x}, \mathbf{y})$, Params $\boldsymbol{\alpha}, \boldsymbol{\beta}$, 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: $\mathbf{g}_{\mathbf{b}} = \mathbf{g}_{\hat{\mathbf{y}}}^T \left(\text{diag}(\hat{\mathbf{y}}) - \hat{\mathbf{y}}\hat{\mathbf{y}}^T\right)$
5: $\mathbf{g}_{\boldsymbol{\beta}} = \mathbf{g}_{\mathbf{b}}^T \mathbf{z}^T$
6: $\mathbf{g}_{\mathbf{z}} = \boldsymbol{\beta}^T \mathbf{g}_{\mathbf{b}}^T$
7: $\mathbf{g}_{\mathbf{a}} = \mathbf{g}_{\mathbf{z}} \odot \mathbf{z} \odot (1 - \mathbf{z})$
8: $\mathbf{g}_{\boldsymbol{\alpha}} = \mathbf{g}_{\mathbf{a}}\mathbf{x}^T$
9: **return** parameter gradients $\mathbf{g}_{\boldsymbol{\alpha}}, \mathbf{g}_{\boldsymbol{\beta}}$

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)

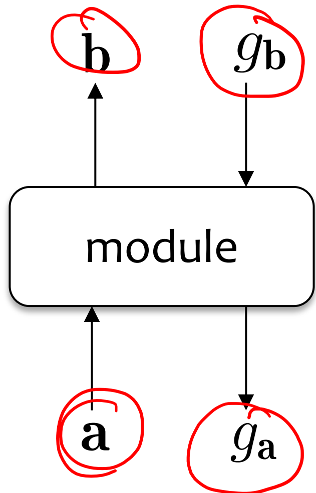
Module-based AutoDiff

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 $g_{\mathbf{a}} = \nabla_{\mathbf{a}} J = [\frac{\partial J}{\partial a_1}, \dots, \frac{\partial J}{\partial a_A}]$ given the gradient of output $g_{\mathbf{b}} = \nabla_{\mathbf{b}} J = [\frac{\partial J}{\partial b_1}, \dots, \frac{\partial J}{\partial b_B}]$, where J is the final real-valued output of the entire computation graph. This is done via the chain rule $\frac{\partial J}{\partial a_i} = \sum_{j=1}^{J} \frac{\partial J}{\partial b_j} \frac{d b_j}{d a_i}$ 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 σ is the sigmoid applied element-wise, and $\odot$ is element-wise multiplication s.t. $\mathbf{u} \odot \mathbf{v} = [u_1 v_1, \ldots, u_M v_M]$.

1: **procedure** SIGMOIDFORWARD($\mathbf{a}$)
2: $\quad \mathbf{b} = \sigma(\mathbf{a})$
3: $\quad$ **return** $\mathbf{b}$
4: **procedure** SIGMOIDBACKWARD($\mathbf{a}, \mathbf{b}, \mathbf{g_b}$)
5: $\quad \mathbf{g_a} = \mathbf{g_b} \odot \mathbf{b} \odot (1 - \mathbf{b})$
6: $\quad$ **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 $v_1, v_2, \ldots, v_D$ and whose non-diagonal entries are zero.

1: **procedure** SOFTMAXFORWARD($\mathbf{a}$)
2: $\quad \mathbf{b} = \text{softmax}(\mathbf{a})$
3: $\quad$ **return** $\mathbf{b}$
4: **procedure** SOFTMAXBACKWARD($\mathbf{a}, \mathbf{b}, \mathbf{g_b}$)
5: $\quad \mathbf{g_a} = \mathbf{g_b}^T \left(\text{diag}(\mathbf{b}) - \mathbf{b}\mathbf{b}^T\right)$
6: $\quad$ **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($\mathbf{a}, \boldsymbol{\omega}$)
2: $\quad \mathbf{b} = \boldsymbol{\omega}\mathbf{a}$
3: $\quad$ **return** $\mathbf{b}$
4: **procedure** LINEARBACKWARD($\mathbf{a}, \omega, \mathbf{b}, \mathbf{g_b}$)
5: $\quad \mathbf{g}_{\boldsymbol{\omega}} = \mathbf{g_b}\mathbf{a}^T$
6: $\quad \mathbf{g_a} = \boldsymbol{\omega}^T \mathbf{g_b}$
7: $\quad$ **return** $\mathbf{g}_{\boldsymbol{\omega}}, \mathbf{g_a}$

Cross-Entropy Module The cross-entropy layer has two inputs: a gold one-hot vector $\mathbf{a}$ and a predicted probability distribution $\hat{\mathbf{a}}$. It's 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($\mathbf{a}, \hat{\mathbf{a}}$)
2: $\quad b = -\mathbf{a}^T \log \hat{\mathbf{a}}$
3: $\quad$ **return** $\mathbf{b}$
4: **procedure** CROSSENTROPYBACKWARD($\mathbf{a}, \hat{\mathbf{a}}, b, g_b$)
5: $\quad \mathbf{g}_{\hat{\mathbf{a}}} = -g_b(\mathbf{a} \div \hat{\mathbf{a}})$
6: $\quad$ **return** $\mathbf{g_a}$

Module-based AutoDiff

Algorithm 1 Forward Computation

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

Algorithm 2 Backpropagation

1: **procedure** NNBACKWARD(Training example $(\mathbf{x}, \mathbf{y})$, Parameters $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, 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: $g_J = \frac{dJ}{dJ} = 1$ ▷ Base case
4: $\mathbf{g}_{\hat{\mathbf{y}}} = \text{CROSSENTROPYBACKWARD}(\mathbf{y}, \hat{\mathbf{y}}, J, g_J)$
5: $\mathbf{g}_{\mathbf{b}} = \text{SOFTMAXBACKWARD}(\mathbf{b}, \hat{\mathbf{y}}, \mathbf{g}_{\hat{\mathbf{y}}})$
6: $\mathbf{g}_{\boldsymbol{\beta}}, \mathbf{g}_{\mathbf{z}} = \text{LINEARBACKWARD}(\mathbf{z}, \mathbf{b}, \mathbf{g}_{\mathbf{b}})$
7: $\mathbf{g}_{\mathbf{a}} = \text{SIGMOIDBACKWARD}(\mathbf{a}, \mathbf{z}, \mathbf{g}_{\mathbf{z}})$
8: $\mathbf{g}_{\boldsymbol{\alpha}}, \mathbf{g}_{\mathbf{x}} = \text{LINEARBACKWARD}(\mathbf{x}, \mathbf{a}, \mathbf{g}_{\mathbf{a}})$ ▷ We discard $\mathbf{g}_{\mathbf{x}}$
9: **return** parameter gradients $\mathbf{g}_{\boldsymbol{\alpha}}, \mathbf{g}_{\boldsymbol{\beta}}$

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(·), just follow the computation graph in **reverse topological order**

```
1  class Sigmoid(Module)
2      method forward(a)
3          b = σ(a)
4          return b
5      method backward(a, b, g_b)
6          g_a = g_b ⊙ b ⊙ (1 − b)
7          return g_a
```

```
1  class Linear(Module)
2      method forward(a, ω)
3          b = ωa
4          return b
5      method backward(a, ω, b, g_b)
6          g_ω = g_b a^T
7          g_a = ω^T g_b
8          return g_ω, g_a
```

```
1  class Softmax(Module)
2      method forward(a)
3          b = softmax(a)
4          return b
5      method backward(a, b, g_b)
6          g_a = g_b^T (diag(b) − bb^T)
7          return g_a
```

```
1  class CrossEntropy(Module)
2      method forward(a, â)
3          b = −a^T log â
4          return b
5      method backward(a, â, b, g_b)
6          g_â = −g_b(a ÷ â)
7          return 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 x, Tensor y, Tensor α, Tensor β)
11         a = lin1_layer.apply_fwd(x, α)
12         z = sig_layer.apply_fwd(a)
13         b = lin2_layer.apply_fwd(z, β)
14         ŷ = soft_layer.apply_fwd(b)
15         J = ce_layer.apply_fwd(y, ŷ)
16         return J.out_tensor
17
18     method backward(Tensor x, Tensor y, Tensor α, Tensor β)
19         tape_bwd()
20         return lin1_layer.in_gradients[1], lin2_layer.in_gradients[1]
```

Module-based AutoDiff (OOP Version)

class Tensor

out_tensor = self

```
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, α)
12         z =sig_layer.apply_fwd(a)
13         b =lin2_layer.apply_fwd(z, β)
14         ŷ =soft_layer.apply_fwd(b)
15         J =ce_layer.apply_fwd(y, ŷ)
16         return J.out_tensor
17
18     method backward(Tensor x, Tensor y, Tenso
19         tape_bwd()
20         return lin1_layer.in_gradients[1], lin2_la
```

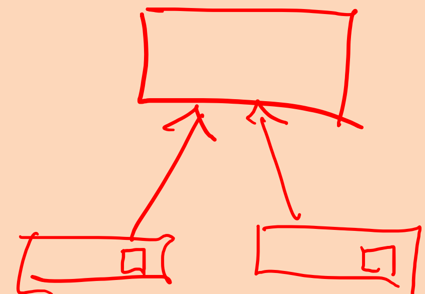
```
1  global tape = stack()
2
3  class Module:
4
5      method init()
6          out_tensor = null
7          out_gradient = 1
8
9      method apply_fwd(List in_modules)
10         in_tensors = [x.out_tensor for x in in_modules]
11         out_tensor = forward(in_tensors)
12         tape.push(self)
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()
```

= [lin1_layer]

= [lin1_layer, sig_layer]

= [lin1_layer, ..., ce_layer]

Tensor(1s)



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, α)
12         z =sig_layer.apply_fwd(a)
13         b =lin2_layer.apply_fwd(z, β)
14         ŷ =soft_layer.apply_fwd(b)
15         J =ce_layer.apply_fwd(y, ŷ)
16         return J.out_tensor
17
18     method backward(Tensor x, Tensor y, Tenso
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     model = NeuralNetwork()
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()
```

Example adapted from https://pytorch.org/tutorials/beginner/basics/quickstart_tutorial.html

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 α, Tensor β)
11          a = lin1_layer.apply_fwd(x, α)
12          z = sig_layer.apply_fwd(a)
13          b = lin1_layer.apply_fwd(z, β)
14          ŷ = soft_layer.apply_fwd(b)
15          J = ce_layer.apply_fwd(y, ŷ)
16          return J.out_tensor
```

```
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