10-423/623: Generative AI Lecture 3 – Learning LLMs and Decoding

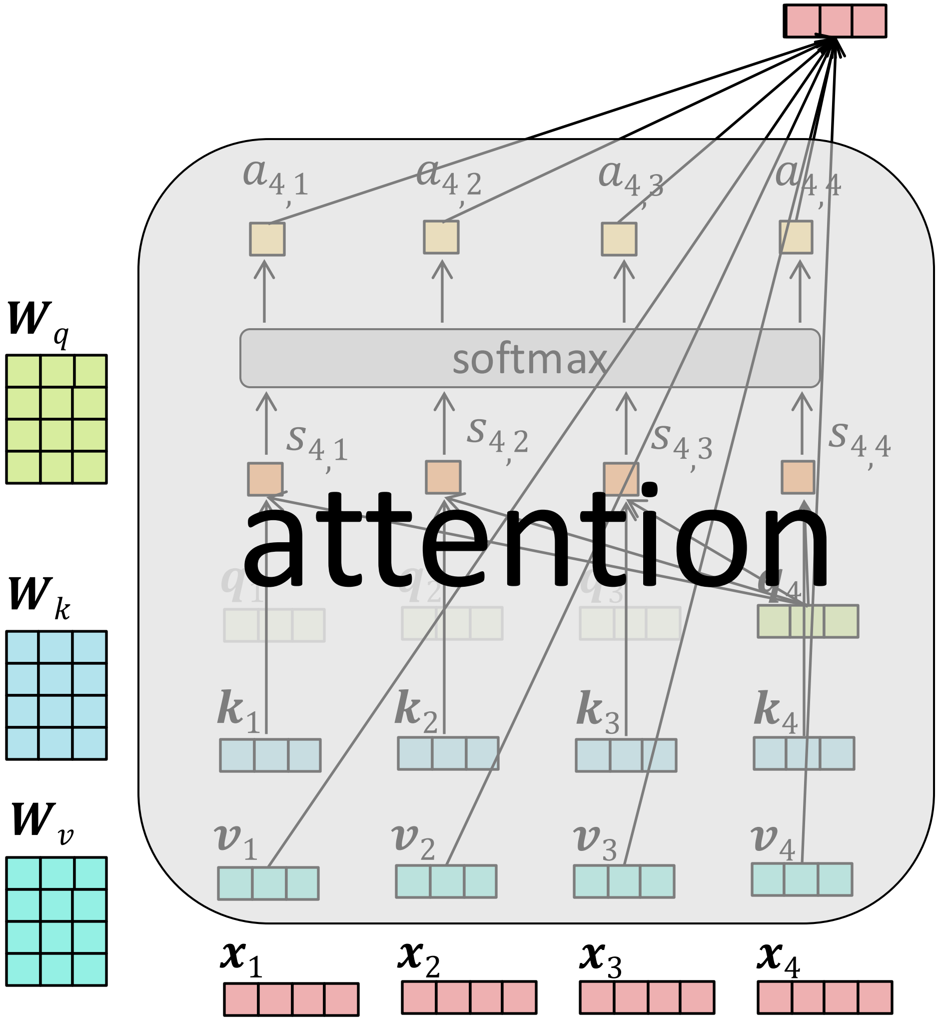
Henry Chai & Matt Gormley

9/4/24

Front Matter

- Announcements:
  - HW0 released 8/28, due 9/9 (next Monday) at 11:59 PM
    - Two components: written and programming
      - Separate assignments on Gradescope
    - Unique policy specific to HW0: **we will grant (almost) any extension request**
  - Quiz 1 in-class on 9/11 (next Wednesday)
  - Instructor OH start this week; see the OH calendar for more details

Recall: Scaled Dot-Product Attention



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

$$a_{4,j} = \mathrm{softmax}(s_{4,j})$$ Attention weights

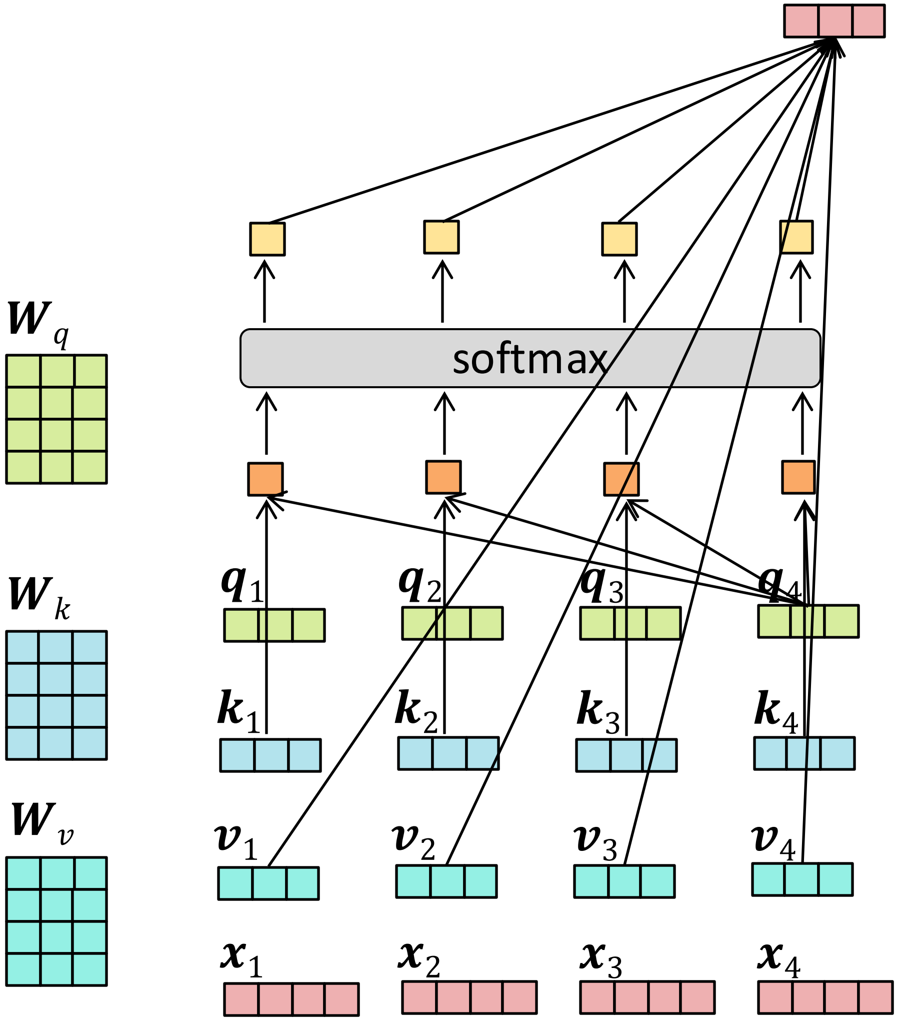
$$s_{4,j} = \frac{\boldsymbol{k}_j^T \boldsymbol{q}_4}{\sqrt{d_k}}$$ Scores

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

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

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

Scaled Dot-Product Attention: Matrix Form

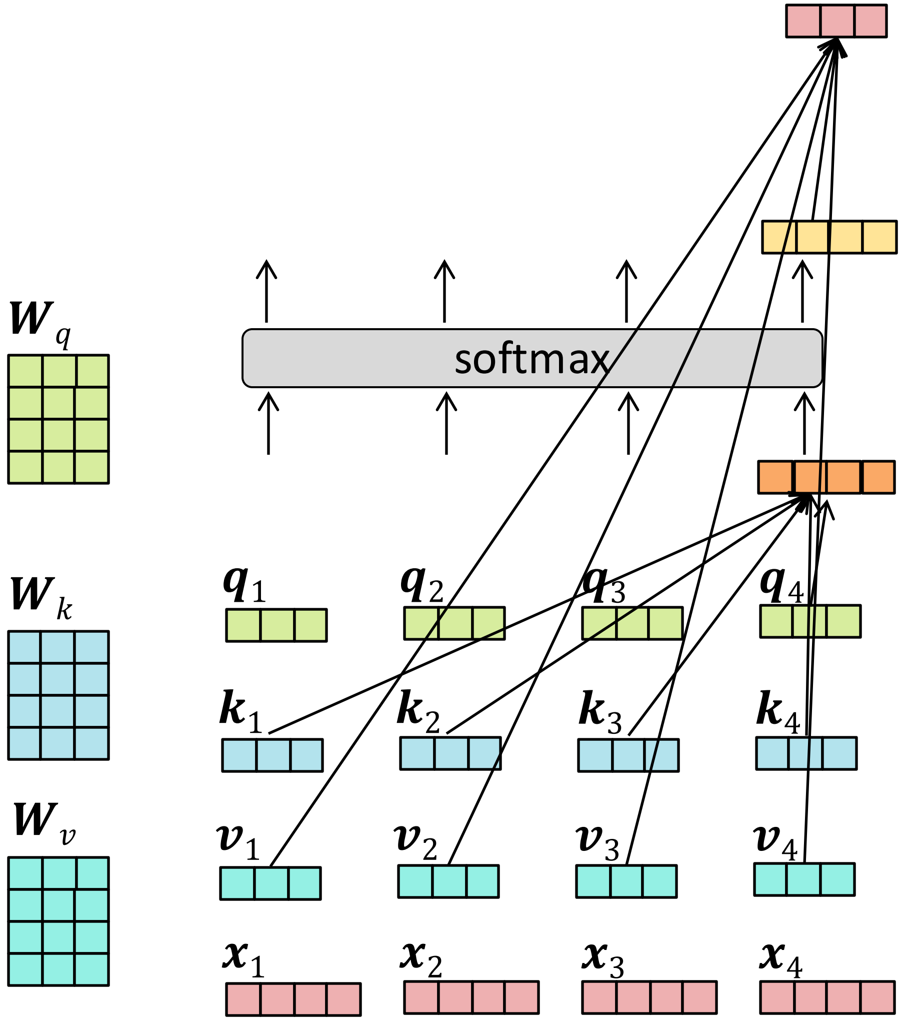


$$[\boldsymbol{q}_1, \cdots, \boldsymbol{q}_N] = \boldsymbol{W}_q^T[\boldsymbol{x}_1, \cdots, \boldsymbol{x}_N]$$

$$[\boldsymbol{k}_1, \cdots, \boldsymbol{k}_N] = \boldsymbol{W}_k^T[\boldsymbol{x}_1, \cdots, \boldsymbol{x}_N]$$

$$[\boldsymbol{v}_1, \cdots, \boldsymbol{v}_N] = \boldsymbol{W}_v^T[\boldsymbol{x}_1, \cdots, \boldsymbol{x}_N]$$

Scaled Dot-Product Attention: Matrix Form

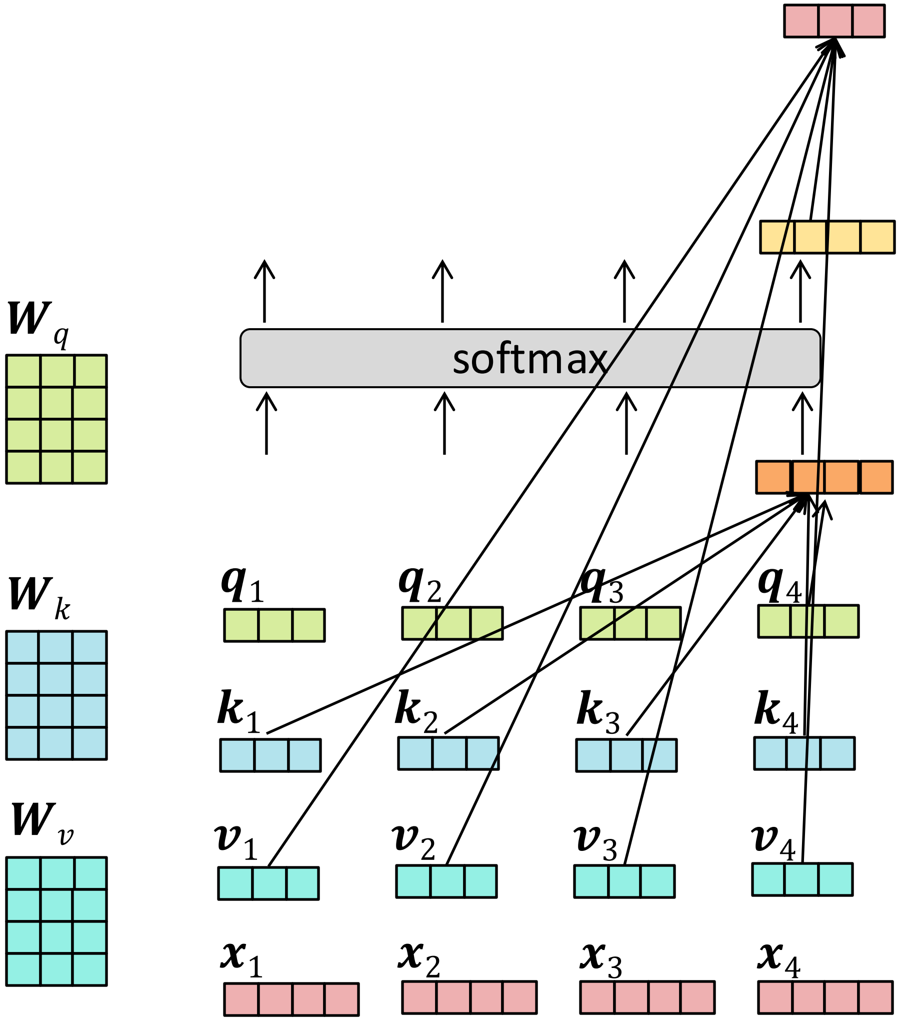


$$Q = [\boldsymbol{q}_1, \cdots, \boldsymbol{q}_N]^T = [\boldsymbol{x}_1, \cdots, \boldsymbol{x}_N]^T \boldsymbol{W}_q$$

$$K = [\boldsymbol{k}_1, \cdots, \boldsymbol{k}_N]^T = [\boldsymbol{x}_1, \cdots, \boldsymbol{x}_N]^T \boldsymbol{W}_k$$

$$V = [\boldsymbol{v}_1, \cdots, \boldsymbol{v}_N]^T = [\boldsymbol{x}_1, \cdots, \boldsymbol{x}_N]^T \boldsymbol{W}_v$$

Scaled Dot-Product Attention: Matrix Form

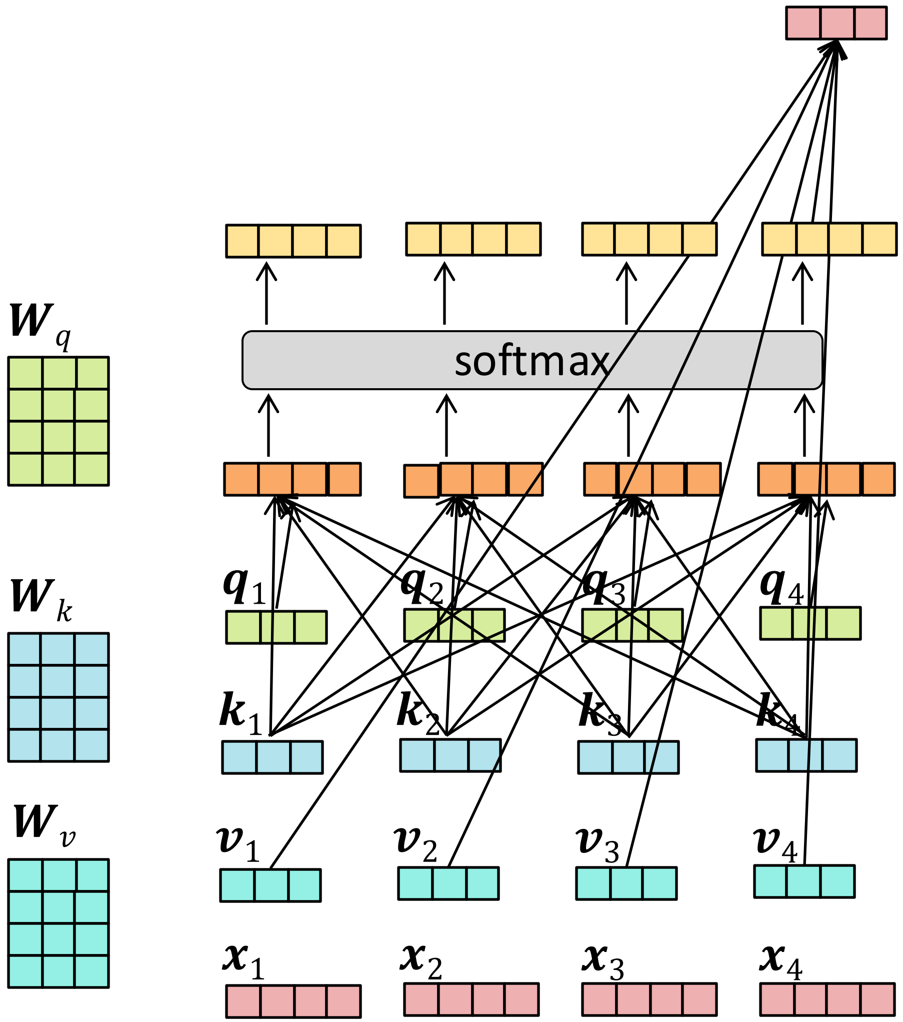


$Q = XW_q$

$K = XW_k$

$V = XW_v$

Scaled Dot-Product Attention: Matrix Form



$$A = \mathrm{softmax}(S)$$

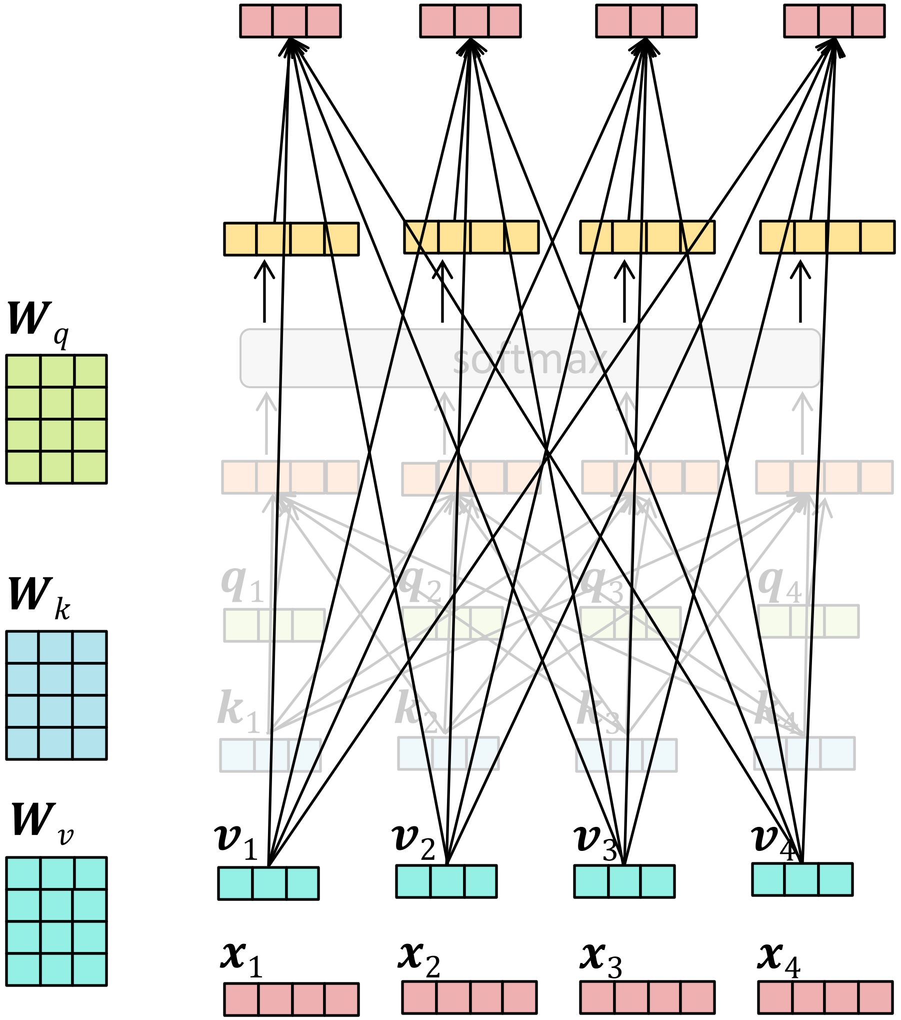
$$S = \frac{QK^T}{\sqrt{d_k}}$$

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

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

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

Scaled Dot-Product Attention: Matrix Form



$$X' = AV = \boxed{\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V}$$

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

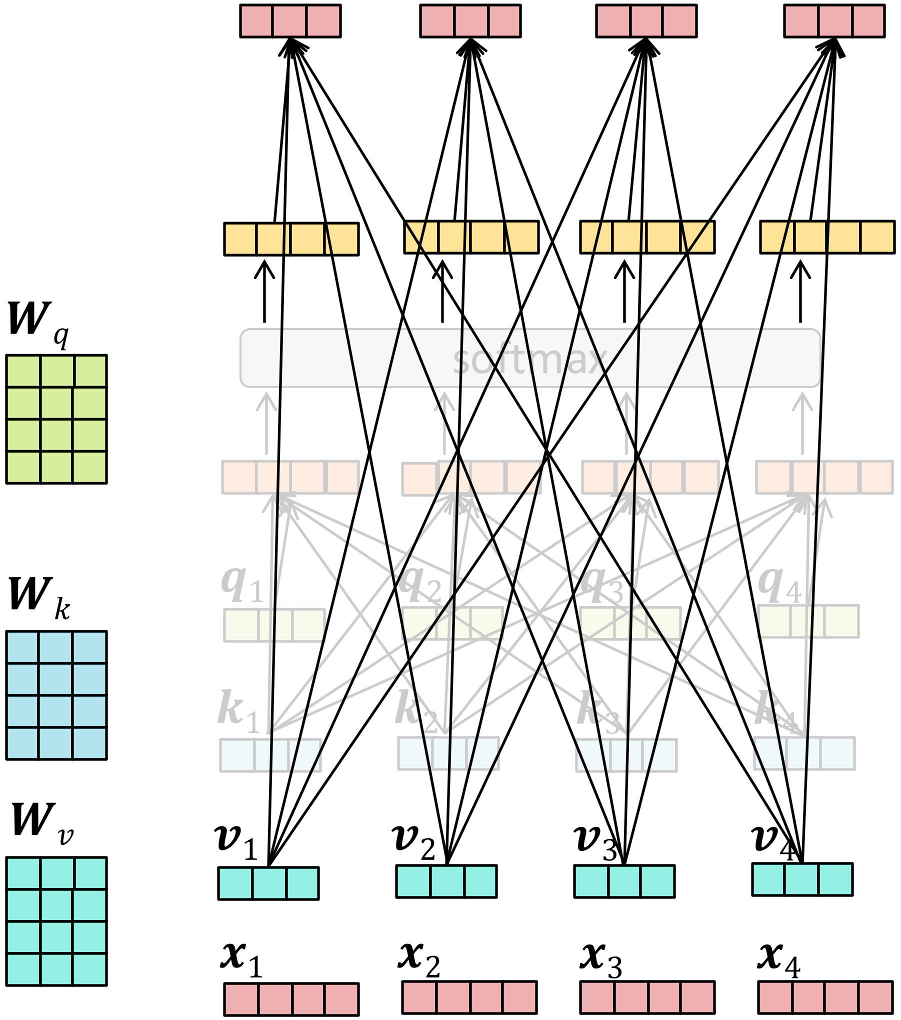
$$S = \frac{QK^T}{\sqrt{d_k}}$$

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

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

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

Which dimension is the softmax applied over: row-wise or column-wise?



$$X' = AV = \boxed{\mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V}$$

$$A = \mathrm{softmax}(S)$$

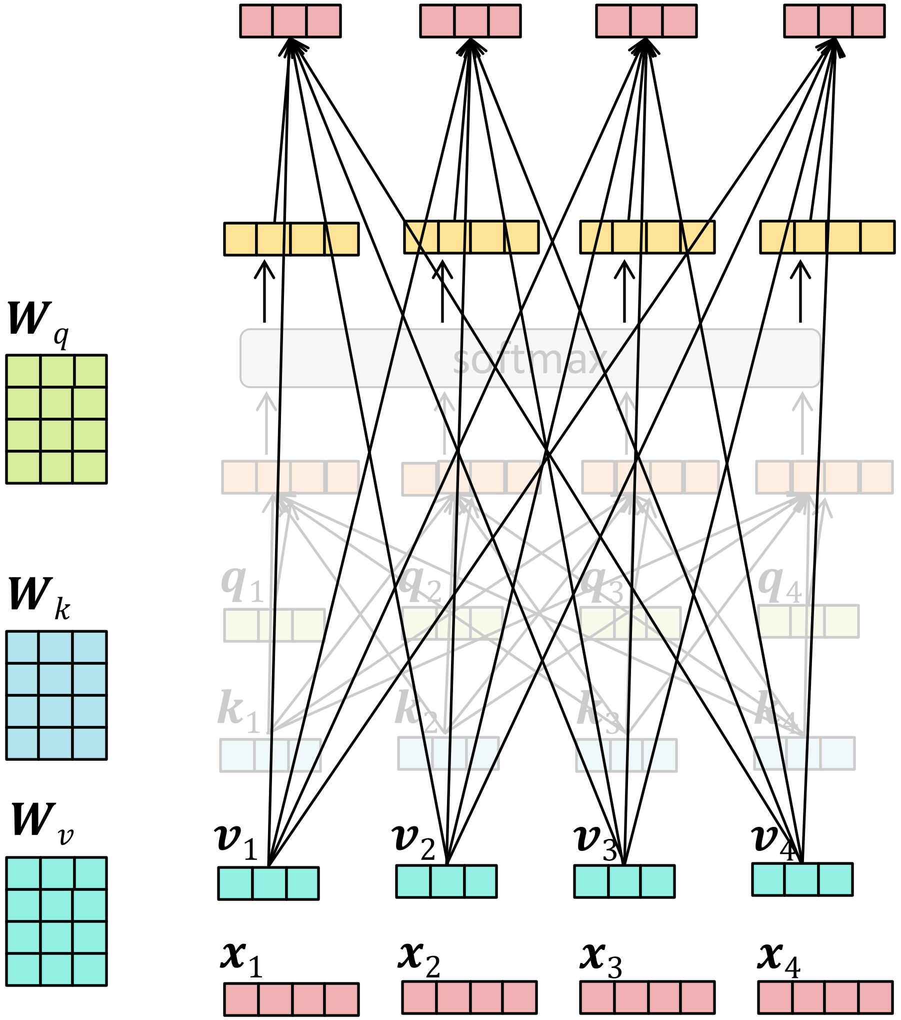
$$S = \frac{QK^T}{\sqrt{d_k}}$$

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

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

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

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



$$X' = AV = \boxed{\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V}$$

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

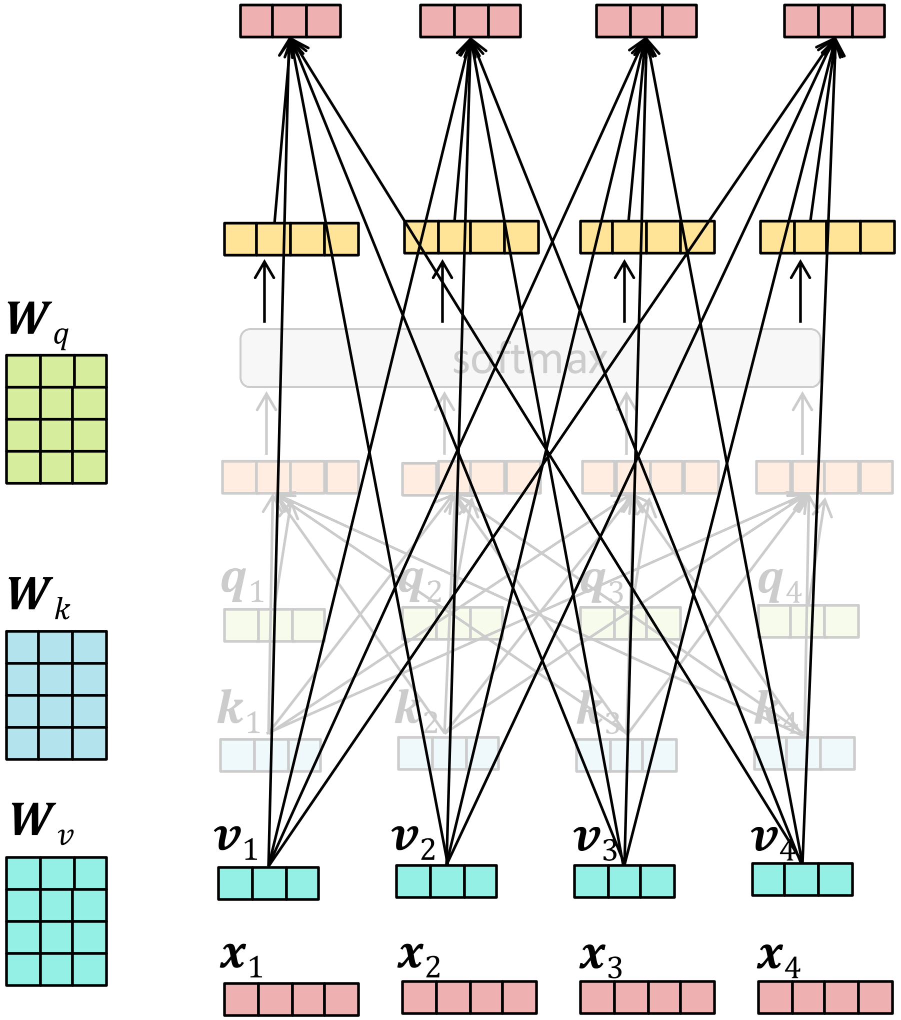
$$S = \frac{QK^T}{\sqrt{d_k}}$$

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

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

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

Causal Attention



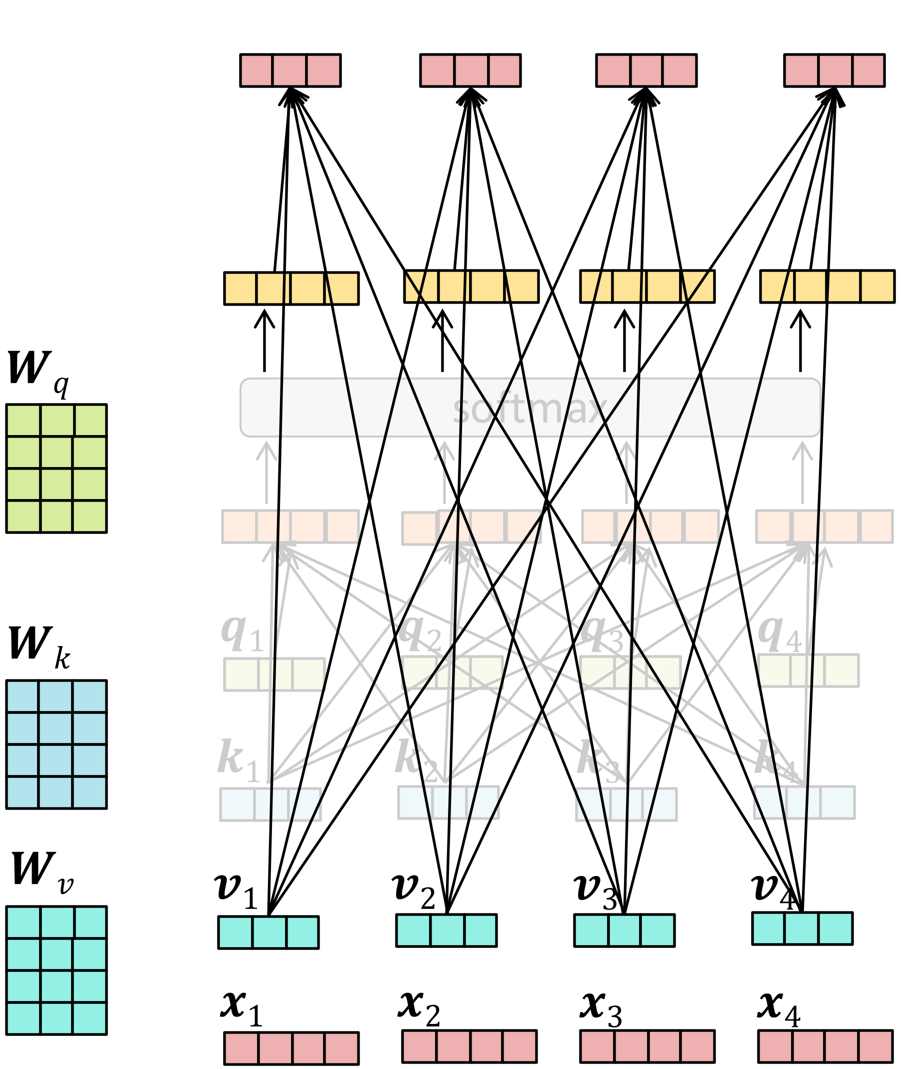
$$X' = AV = \boxed{\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V}$$

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

- 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!

Masking

Idea: we can effectively delete or "mask" some of these arrows by selectively setting attention weights to 0

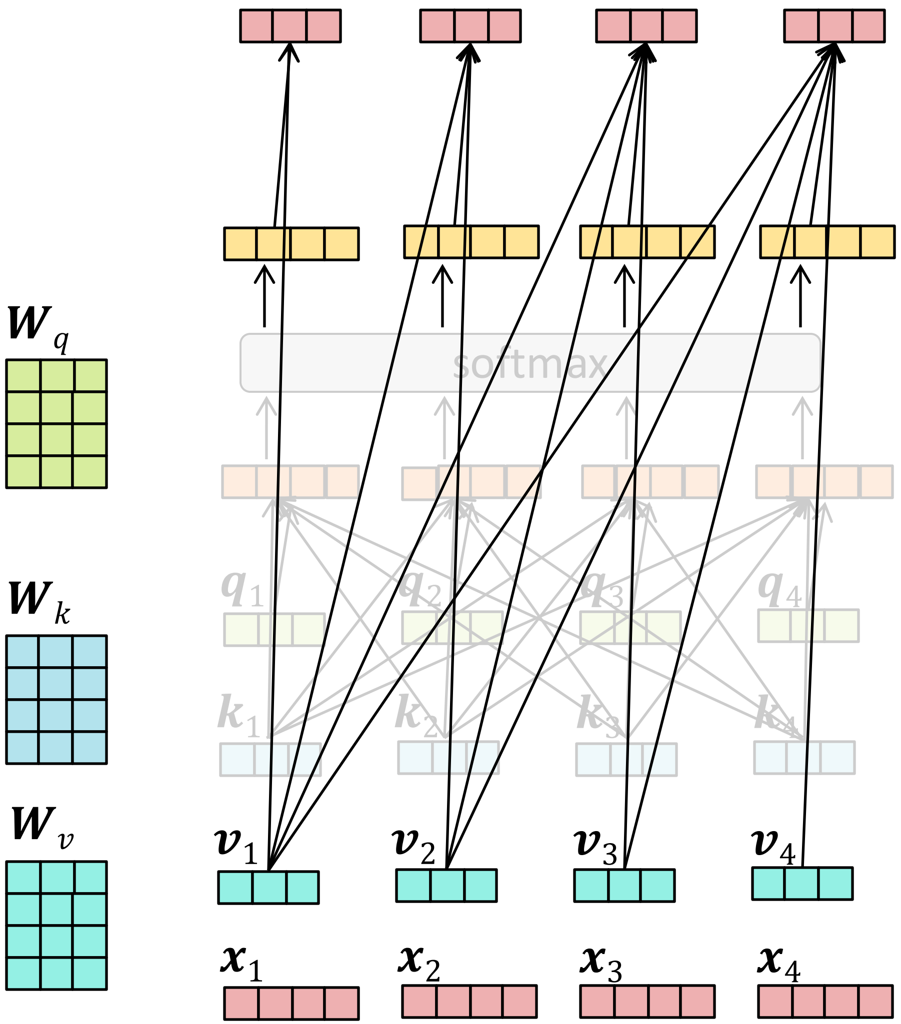


$$X' = AV = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

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

Masking

Idea: we can effectively delete or "mask" some of these arrows by selectively setting attention weights to 0



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

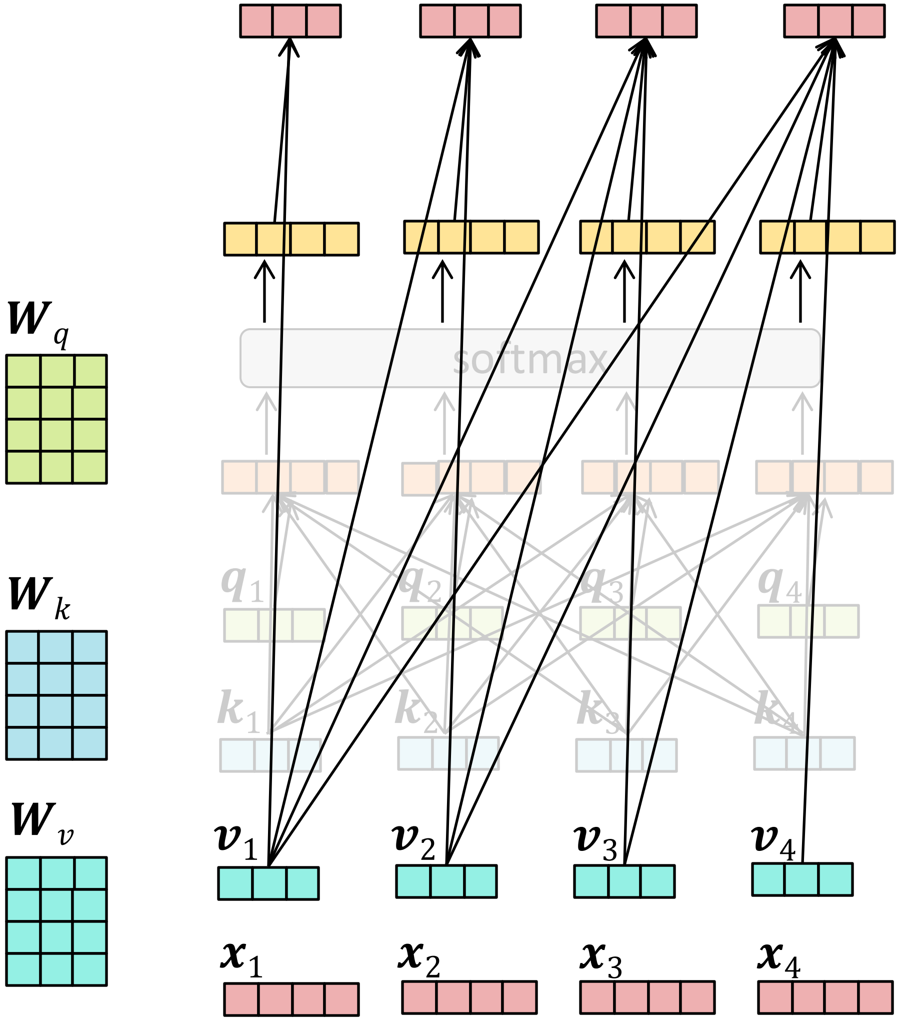
$$A = \text{softmax}(S)$$

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

$$\frac{\exp(-\infty)}{\sum_j \exp s_j} = \frac{0}{\sum_j \exp s_j}$$

Idea: we can effectively delete or "mask" some of these arrows by selectively setting attention weights to 0

Which of the mask matrices corresponds to this set of arrows?



$$X' = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V$$

$$A_{mask} = \mathrm{softmax}(S + M)$$

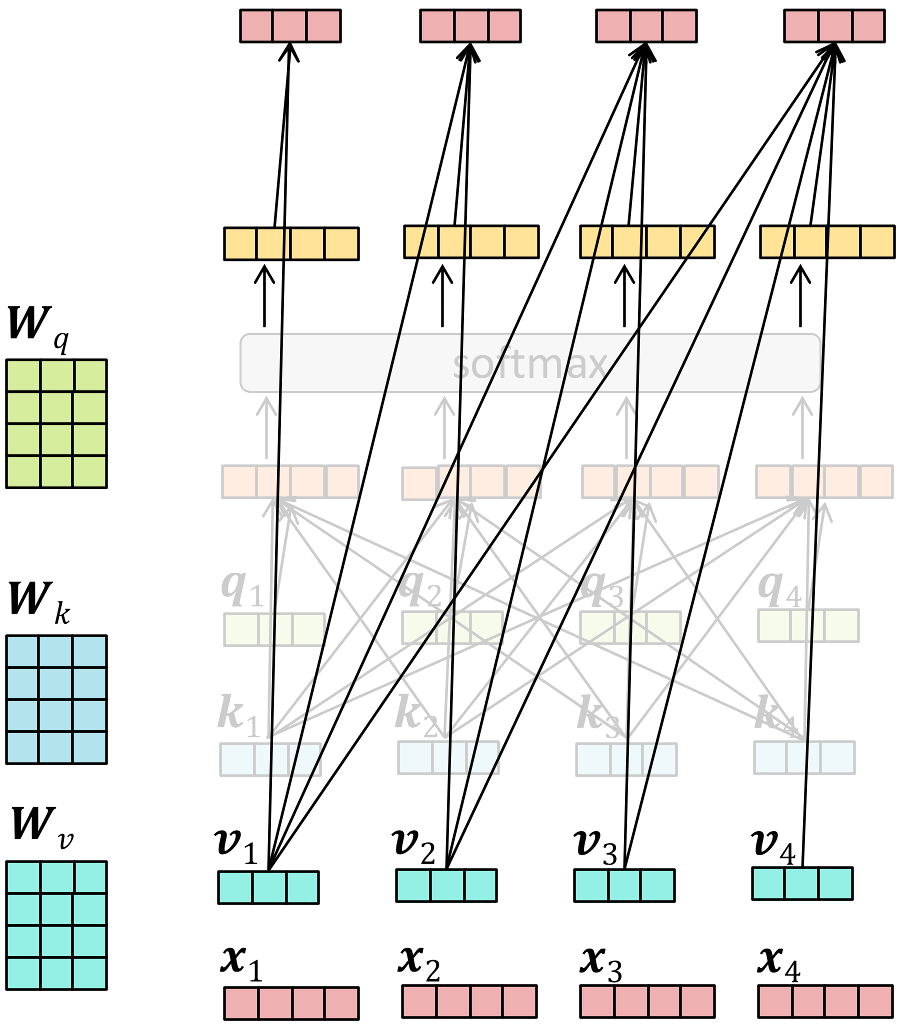
A. $M = \begin{bmatrix} 0 & 0 & 0 & 0 \\ -\infty & 0 & 0 & 0 \\ -\infty & -\infty & 0 & 0 \\ -\infty & -\infty & -\infty & 0 \end{bmatrix}$

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

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

Masked Scaled Dot-Product Attention: Matrix Form

Idea: we can effectively delete or "mask" some of these arrows by selectively setting attention weights to 0



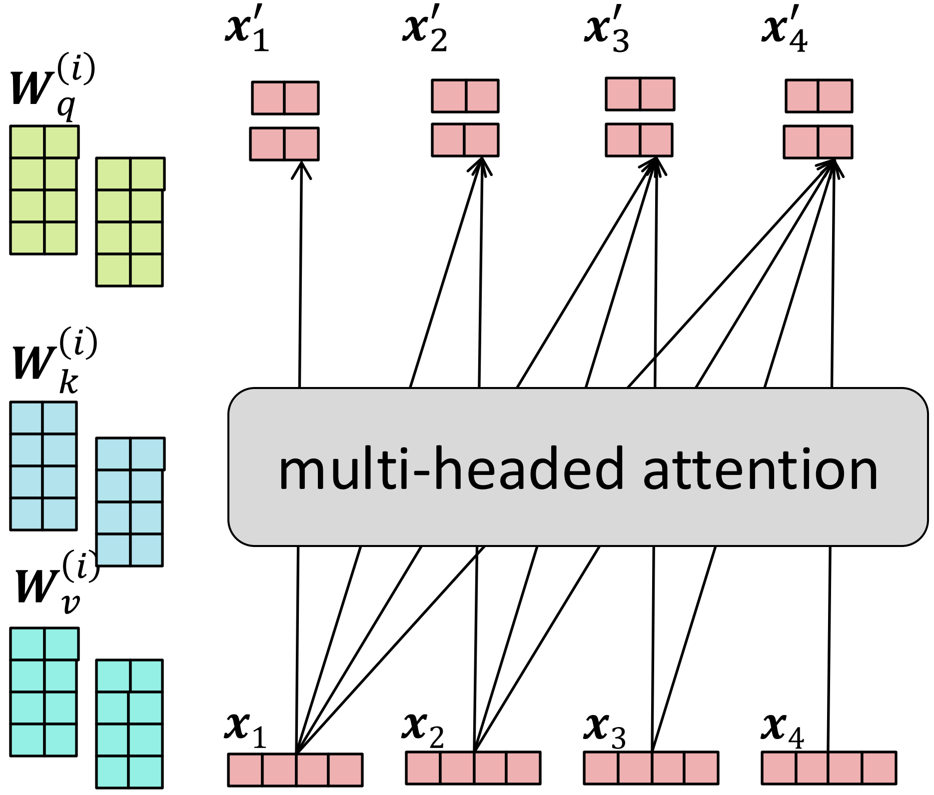
$$X' = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V$$

$$A_{mask} = \text{softmax}(S + M)$$

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

Masked Multi-headed Attention: Matrix Form

$$X' = \underset{i}{\mathrm{concat}}\left\{\mathrm{softmax}\left(\frac{Q^{(i)}K^{(i)^T}}{\sqrt{d_k}} + M\right)V^{(i)}\right\} \text{ where } \begin{aligned} Q^{(i)} &= X\boldsymbol{W}_q^{(i)} \\ K^{(i)} &= X\boldsymbol{W}_k^{(i)} \\ V^{(i)} &= X\boldsymbol{W}_b^{(i)} \end{aligned}$$



Summary thus Far

1. Language Modeling
   - Key idea: condition on previous words to **sample the next word**
   - To define the **probability** of the next word, we can…
     - use conditional independence assumption (n-grams)
     - throw a neural network at it (RNN-LM or Transformer-LM)

2. (Module-based) AutoDiff
   - A tool for **computing gradients** of a differentiable function,
     $$b = f(a)$$
   - Key building block: **modules** with `forward()` and `backward()`
     - Can define f as **code** in `forward()` by chaining existing modules together
     - Can define f as a **computation graph**

Summary thus Far

1. Language Modeling
   - Key idea: condition on previous words to **sample the next word**
   - To define the **probability** of the next word, we can…
     - use conditional independence assumption (n-grams)
     - throw a neural network at it (RNN-LM or Transformer-LM)

…to learn one of these?

2. (Module-based) AutoDiff
   - A tool for **computing gradients** of a differentiable function,
     $$b = f(a)$$
   - Key building block: modules with `forward()` and `backward()`
     - Can define f as **code** in `forward()` by chaining existing modules together
     - Can define f as a **computation graph**

How can we use this stuff…

Stochastic Gradient Descent

- Input: training dataset $\mathcal{D} = \{(\boldsymbol{x}^{(n)}, y^{(n)})\}_{n=1}^{N}$, step size γ

1. Randomly initialize the parameters of your neural LM $\boldsymbol{\theta}^{(0)}$ and set $t = 0$
2. While TERMINATION CRITERION is not satisfied
   a. Randomly sample a data point from $\mathcal{D}$, $(\boldsymbol{x}^{(i)}, y^{(i)})$
   b. Compute the gradient of the loss w.r.t. the sample using (module-based) AutoDiff: $\nabla J^{(i)}(\boldsymbol{\theta}^{(t)})$
   c. Update $\boldsymbol{\theta}$: $\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \gamma \nabla J^{(i)}(\boldsymbol{\theta}^{(t)})$
   d. Increment t: $t \leftarrow t + 1$

- Output: $\boldsymbol{\theta}^{(t)}$

Mini-batch Stochastic Gradient Descent

- Input: training dataset $\mathcal{D} = \left\{\left(\boldsymbol{x}^{(n)}, y^{(n)}\right)\right\}_{n=1}^{N}$, step size γ, and batch size B

1. Randomly initialize the parameters of your neural LM $\boldsymbol{\theta}^{(0)}$ and set $t = 0$
2. While TERMINATION CRITERION is not satisfied
   a. Randomly sample B data points from $\mathcal{D}$, $\left\{\left(\boldsymbol{x}^{(b)}, y^{(b)}\right)\right\}_{b=1}^{B}$
   b. Compute the gradient of the loss w.r.t. the sampled *batch* using (module-based) AutoDiff: $\nabla J^{(B)}\left(\boldsymbol{\theta}^{(t)}\right)$
   c. Update $\boldsymbol{\theta}$: $\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \gamma \nabla J^{(B)}\left(\boldsymbol{\theta}^{(t)}\right)$
   d. Increment t: $t \leftarrow t + 1$

- Output: $\boldsymbol{\theta}^{(t)}$

Recall: n-gram Language Model Training

- How do we train an n-gram language model?
- Using training data! Simply count frequency of next words, which **maximizes the likelihood** of the data under the various categorial distributions in the model

Narwhals are big aquatic mammals that...
Who knows what **narwhals are** hiding?
Watch out, the **narwhals are** coming!
These **narwhals are** friendly!
Narwhals are a surprisingly la
The **narwhals are** a punk rock
Narwhals are big fans of mac
Narwhals are generated by A

x_t	$p(x_t \| \text{narwhals, are})$
big	2/8
hiding	1/8
coming	1/8
friendly	1/8
a	2/8
generated	1/8

We can use the same principle of MLE to optimize the parameters of our Neural LMs!

- How do we train an n-gram language model?
- Using training data! Simply count frequency of next words, which **maximizes the likelihood** of the data under the various categorial distributions in the model

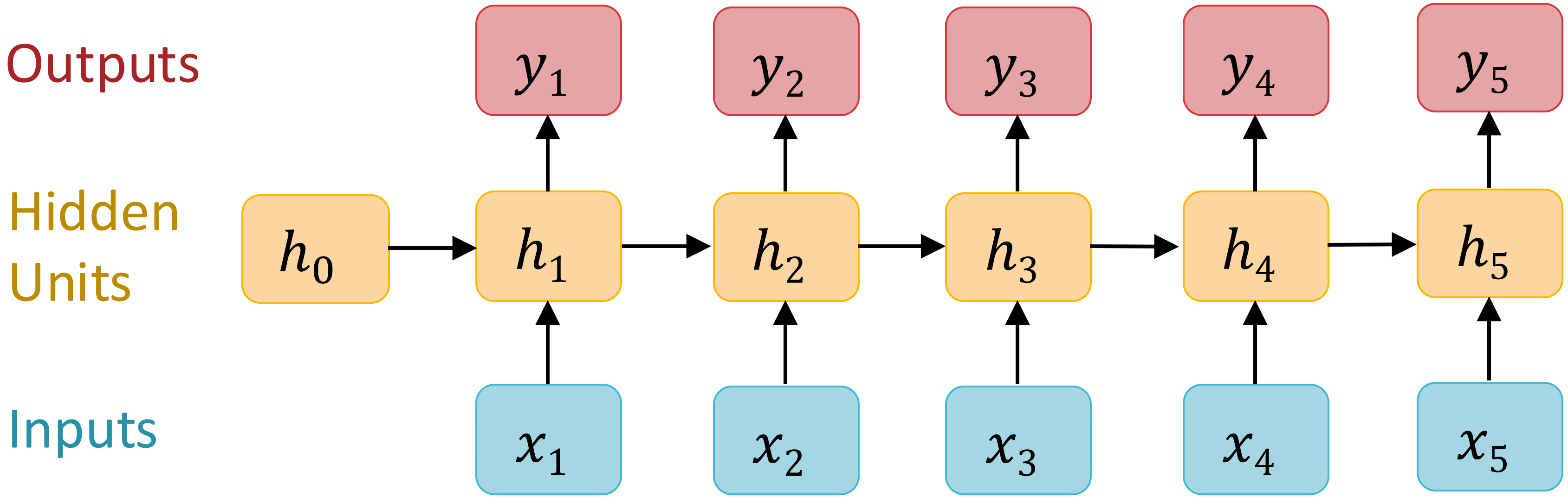
Narwhals are big aquatic mammals that...
Who knows what **narwhals are** hiding?
Watch out, the **narwhals are** coming!
These **narwhals are** friendly!
Narwhals are a surprisingly la
The **narwhals are** a punk rock
Narwhals are big fans of mac
Narwhals are generated by A

x_t	$p(x_t \mid \text{narwhals, are})$
big	2/8
hiding	1/8
coming	1/8
friendly	1/8
a	2/8
generated	1/8

Recurrent Neural Networks

$$h_t = \phi(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

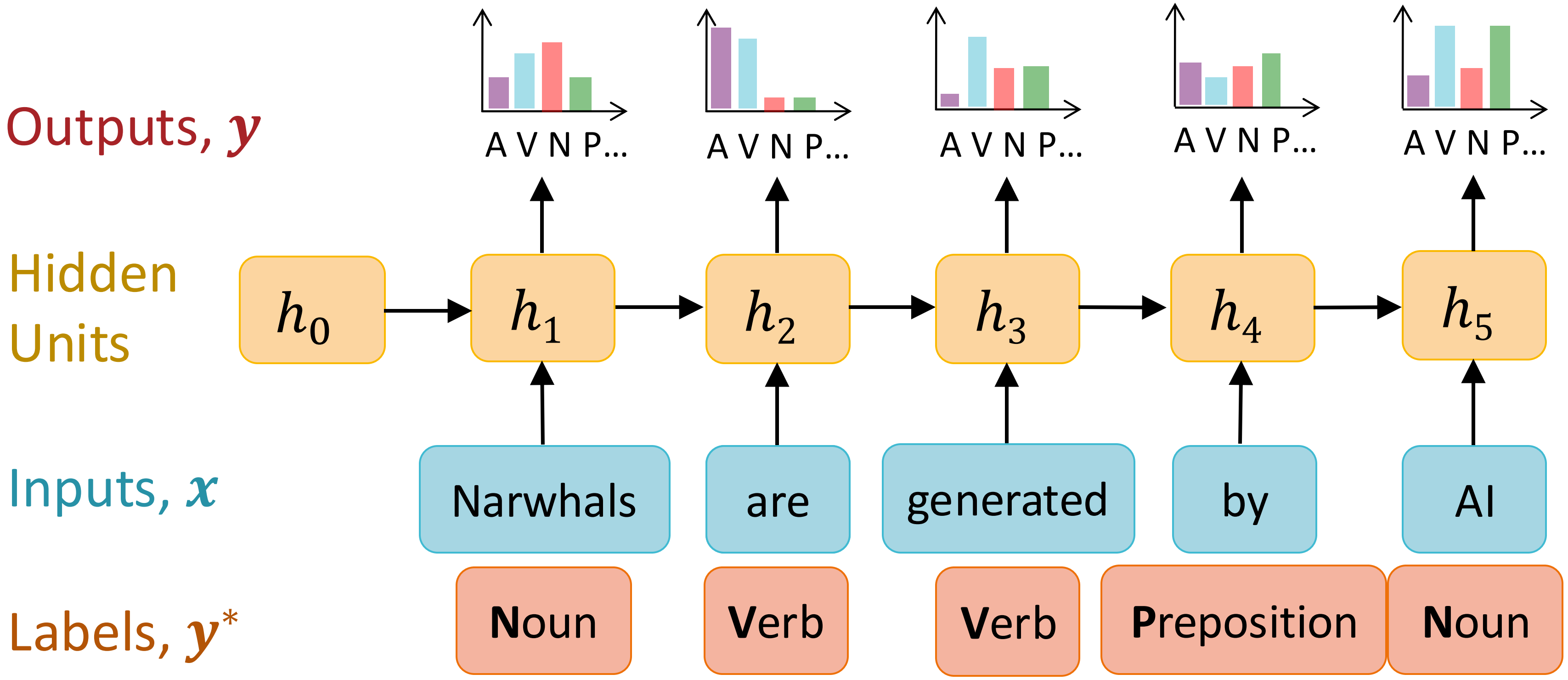
$$y_t = \psi(W_{hy}h_t + b_y)$$



Recurrent Neural Networks for Part of Speech Tagging

$$h_t = \phi(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

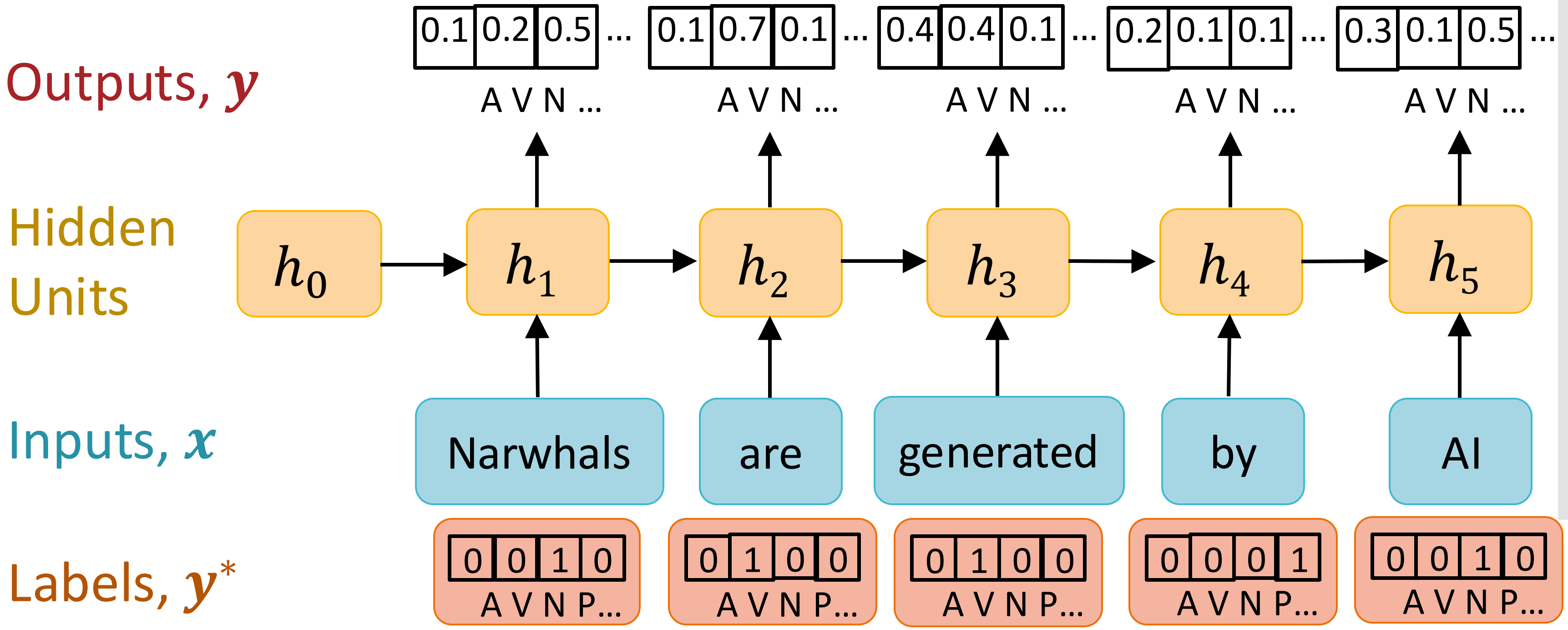
$$y_t = \text{softmax}\left(W_{hy}h_t + b_y\right)$$



Recurrent Neural Networks for Part of Speech Tagging

$$h_t = \phi(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

$$y_t = \text{softmax}\left(W_{hy}h_t + b_y\right)$$

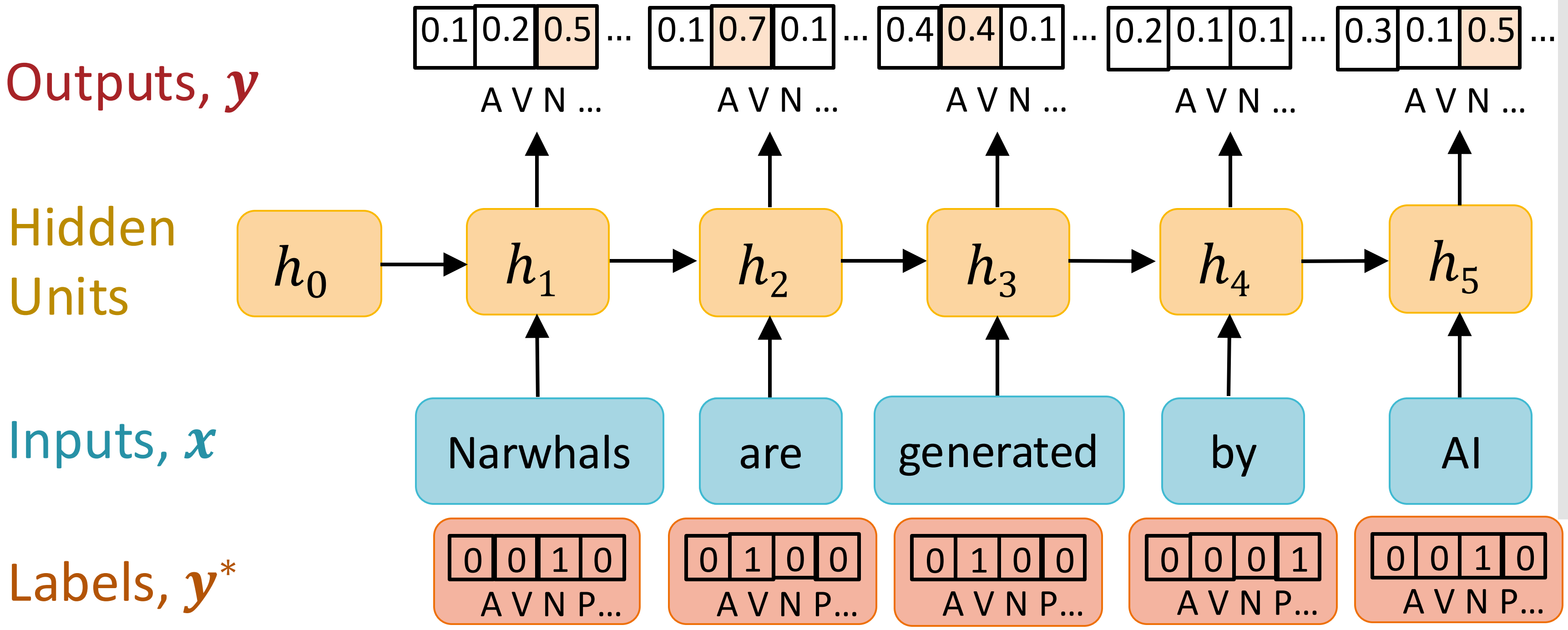


Recurrent Neural Networks for Part of Speech Tagging

$$h_t = \phi(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

$$y_t = \text{softmax}\left(W_{hy}h_t + b_y\right)$$

- Intuition: we want the true label to have high probability under the output distribution
- Idea: use $\boldsymbol{y}^*$ to index into the desired element of $\boldsymbol{y}$

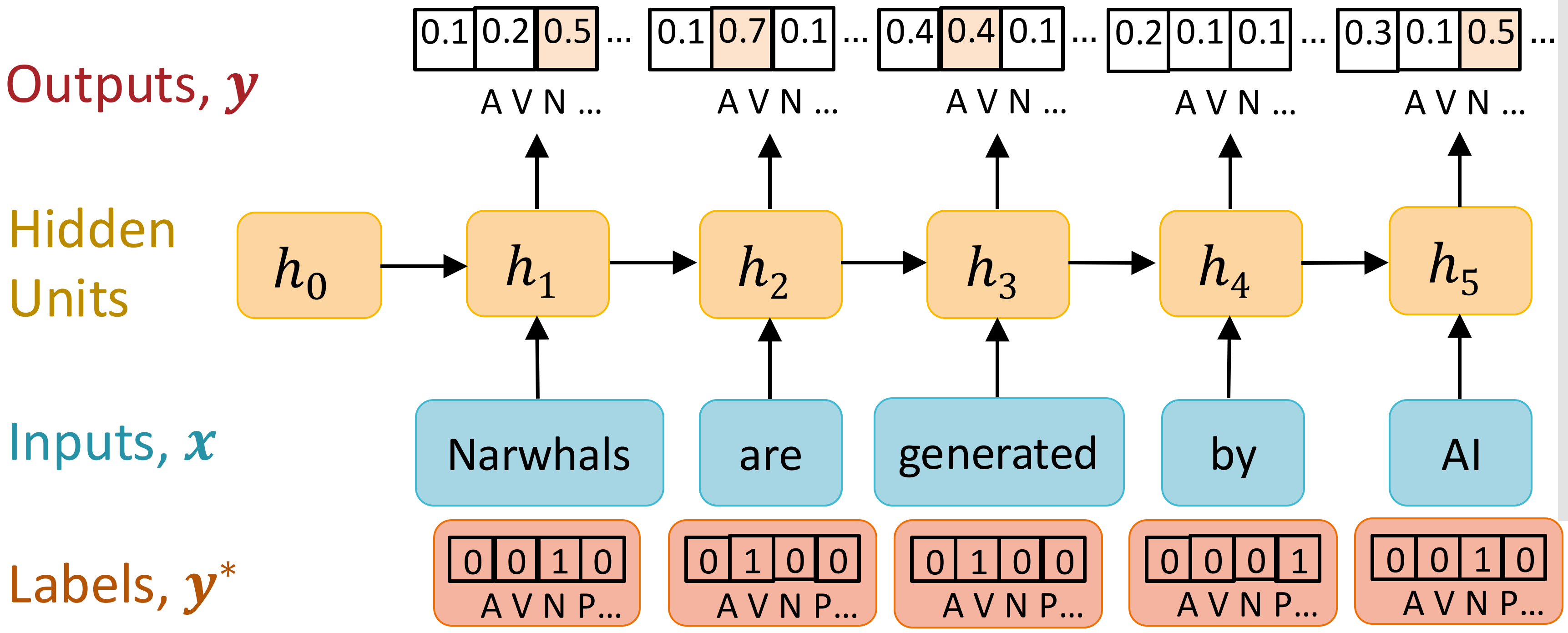


Recurrent Neural Networks for Part of Speech Tagging

$$h_t = \phi(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

$$y_t = \text{softmax}(W_{hy}h_t + b_y)$$

$$\text{maximize} \sum_{c=1}^{C} \boldsymbol{y}_t^*[c] \log \boldsymbol{y}_t[c]$$

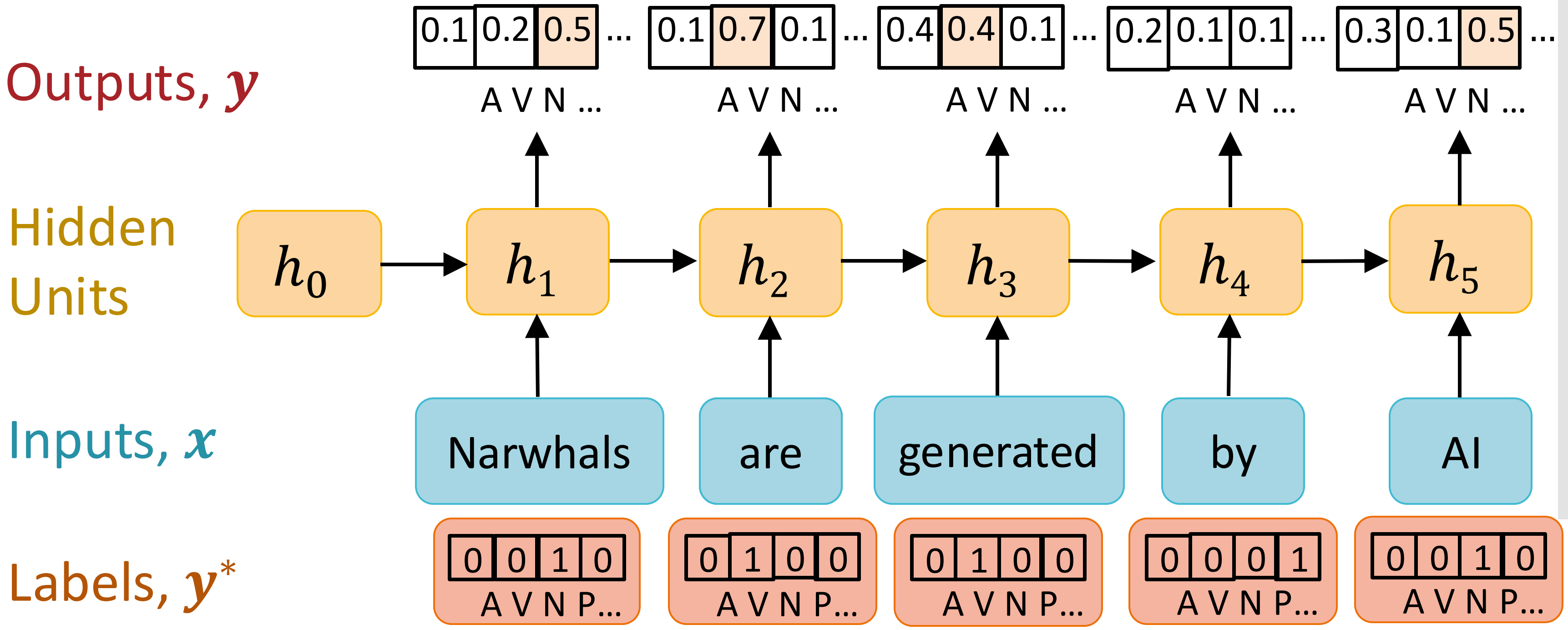


Recurrent Neural Networks for Part of Speech Tagging

$$h_t = \phi(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

$$y_t = \mathrm{softmax}(W_{hy}h_t + b_y)$$

$$\text{minimize } \ell_t = -\sum_{c=1}^{C} \boldsymbol{y}_t^*[c] \log \boldsymbol{y}_t[c]$$

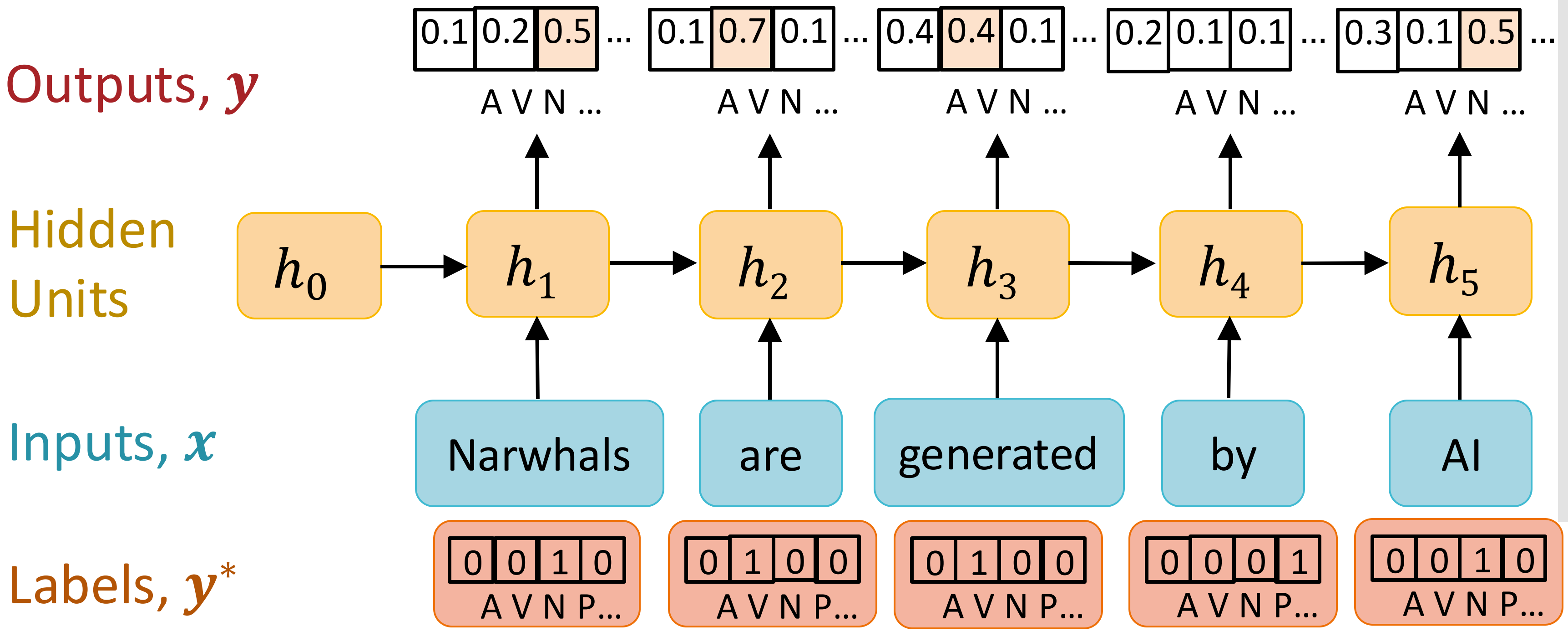


Recurrent Neural Networks for Part of Speech Tagging

$$h_t = \phi(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

$$y_t = \mathrm{softmax}\big(W_{hy}h_t + b_y\big)$$

$$\text{minimize } J = \sum_{t=1}^{T} \ell_t = \sum_{t=1}^{T}\left(-\sum_{c=1}^{C} \boldsymbol{y}_t^*[c] \log \boldsymbol{y}_t[c]\right)$$

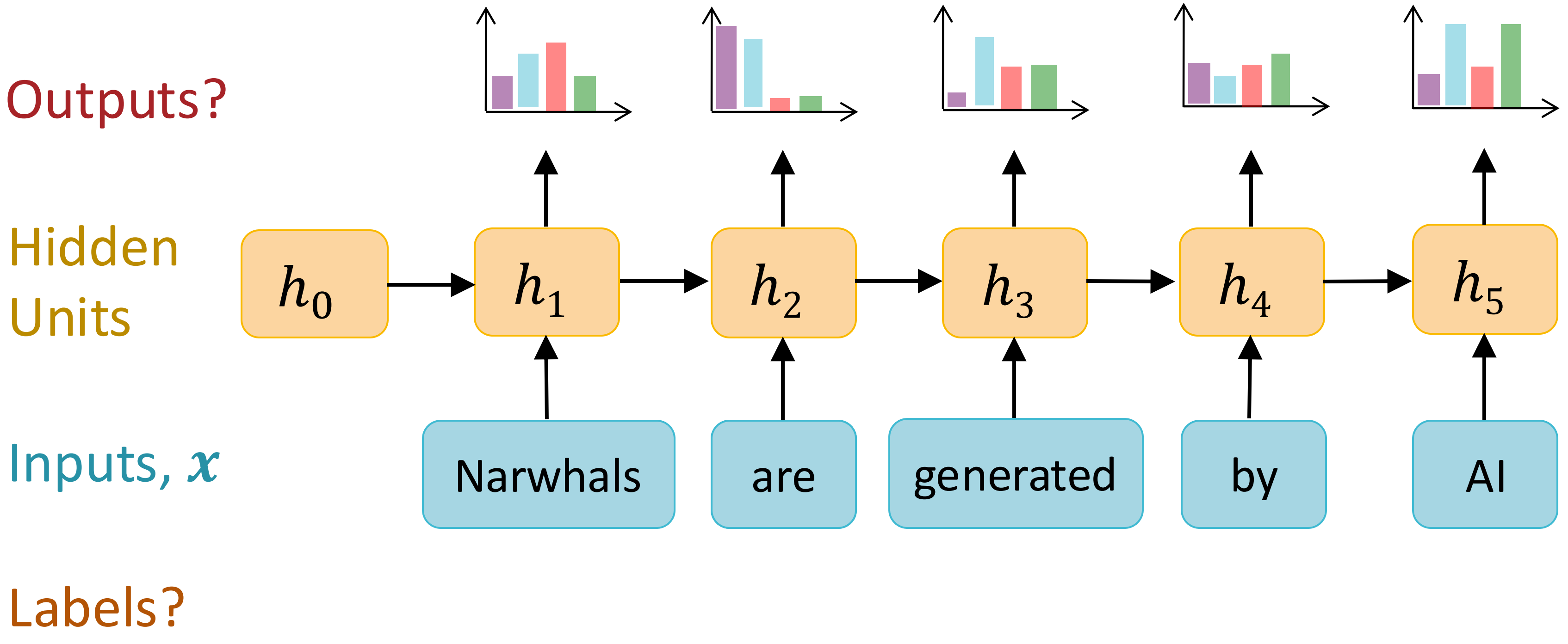


Recurrent Neural Network Language Models: Loss

$$h_t = \phi(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

$$y_t = \text{softmax}\left(W_{hy}h_t + b_y\right)$$

$$\text{minimize } J = \sum_{t=1}^{T} \ell_t = \sum_{t=1}^{T}\left(-\sum_{c=1}^{C} \boldsymbol{y}_t^*[c] \log \boldsymbol{y}_t[c]\right)$$

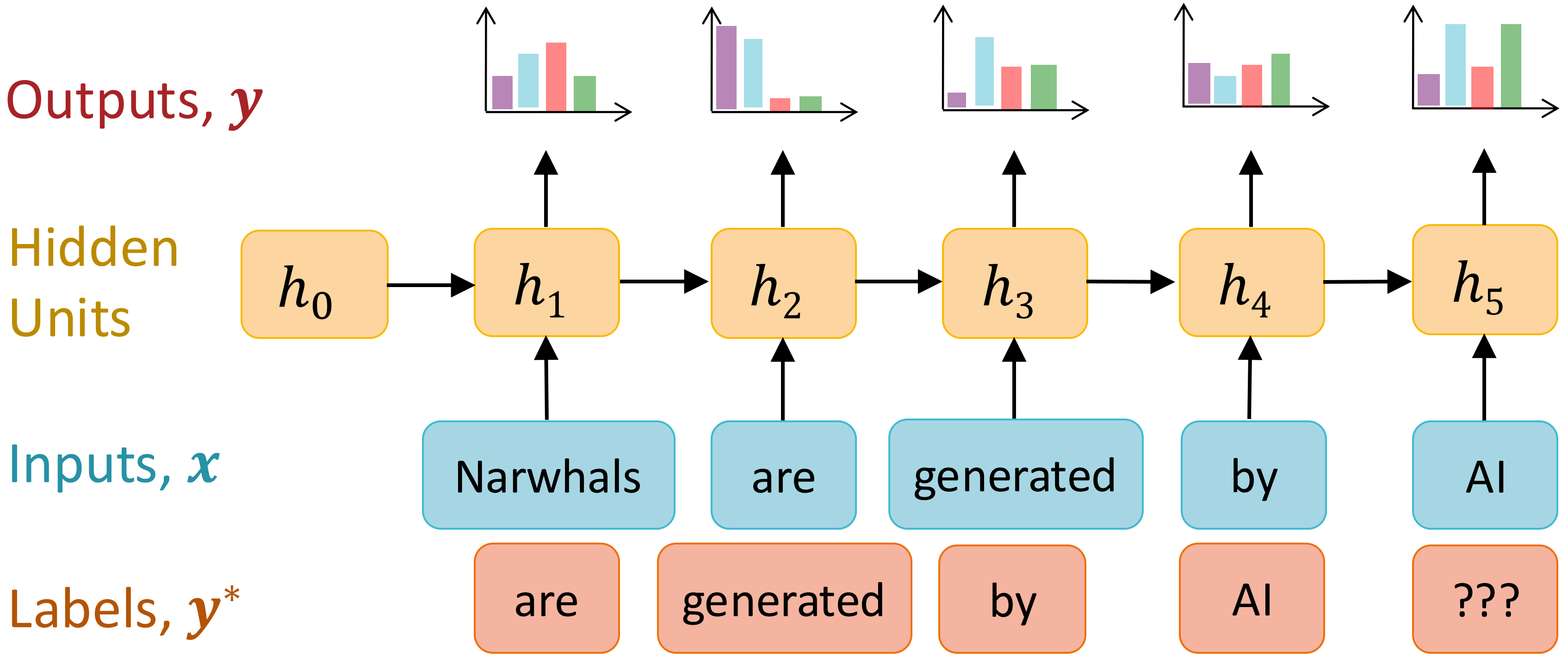


Recurrent Neural Network Language Models: Loss

$$h_t = \phi(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

$$y_t = \mathrm{softmax}\big(W_{hy}h_t + b_y\big)$$

$$\text{minimize } J = \sum_{t=1}^{T} \ell_t = \sum_{t=1}^{T}\left(-\sum_{c=1}^{C} \boldsymbol{y}_t^*[c] \log \boldsymbol{y}_t[c]\right)$$

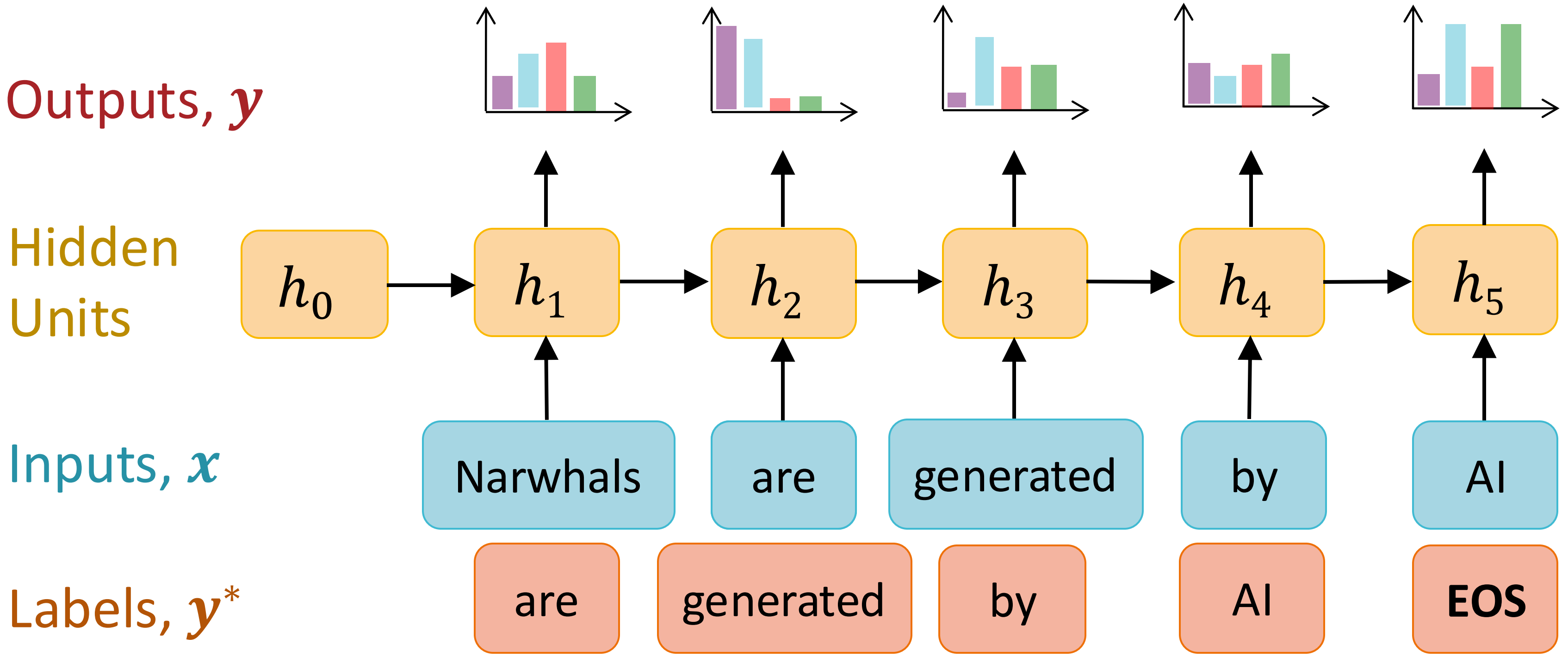


Recurrent Neural Network Language Models: Loss

$$h_t = \phi(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

$$y_t = \mathrm{softmax}\left(W_{hy}h_t + b_y\right)$$

$$\text{minimize } J = \sum_{t=1}^{T} \ell_t = \sum_{t=1}^{T}\left(-\sum_{c=1}^{C} \boldsymbol{y}_t^*[c] \log \boldsymbol{y}_t[c]\right)$$

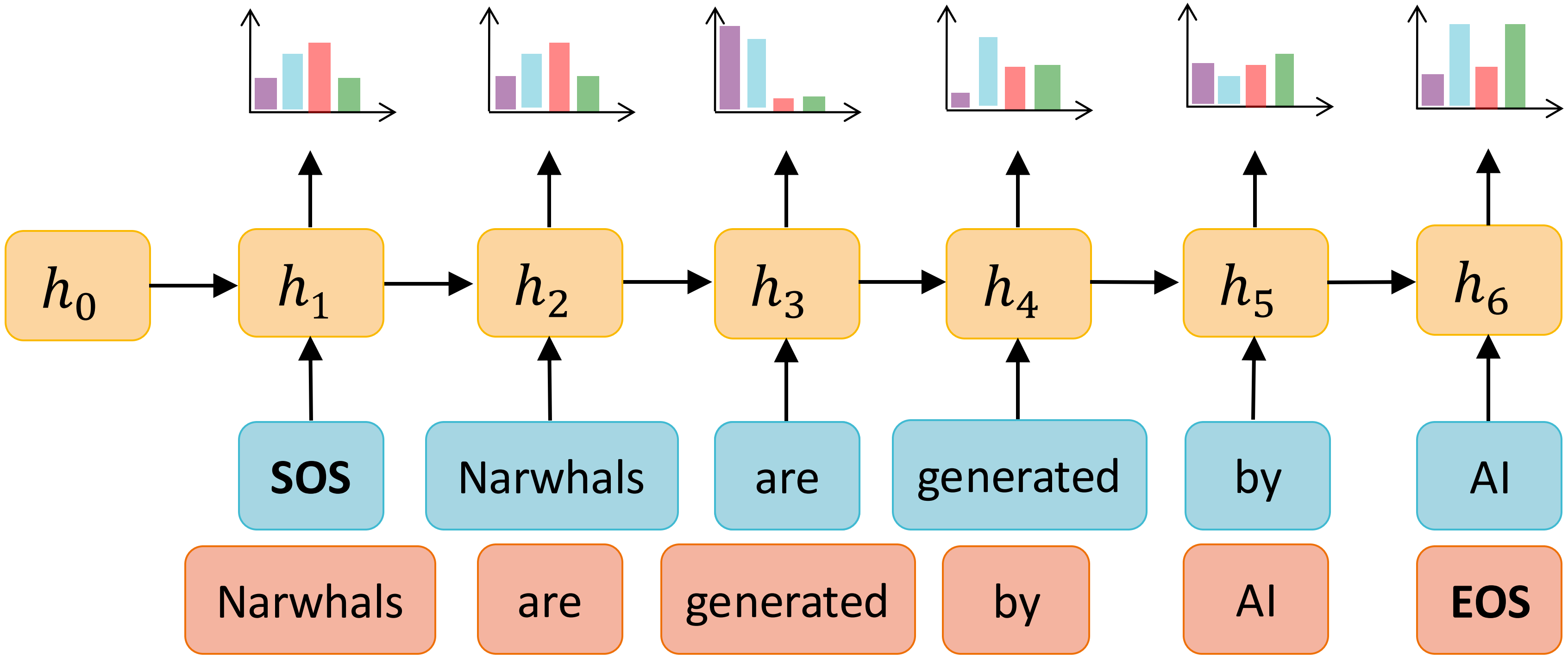


Recurrent Neural Network Language Models: Loss

$$h_t = \phi(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

$$y_t = \text{softmax}\left(W_{hy}h_t + b_y\right)$$

$$\text{minimize } J = \sum_{t=1}^{T} \ell_t = \sum_{t=1}^{T}\left(-\sum_{c=1}^{C} \boldsymbol{y}_t^*[c] \log \boldsymbol{y}_t[c]\right)$$



Recurrent Neural Network Language Models: Training

- Each training data point is a *sequence* $\boldsymbol{x}^{(i)} = \left[\boldsymbol{x}_1^{(i)}, \ldots, \boldsymbol{x}_{T_i}^{(i)}\right]$

- The objective function is the log-likelihood of a mini-batch:

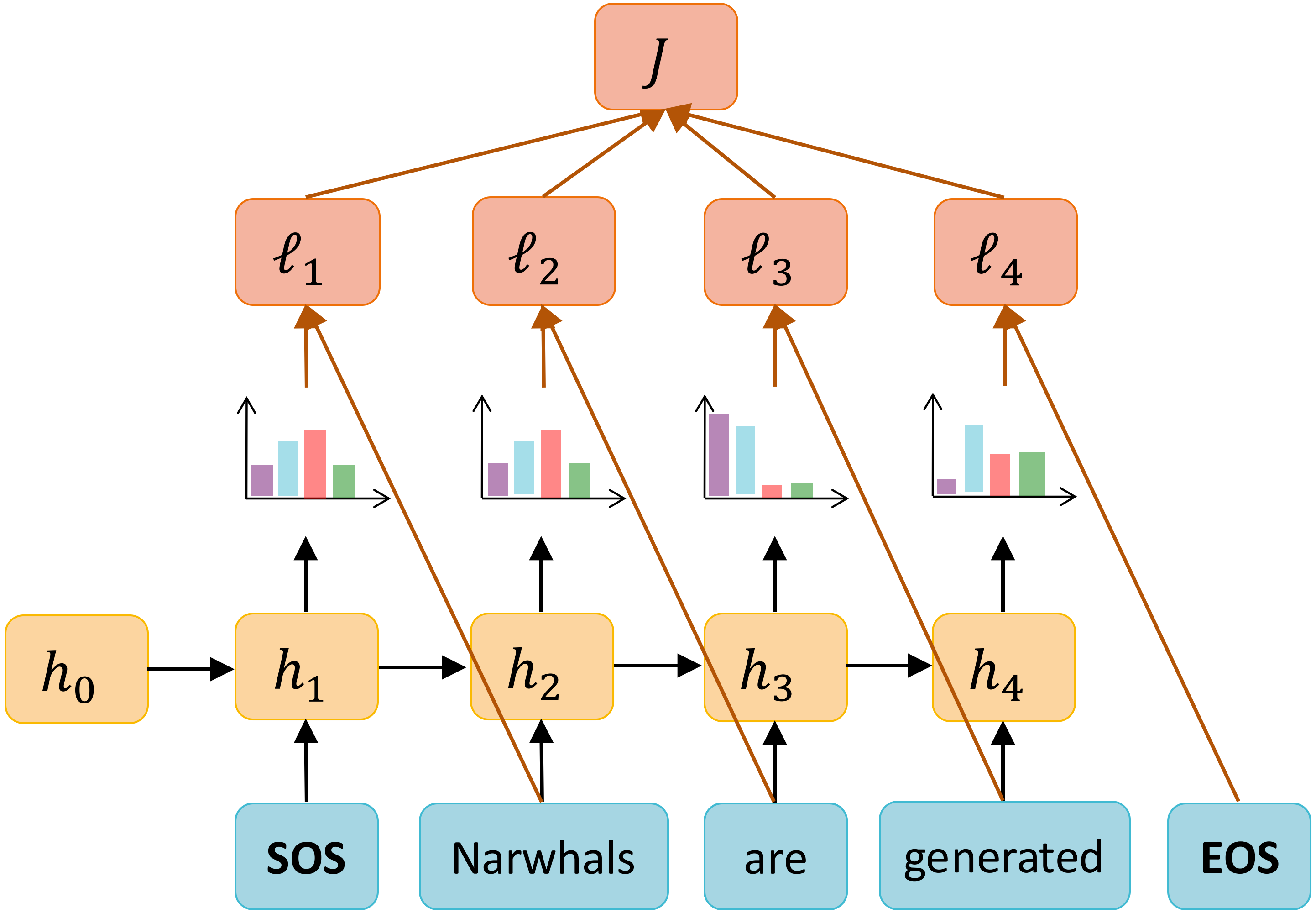
$$J^{(B)}(\boldsymbol{\theta}) = \log \prod_{b=1}^{B} p_{\boldsymbol{\theta}}(\boldsymbol{x}^{(b)}) = \sum_{b=1}^{B} \log p_{\boldsymbol{\theta}}(\boldsymbol{x}^{(b)})$$

(assuming i.i.d. sequences) where

$$\log p_{\boldsymbol{\theta}}(\boldsymbol{x}^{(b)}) \coloneqq \log p_{\boldsymbol{\theta}}\left(\boldsymbol{x}_1^{(b)} \middle| \boldsymbol{h}_1\right) + \cdots + \log p_{\boldsymbol{\theta}}\left(\boldsymbol{x}_{T_b}^{(b)} \middle| \boldsymbol{h}_{T_b}\right)$$

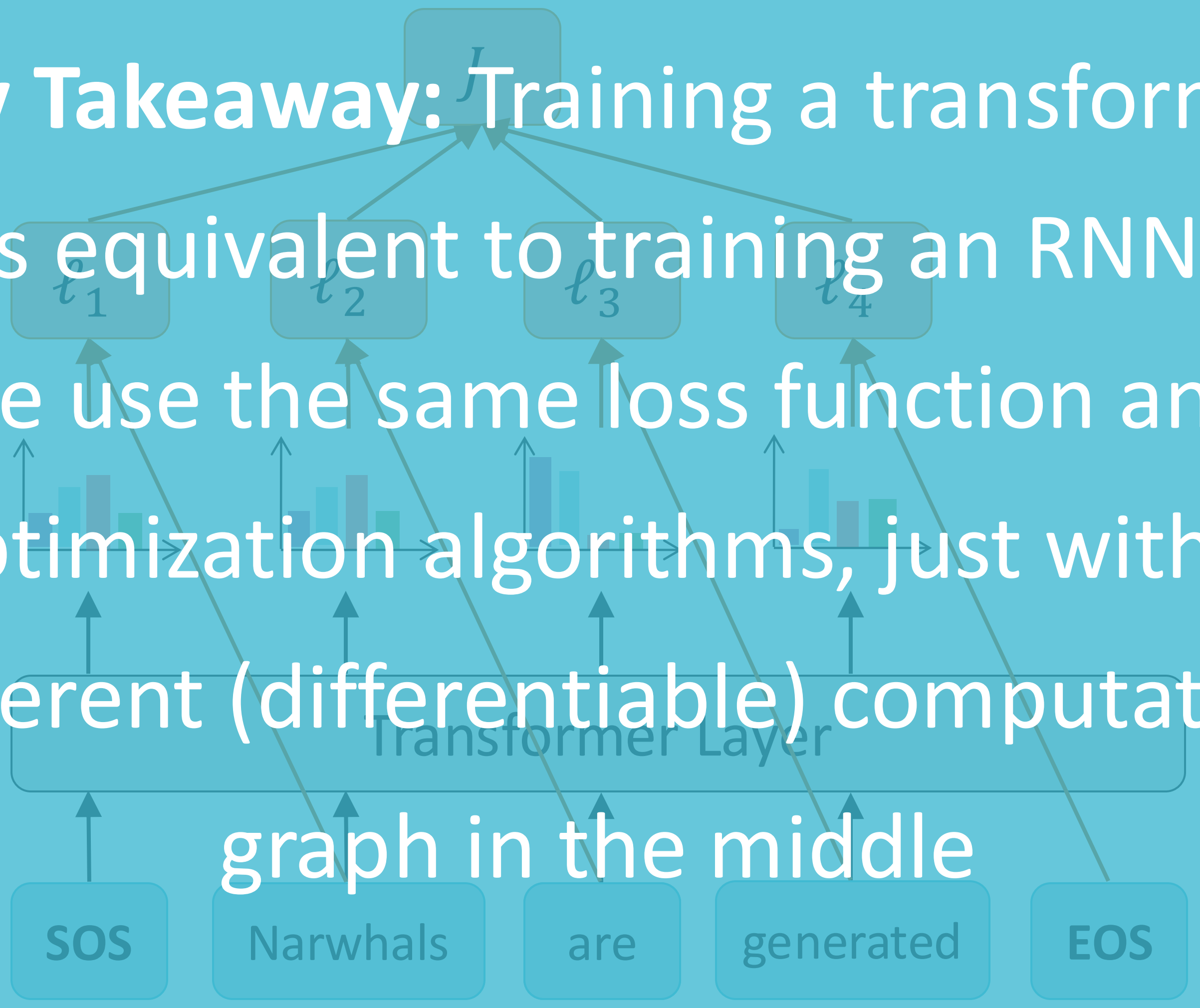
$$\coloneqq l_1 + \cdots + l_{T_b}$$

Recurrent Neural Network Language Models: Training

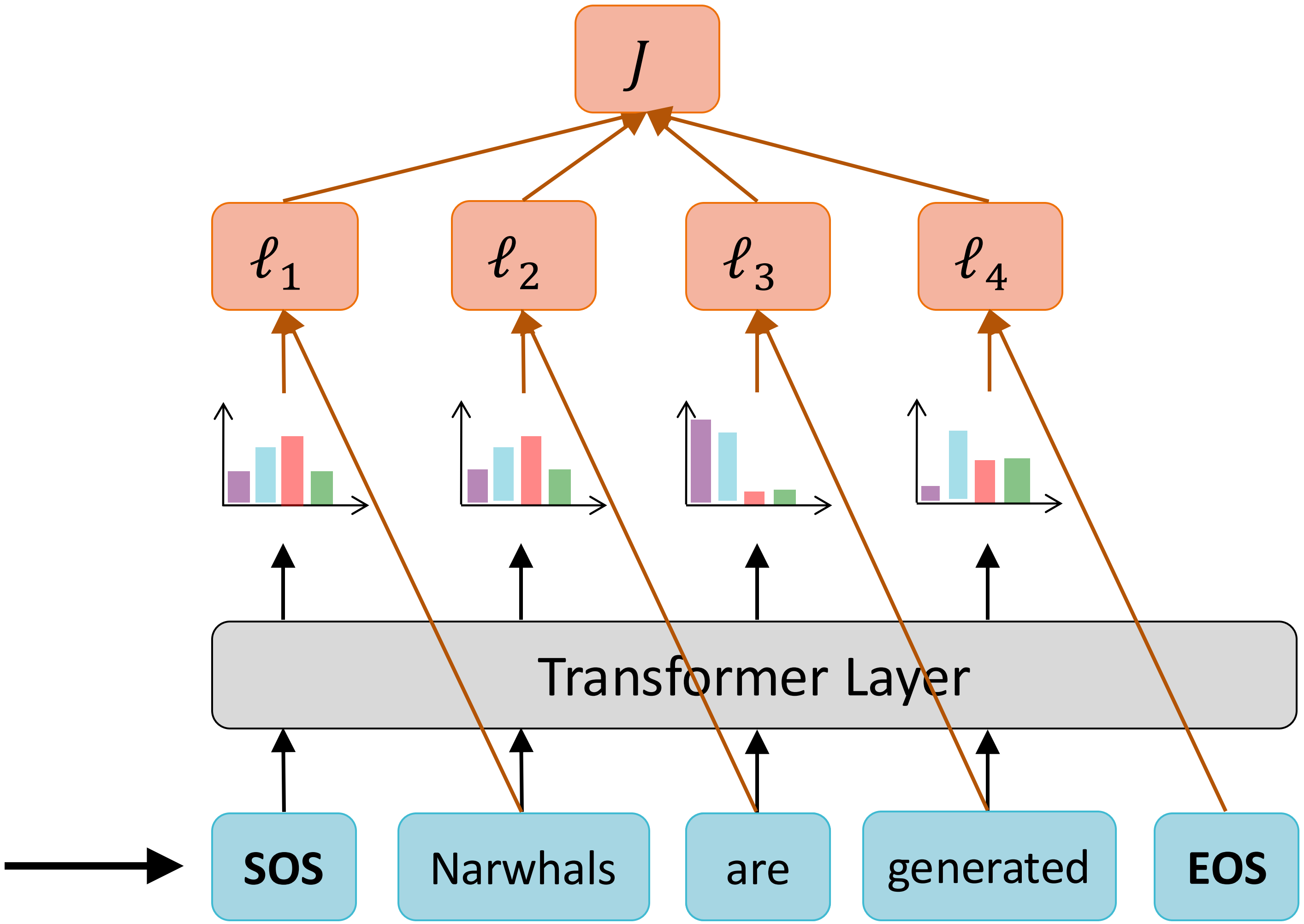


Transformer Language Models: Training

Key Takeaway: Training a transformer LM is equivalent to training an RNN LM: we use the same loss function and optimization algorithms, just with a different (differentiable) computation graph in the middle



Are we really passing in “words” to this transformer?



Tokenization

- How can we break a sequence of text into individual units?
  - Example: “Henry is giving a lecture on transformers”
  - Word-based tokenization:

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

Tokenization

- How can we break a sequence of text into individual units?
  - Example: “Henry is givin’ a lectrue on transformers”
  - Word-based tokenization:

    [“henry”, “is”, ???, “a”, ???, “on”, “transformers”]
    - 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

- How can we break a sequence of text into individual units?
  - Example: “Henry is givin’ a lectrue on transformers”
  - Character-based tokenization:

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

- 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

- How can we break a sequence of text into individual units?
  - Example: “Henry is givin’ a lectrue on transformers”
  - Subword tokenization:

[“henry”, “is”, “giv”, “##in”, “ ‘ ”, “a”, “lect”, “#u”, “##re”, “on”, “transform”, “##ers”]

  - 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 (all individual characters are subwords)

Okay, but these are still strings: how do I convert these into things my transformer can work with?

- How can we break a sequence of text into individual units?
  - Example: “Henry is givin’ a lectrue on transformers”
  - Subword tokenization:

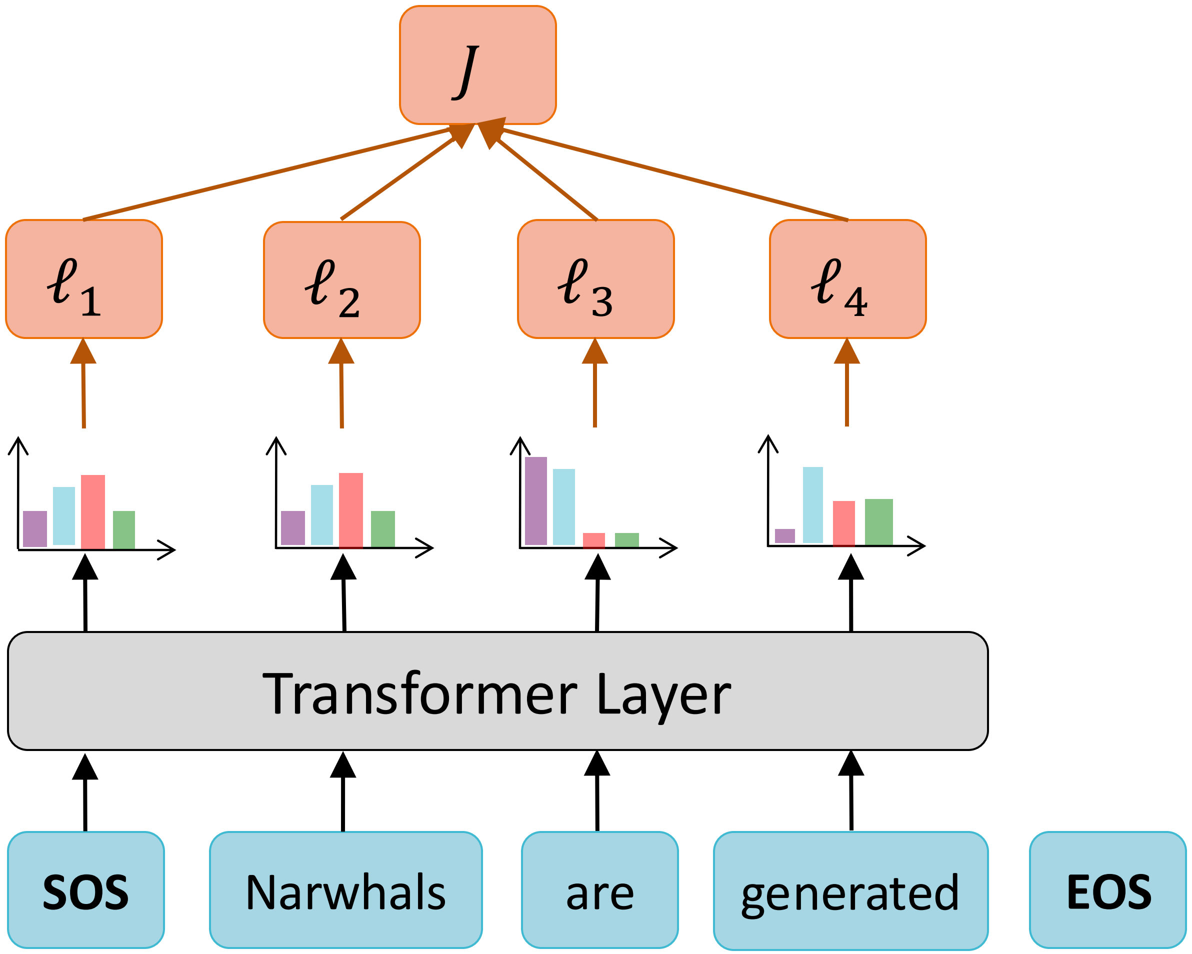
[“henry”, “is”, “giv”, “##in”, “ ‘ ”, “a”, “lect”, “#u”, “##re”, “on”, “transform”, “##ers”]

- 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 (all individual characters are subwords)

Embeddings

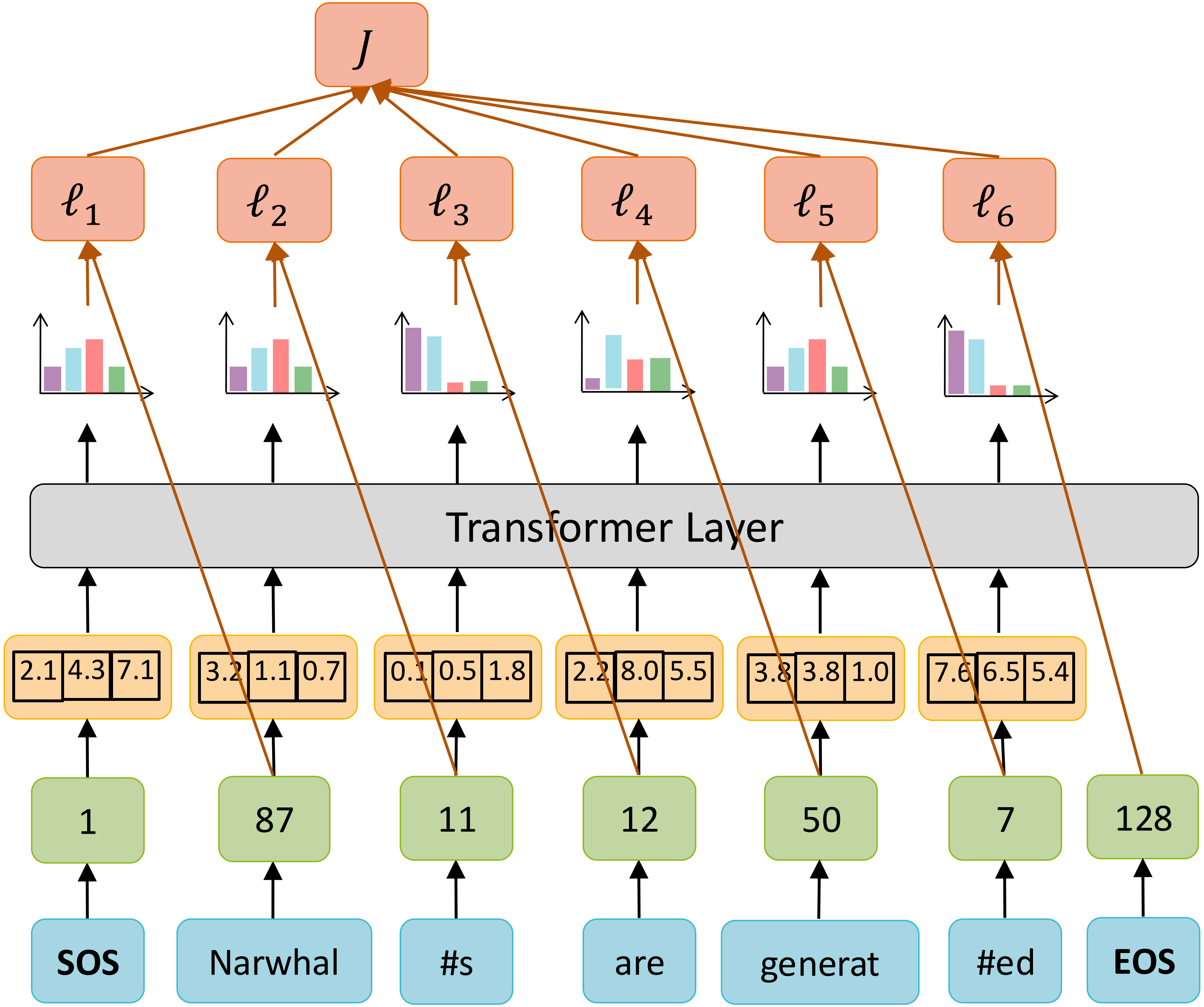
- Given a vocabulary V with $|V|$ tokens:
  1. Map each token to a (non-negative) integer
  2. Define a $|V| \times d_e$ lookup table, where each row is a dense, numerical vector of length d_e
  3. The row corresponding to each token's integer assignment is that token's *embedding*

Are we really passing in "words" to this transformer?



Are we really passing in “words” to this transformer?

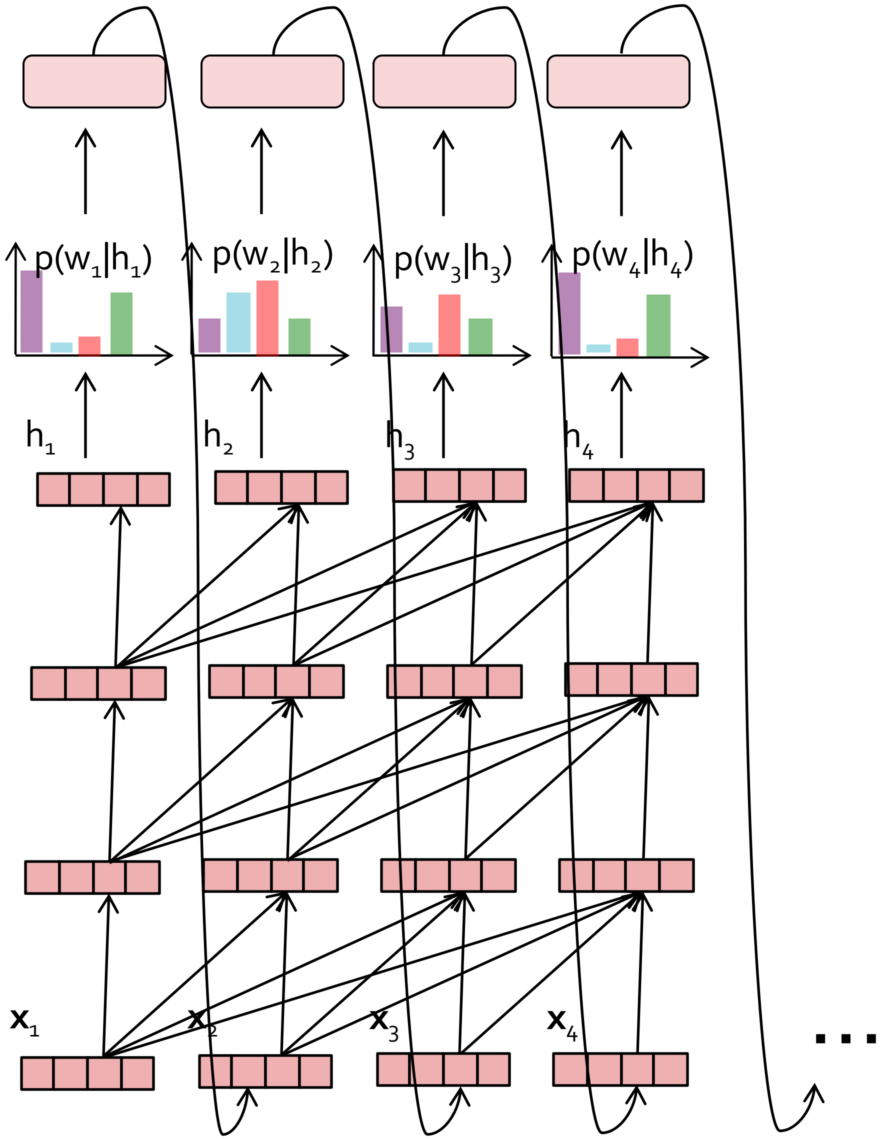
NO



Recall: Transformer Computational Complexity

Important!

- RNN computation graph grows **linearly** with the number of input tokens
- Transformer LM computation graph grows **quadratically** with the number of input tokens
- However, this computation (and therefore, the training of transformer LMs) is **highly parallelizable**



Parallelizing Transformer LM Computation

- **Scaled dot-product attention** can be easily parallelized because the attention scores at one timestep do not depend on other timesteps.
- In **multi-headed attention**, each head is also independent of the other heads, which permits yet more parallelism.
- The core computation in attention is **matrix multiplication**, and GPUs/TPUs make this very fast.
- **Model parallelism:** for large models, we can divide the model over multiple GPUs/machines.
- **Key-value caching**: keys and values are re-used over many timesteps so we can cache them for faster access
- **Batching**: rather than process one sequence at a time, transformers take in a *batch*; the computation is identical for each sequence **(if they're of the same length)**

Parallelizing Transformer LM Computation

- **Scaled dot-product attention** can be easily parallelized because the attention scores at one timestep do not depend on other timesteps.
- In **multi-headed attention**, each head is also independent of the other heads, which permits yet more parallelism.
- The core computation in attention is **matrix multiplication**, and GPUs/TPUs make this very fast.
- **Model parallelism:** for large models, we can divide the model over multiple GPUs/machines.
- **Key-value caching**: keys and values are re-used over many timesteps so we can cache them for faster access
- **Batching**: rather than process one sequence at a time, transformers take in a *batch*; the computation is identical for each sequence **(if they're of the same length)**

Batching: Padding & Truncation

- Given a block size or maximum length, L (typically a power of 2):
  - Truncate sequences longer than L by deleting excess tokens
  - Pad sequences shorter than L by adding **PAD** tokens

$x_1^{(i)}$	$x_2^{(i)}$	$x_3^{(i)}$	$x_4^{(i)}$	$x_5^{(i)}$	$x_6^{(i)}$	$x_7^{(i)}$	$x_8^{(i)}$	$x_9^{(i)}$	$x_{10}^{(i)}$
Narwhals	are	generated	by	AI					
Watch	out	,	the	narwhals	are	coming	!		
How	many	sequences	contain	“	narwhals	are	”	?	
Narwhals	are	way	cooler	than	axolotls				
Of	the	large	aquatic	mammals	,	narwhals	are	the	best
Who	knows	what	the	narwhals	are	hiding	?		

Batching: Padding & Truncation

- Given a block size or maximum length, L (typically a power of 2):
  - Truncate sequences longer than L by deleting excess tokens
  - Pad sequences shorter than L by adding **PAD** tokens

$x_1^{(i)}$	$x_2^{(i)}$	$x_3^{(i)}$	$x_4^{(i)}$	$x_5^{(i)}$	$x_6^{(i)}$	$x_7^{(i)}$	$x_8^{(i)}$
Narwhals	are	generated	by	AI			
Watch	out	,	the	narwhals	are	coming	!
How	many	sequences	contain	“	narwhals	are	”
Narwhals	are	way	cooler	than	axolotls		
Of	the	large	aquatic	mammals	,	narwhals	are
Who	knows	what	the	narwhals	are	hiding	?

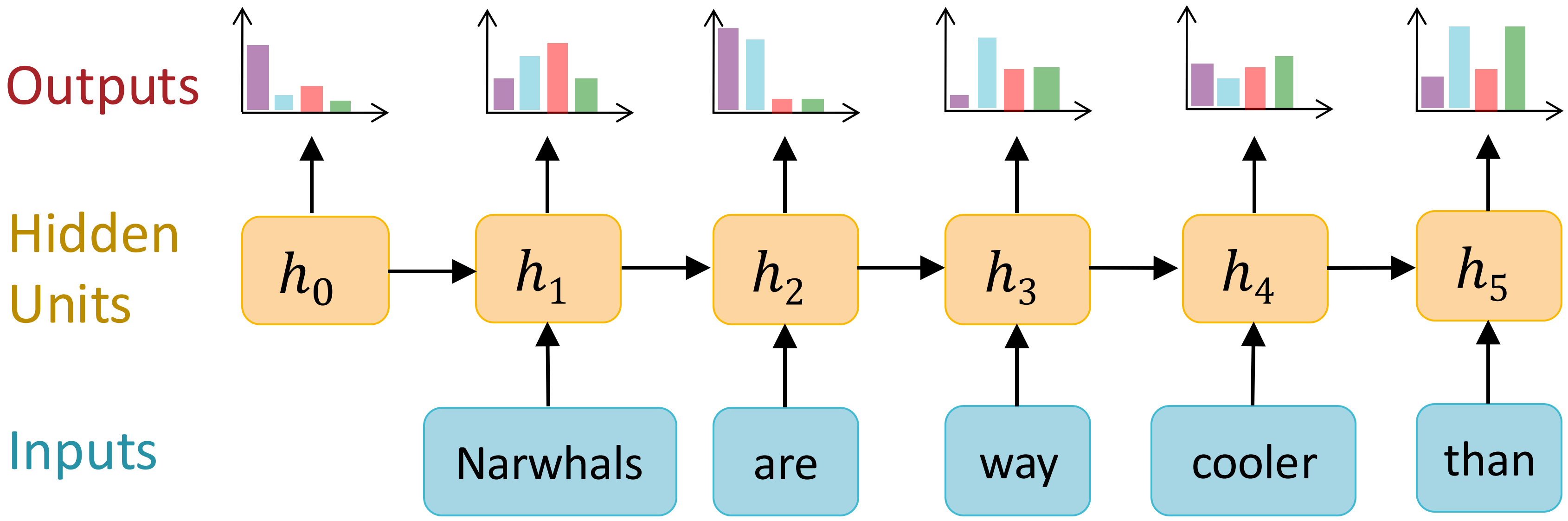
Batching: Padding & Truncation

- Given a block size or maximum length, L (typically a power of 2):
  - Truncate sequences longer than L by deleting excess tokens
  - Pad sequences shorter than L by adding **PAD** tokens

$x_1^{(i)}$	$x_2^{(i)}$	$x_3^{(i)}$	$x_4^{(i)}$	$x_5^{(i)}$	$x_6^{(i)}$	$x_7^{(i)}$	$x_8^{(i)}$
Narwhals	are	generated	by	AI	**PAD**	**PAD**	**PAD**
Watch	out	,	the	narwhals	are	coming	!
How	many	sequences	contain	“	narwhals	are	”
Narwhals	are	way	cooler	than	axolotls	**PAD**	**PAD**
Of	the	large	aquatic	mammals	,	narwhals	are
Who	knows	what	the	narwhals	are	hiding	?

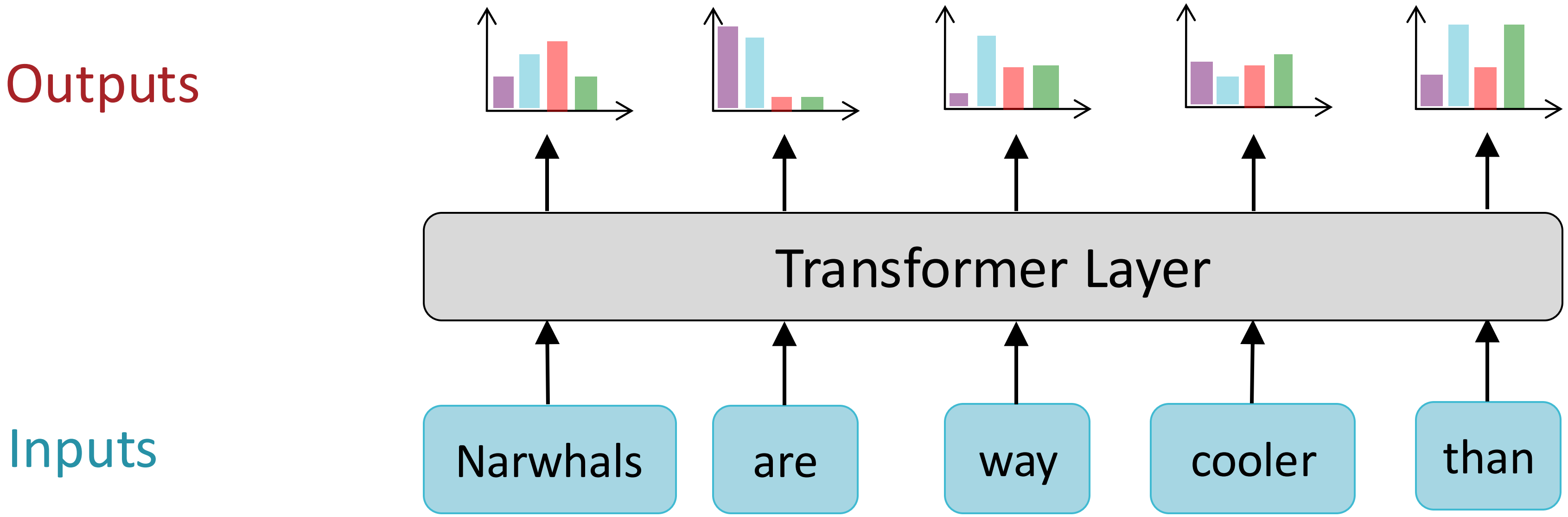
Recall: Language Model Generation

- How do we generate new sequences using an RNN language model?
- Exactly the same way we did for an n-gram language model, by sampling from some learned probability distributions over next words!



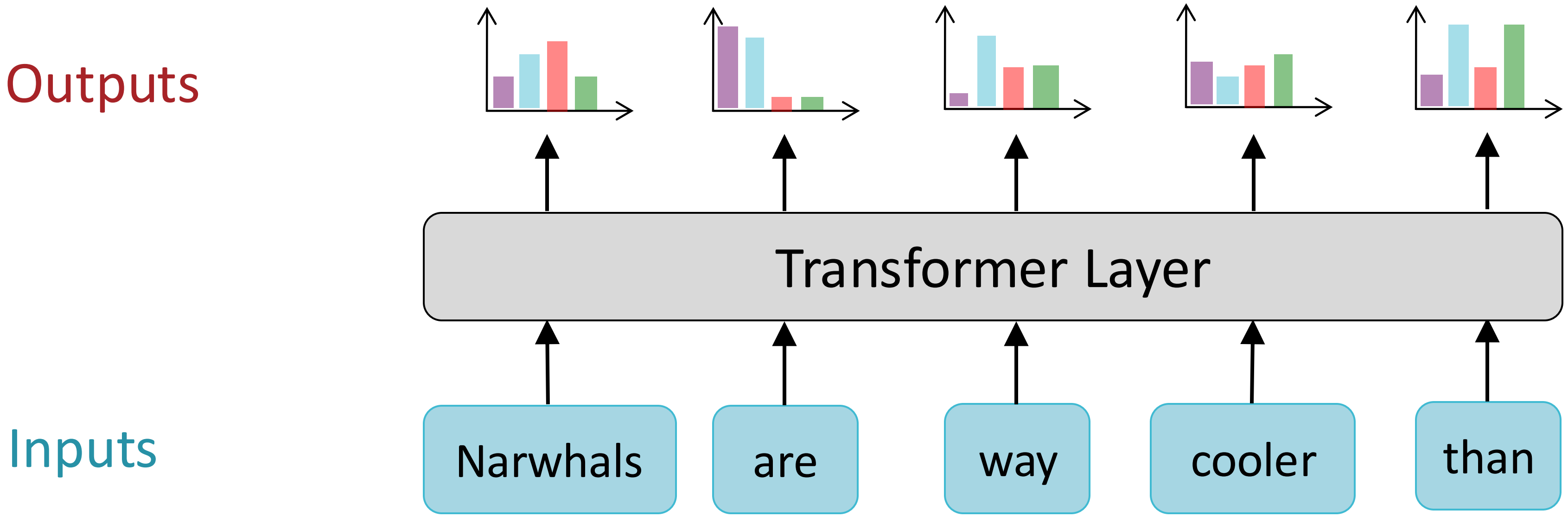
Recall: Language Model Generation

- How do we generate new sequences using a transformer language model?
- Exactly the same way we did for an RNN language model, by sampling from some learned probability distributions over next words!



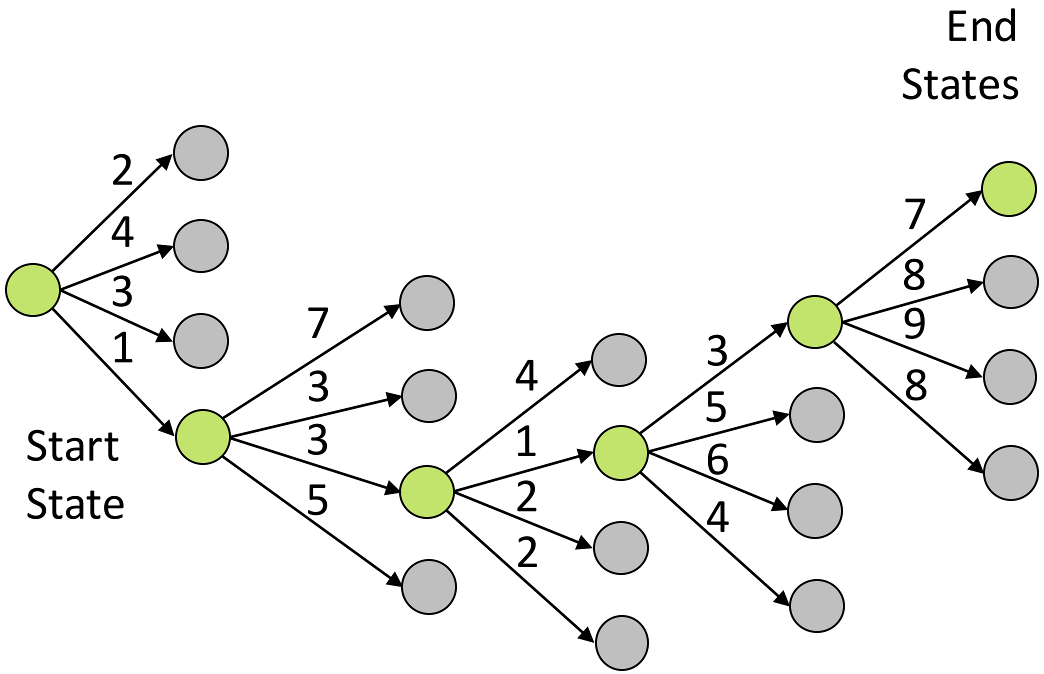
Is this the only thing we could do?

- How do we generate new sequences using a transformer language model?
- Exactly the same way we did for an RNN language model, by sampling from some learned probability distributions over next words!



Background: Greedy Search

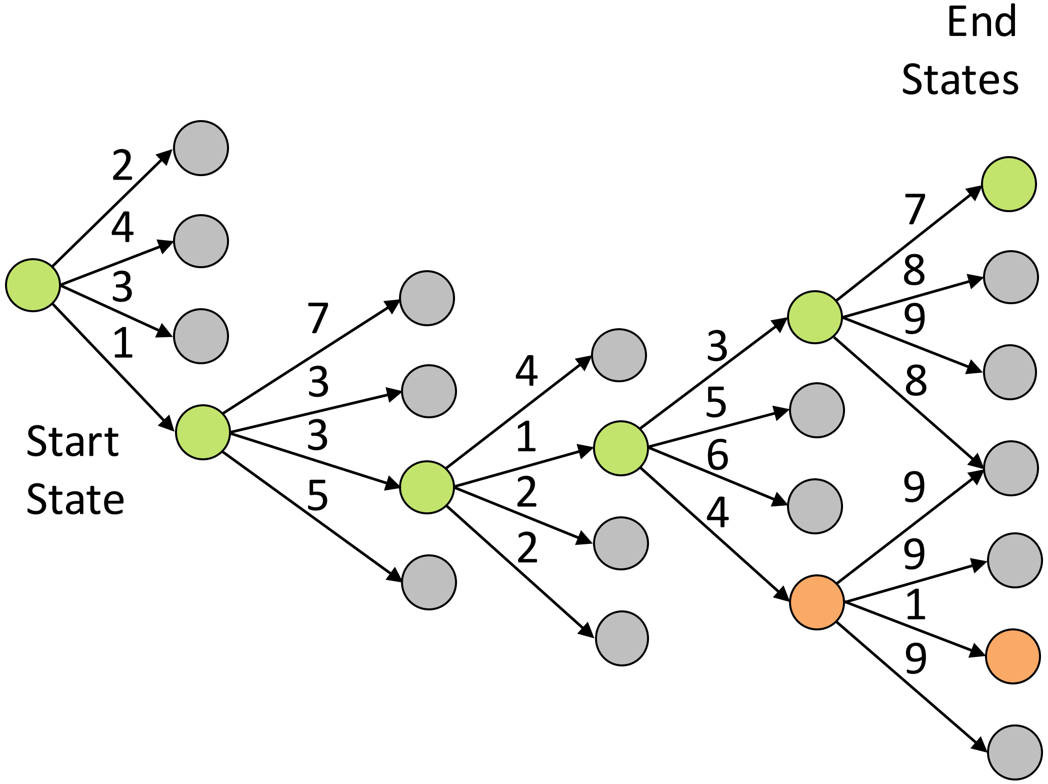
- **Goal**: find the lowest (total) weight path from the Start State to any End State



- **Greedy Search**:
  - At each node, select the edge with lowest weight
  - **Heuristic**: does *not* necessarily find the lowest weight path

Background: Greedy Search

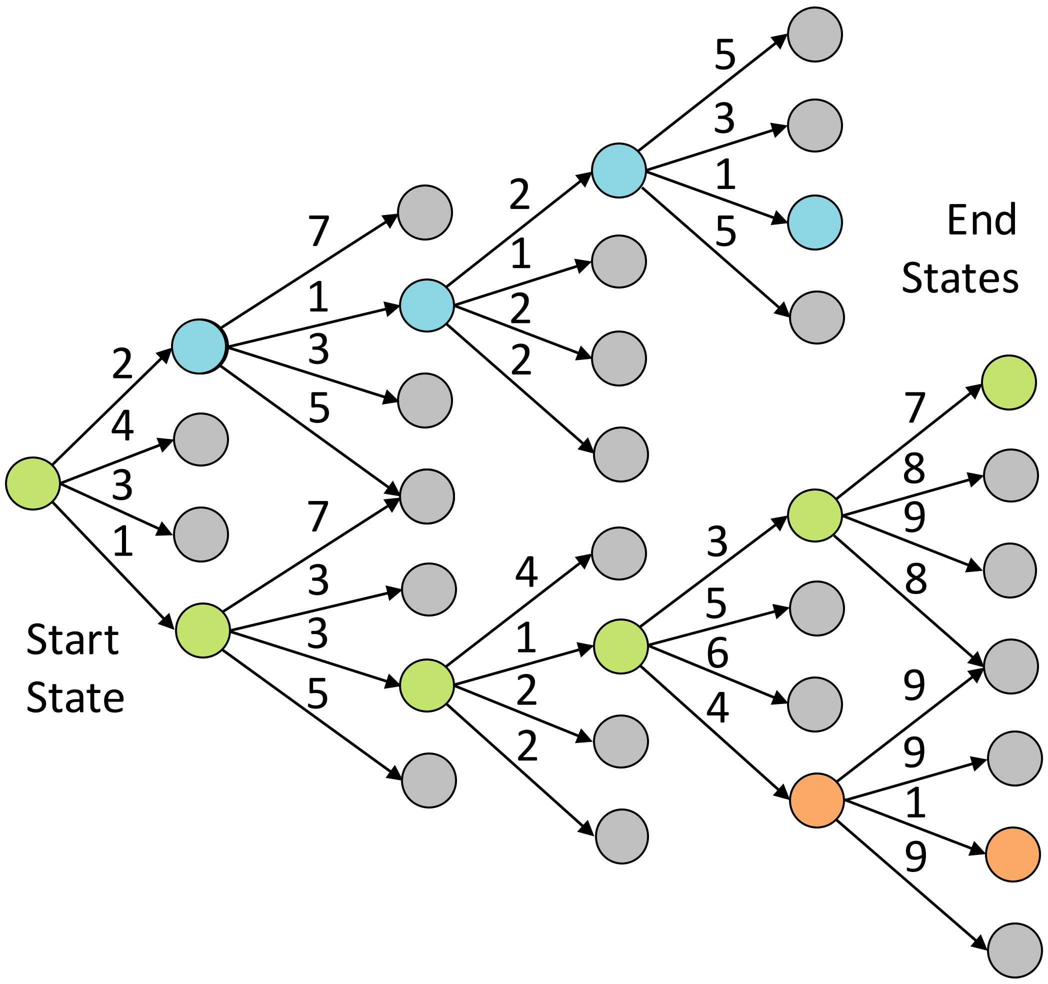
- **Goal**: find the lowest (total) weight path from the Start State to any End State



- **Greedy Search**:
  - At each node, select the edge with lowest weight
  - **Heuristic**: does *not* necessarily find the lowest weight path

Background: Greedy Search

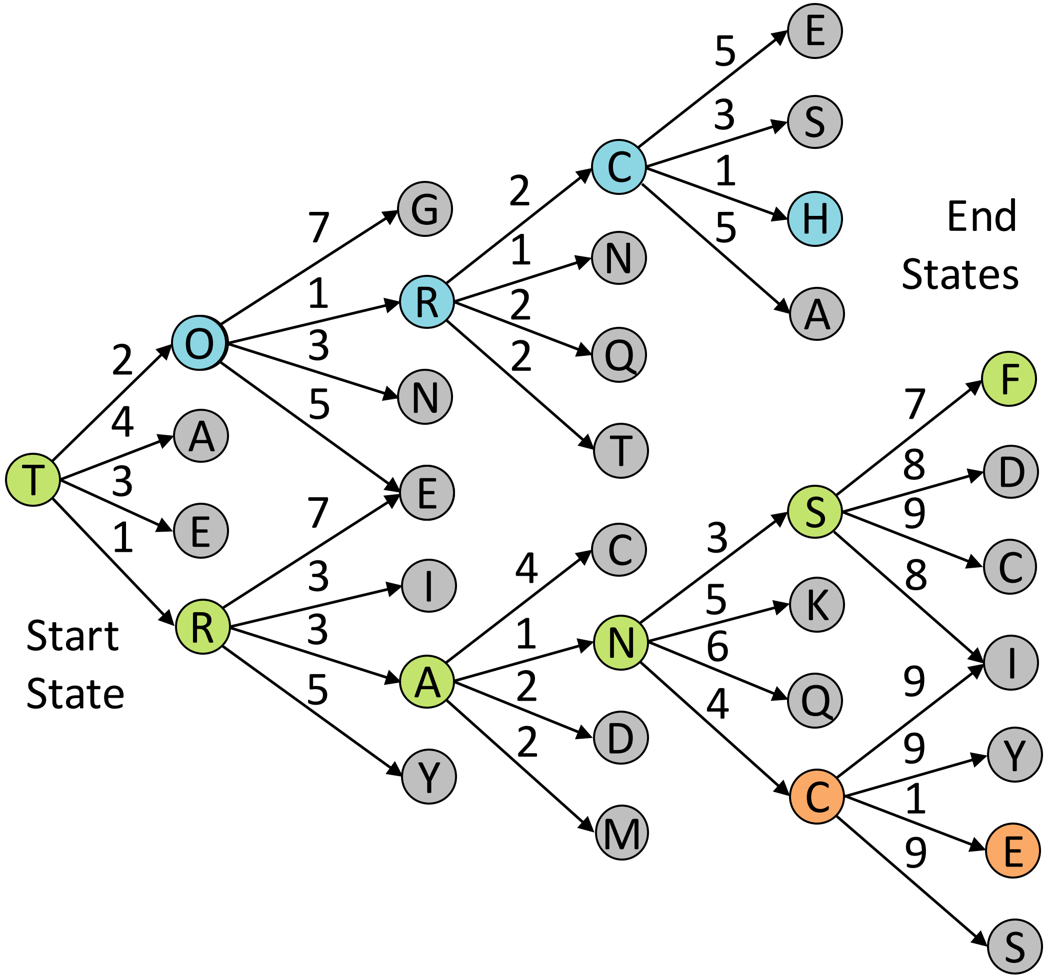
- **Goal**: find the lowest (total) weight path from the Start State to any End State



- **Greedy Search**:
  - At each node, select the edge with lowest weight
  - **Heuristic**: does *not* necessarily find the lowest weight path
  - Computation time is **linear** in max path length

Greedy Decoding for Language Models

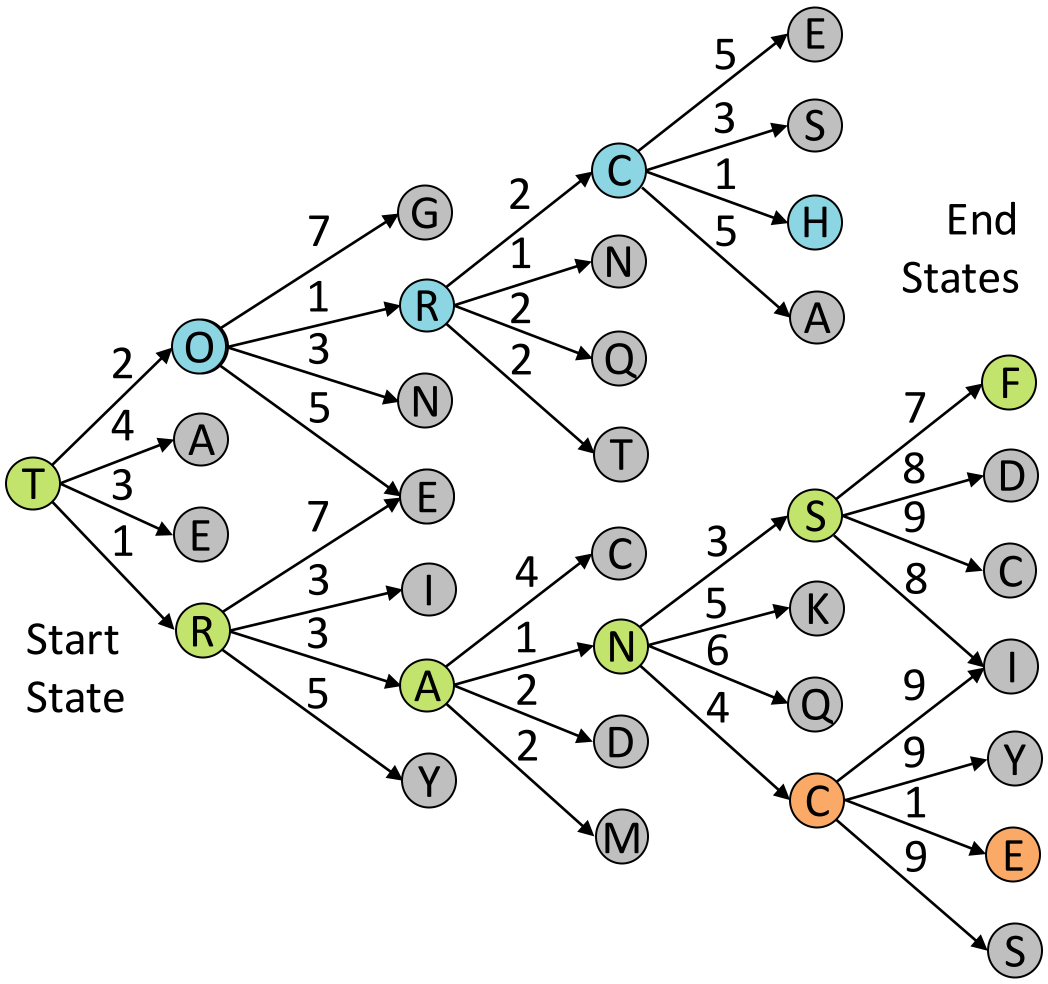
- **Goal**: find the highest probability sequence of tokens
- Nodes are tokens and weights are (negative) log probabilities



- At each node, select the edge with lowest negative log probability
- **Heuristic**: does *not* necessarily find the highest probability output
- Computation time is **linear** in the maximum path length

Ancestral Sampling for Language Models

- **Goal**: find the highest probability sequence of tokens
- Nodes are tokens and weights are (negative) log probabilities



- At each node, sample an edge with probability proportional to the negative exp'ed weights
- **Exact** method of *sampling*
- Computation time is **linear** in the maximum path length