10-701: Introduction to Machine Learning Lecture 13 – Attention & Transformers

Henry Chai

2/28/24

Front Matter

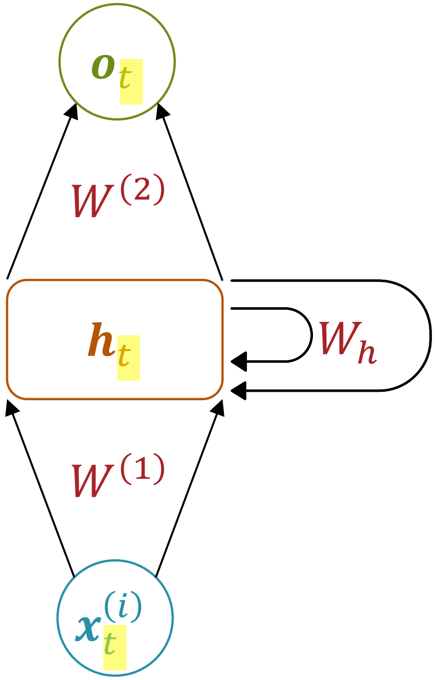
- Announcements
  - HW3 released 2/19, due 2/28 (today!) at 11:59 PM
  - HW4 released 2/28 (today!), due 3/15 (after break) at 11:59 PM
  - Project details will be released 3/1 (Friday)
    - **You must work in groups of 2 or 3 on the project**
- Recommended Readings
  - Zhang, Lipton, Li & Smola, Chapters 9 & 10

Recurrent Neural Networks

- Neural networks are frequently applied to inputs with some inherent temporal or sequential structure (e.g., text or video) of variable length
- Idea: use the information from previous parts of the input to inform subsequent predictions
- Insight: the hidden layers learn a useful representation (relative to the task)
- Approach: incorporate the output from earlier hidden layers into later ones.

Recurrent Neural Networks

$$\boldsymbol{h}_t = \left[1, \theta\left(W^{(1)}\boldsymbol{x}_t^{(i)} + W_h \boldsymbol{h}_{t-1}\right)\right]^T \text{ and } \boldsymbol{o}_t = \hat{y}_t^{(i)} = \theta\left(W^{(2)}\boldsymbol{h}_t\right)$$



- Training dataset consists of (input **sequence**, label **sequence**) pairs, potentially of varying lengths

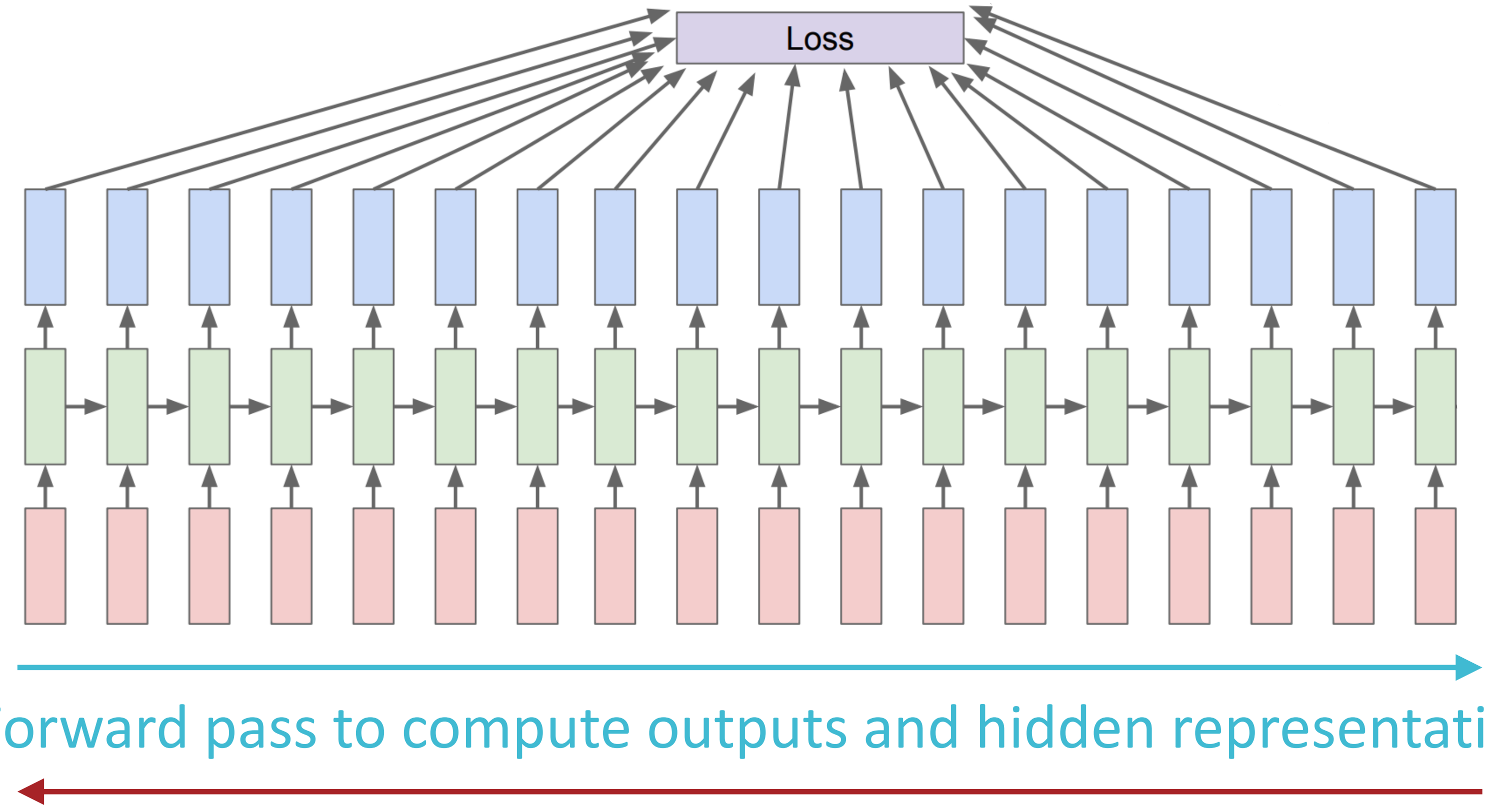
$$\mathcal{D} = \left\{\left(\boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)}\right)\right\}_{n=1}^{N}$$

$$\boldsymbol{x}^{(n)} = \left[\boldsymbol{x}_1^{(n)}, \ldots, \boldsymbol{x}_{T_n}^{(n)}\right]$$

$$\boldsymbol{y}^{(n)} = \left[\boldsymbol{y}_1^{(n)}, \ldots, \boldsymbol{y}_{T_n}^{(n)}\right]$$

- This model requires an initial value for the hidden representation, $\boldsymbol{h}_0$, typically a vector of all zeros

Training RNNs: Challenges



Forward pass to compute outputs and hidden representations

Backward pass to compute gradients

- Issue: as the sequence length grows, the gradient is more likely to explode or vanish

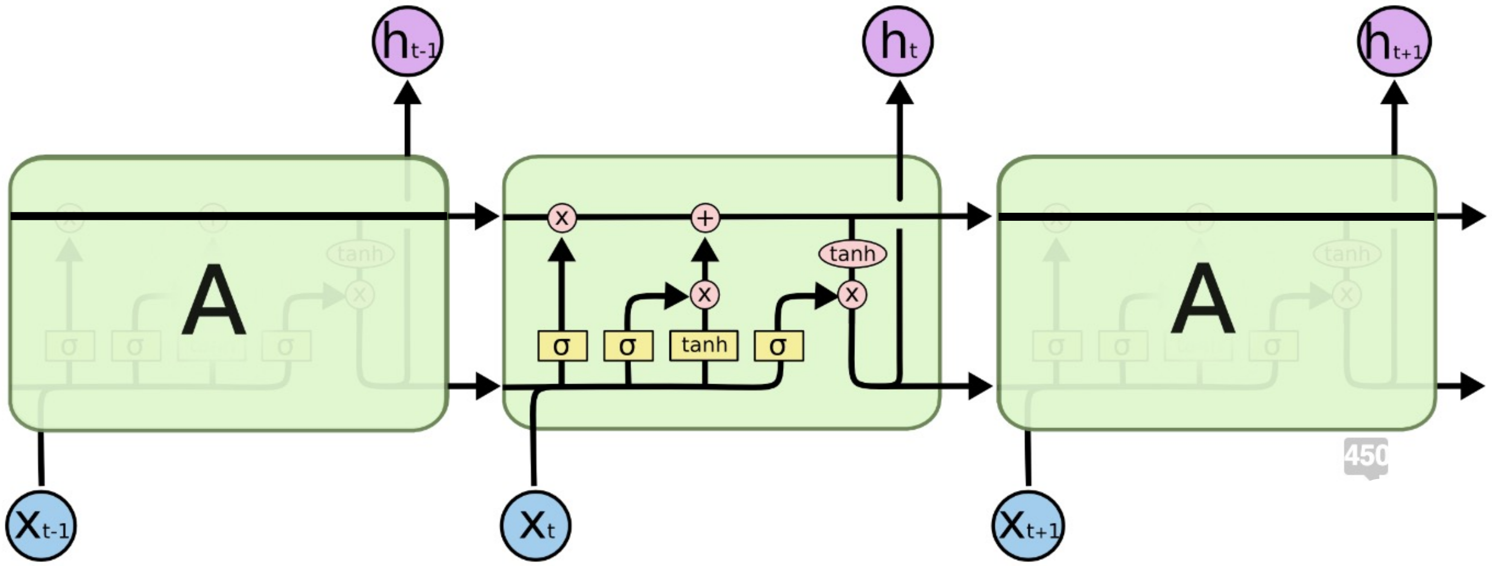
Source: http://cs231n.stanford.edu/slides/2023/lecture_8.pdf

Long Short-Term Memory (Hochreiter & Schmidhuber, 1997)

- LSTM networks address the vanishing gradient problem by replacing hidden layers with *memory cells*
- Each cell still computes a hidden representation but also maintains a separate internal *state,* C_t
- The flow of information through a cell is manipulated by three *gates*:
  - An input gate, I_t, that controls how much the state looks like the normal RNN hidden layer
  - An output gate, O_t, that "releases" the hidden representation to later timesteps
  - A forget gate, F_t, that determines if the previous memory cell's state affects the current internal state

Long Short-Term Memory (Hochreiter & Schmidhuber, 1997)

- LSTM networks address the vanishing gradient problem by replacing hidden layers with *memory cells*
- Each cell still computes a hidden representation but also maintains a separate internal *state*, C_t



- The internal state allows information to move through time without needing to affect the hidden representations!

Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Applications of LSTMs

2018: OpenAI used LSTM trained by policy gradients to beat humans in the complex video game of Dota 2,[11] and to control a human-like robot hand that manipulates physical objects with unprecedented dexterity.[10][54]

2019: DeepMind used LSTM trained by policy gradients to excel at the complex video game of Starcraft II.[12][54]

Source: https://en.wikipedia.org/wiki/Long_short-term_memory

Key Takeaways

- Recurrent neural networks use contextual information to reason about sequential data
  - Can still be learned using backpropagation → backpropagation through time
  - Susceptible to exploding/vanishing gradients for long training sequences
  - LSTMs allow contextual information to reach later timesteps without directly affecting intermediate hidden representations

Language Models

1. Convert raw text into *embeddings*

$$\boldsymbol{x}^{(i)} = \left[\boldsymbol{x}_1^{(i)}, \ldots, \boldsymbol{x}_{T_i}^{(i)}\right]$$

2. Learn or approximate a joint probability distribution over sequences

$$P\left(\boldsymbol{x}^{(i)}\right) = P\left(\boldsymbol{x}_1^{(i)}, \ldots, \boldsymbol{x}_{T_i}^{(i)}\right)$$

3. Sample from the implied conditional distribution to generate new sequences

$$P\left(\boldsymbol{x}_{T_i+1} \mid \boldsymbol{x}_1^{(i)}, \ldots, \boldsymbol{x}_{T_i}^{(i)}\right) = \frac{P\left(\boldsymbol{x}_1^{(i)}, \ldots, \boldsymbol{x}_{T_i}^{(i)}, \boldsymbol{x}_{T_i+1}\right)}{P\left(\boldsymbol{x}_1^{(i)}, \ldots, \boldsymbol{x}_{T_i}^{(i)}\right)}$$

Language Models

1. Convert raw text into *embeddings*

$$\boldsymbol{x}^{(i)} = \left[\boldsymbol{x}_1^{(i)}, \dots, \boldsymbol{x}_{T_i}^{(i)}\right]$$

2. Learn or approximate a joint probability distribution over sequences

$$P\left(\boldsymbol{x}^{(i)}\right) = P\left(\boldsymbol{x}_1^{(i)}, \dots, \boldsymbol{x}_{T_i}^{(i)}\right)$$

- Use the chain rule of probability: predict the next word based on the previous words in the sequence

$$\begin{aligned} P\left(\boldsymbol{x}^{(i)}\right) = P&\left(\boldsymbol{x}_1^{(i)}\right) \\ &* P\left(\boldsymbol{x}_2^{(i)} \mid \boldsymbol{x}_1^{(i)}\right) \\ &\vdots \\ &* P\left(\boldsymbol{x}_{T_i}^{(i)} \mid \boldsymbol{x}_{T_i-1}^{(i)}, \dots, \boldsymbol{x}_1^{(i)}\right) \end{aligned}$$

Language Models

1. Convert raw text into *embeddings*

$$\boldsymbol{x}^{(i)} = \left[\boldsymbol{x}_1^{(i)}, \dots, \boldsymbol{x}_{T_i}^{(i)}\right]$$

2. Learn or approximate a joint probability distribution over sequences

$$P\left(\boldsymbol{x}^{(i)}\right) = P\left(\boldsymbol{x}_1^{(i)}, \dots, \boldsymbol{x}_{T_i}^{(i)}\right)$$

- ~~Use the chain rule of probability~~ Just throw an RNN at it!

$$P\left(\boldsymbol{x}^{(i)}\right) = P\left(\boldsymbol{x}_1^{(i)}\right) * P\left(\boldsymbol{x}_2^{(i)} \mid \boldsymbol{x}_1^{(i)}\right) \vdots * P\left(\boldsymbol{x}_{T_i}^{(i)} \mid \boldsymbol{x}_{T_i-1}^{(i)}, \dots, \boldsymbol{x}_1^{(i)}\right)$$

RNN Language Models

1. Convert raw text into *embeddings*

$$\boldsymbol{x}^{(i)} = \left[\boldsymbol{x}_1^{(i)}, \ldots, \boldsymbol{x}_{T_i}^{(i)}\right]$$

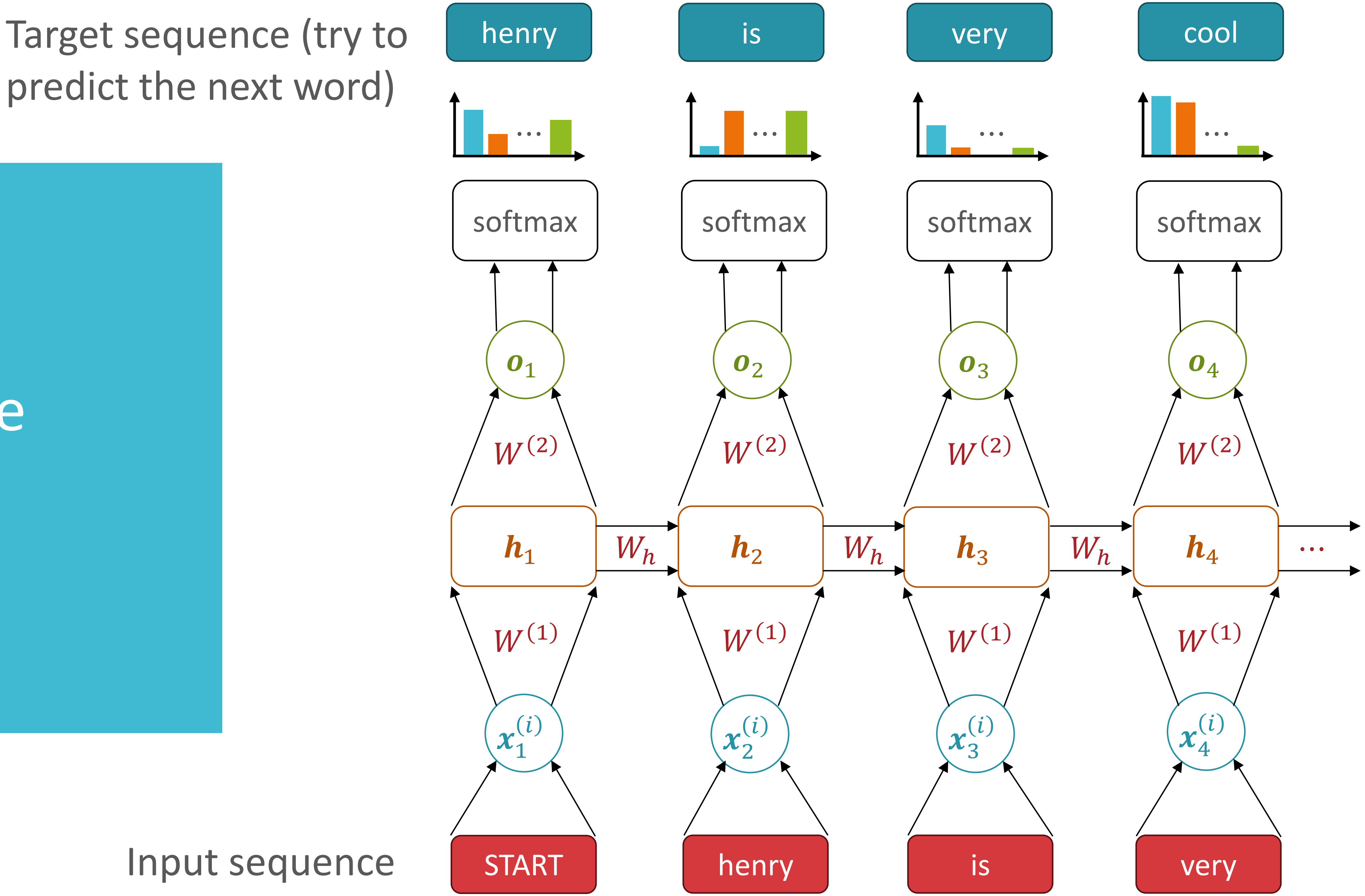
2. Learn or approximate a joint probability distribution over sequences

$$P\left(\boldsymbol{x}^{(i)}\right) = P\left(\boldsymbol{x}_1^{(i)}, \ldots, \boldsymbol{x}_{T_i}^{(i)}\right)$$

- ~~Use the chain rule of probability~~ Just throw an RNN at it!

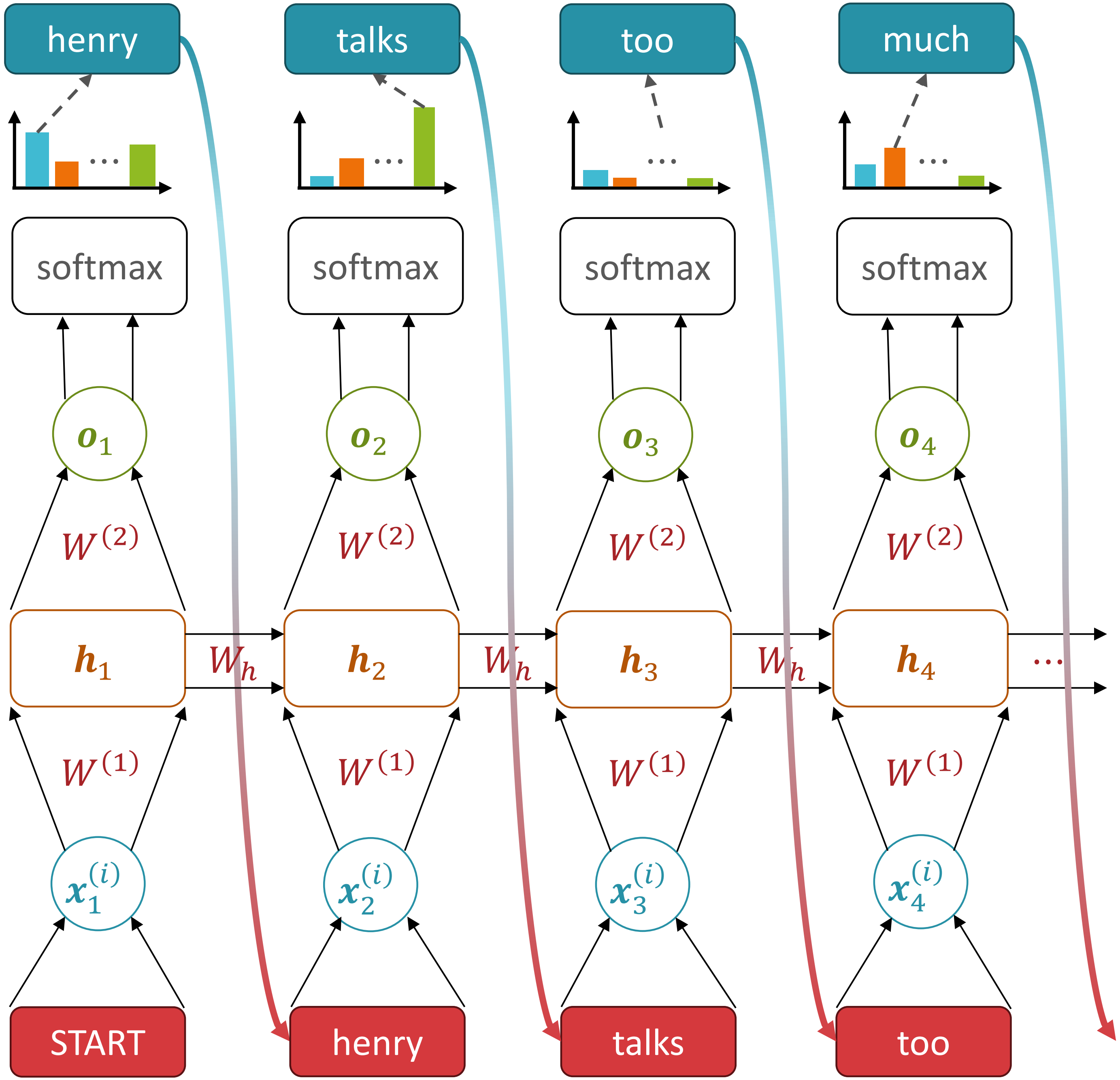
$$P\left(\boldsymbol{x}^{(i)}\right) \approx \boldsymbol{o}_1\left(\boldsymbol{x}_1^{(i)}\right) * \boldsymbol{o}_2\left(\boldsymbol{x}_2^{(i)}, \boldsymbol{h}_1\left(\boldsymbol{x}_1^{(i)}\right)\right) \vdots * \boldsymbol{o}_{T_i}\left(\boldsymbol{x}_{T_i}^{(i)}, \boldsymbol{h}_{T_i-1}\left(\boldsymbol{x}_{T_i-1}^{(i)}, \ldots, \boldsymbol{x}_1^{(i)}\right)\right)$$

RNN Language Models: Training



RNN Language Models: Sampling

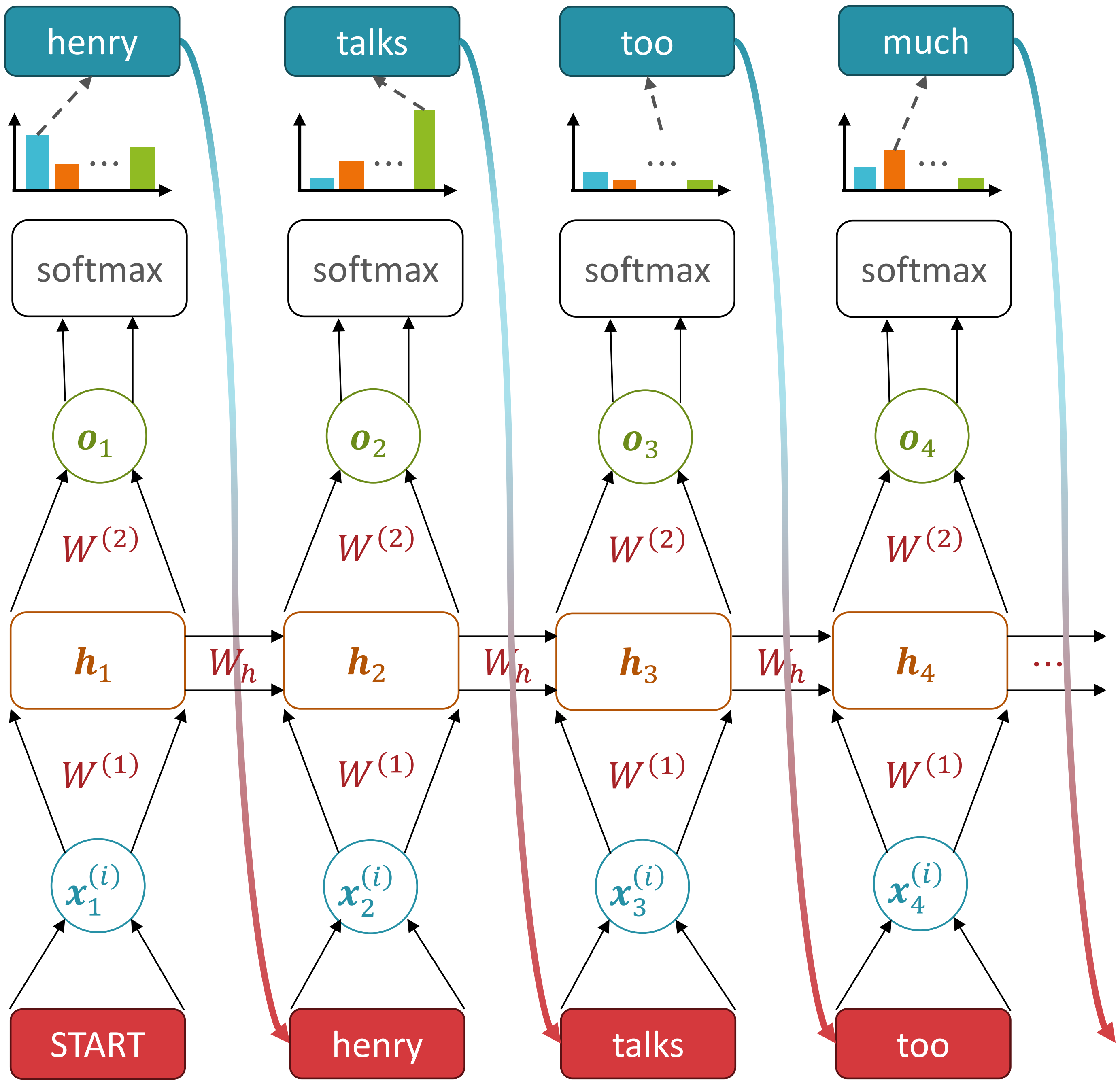
Generated sequence (use each token as the input to the next timestep)



Input sequence

Generated sequence (use each token as the input to the next timestep)

Aside: Sampling from these distributions to get the next word is not always the best thing to do



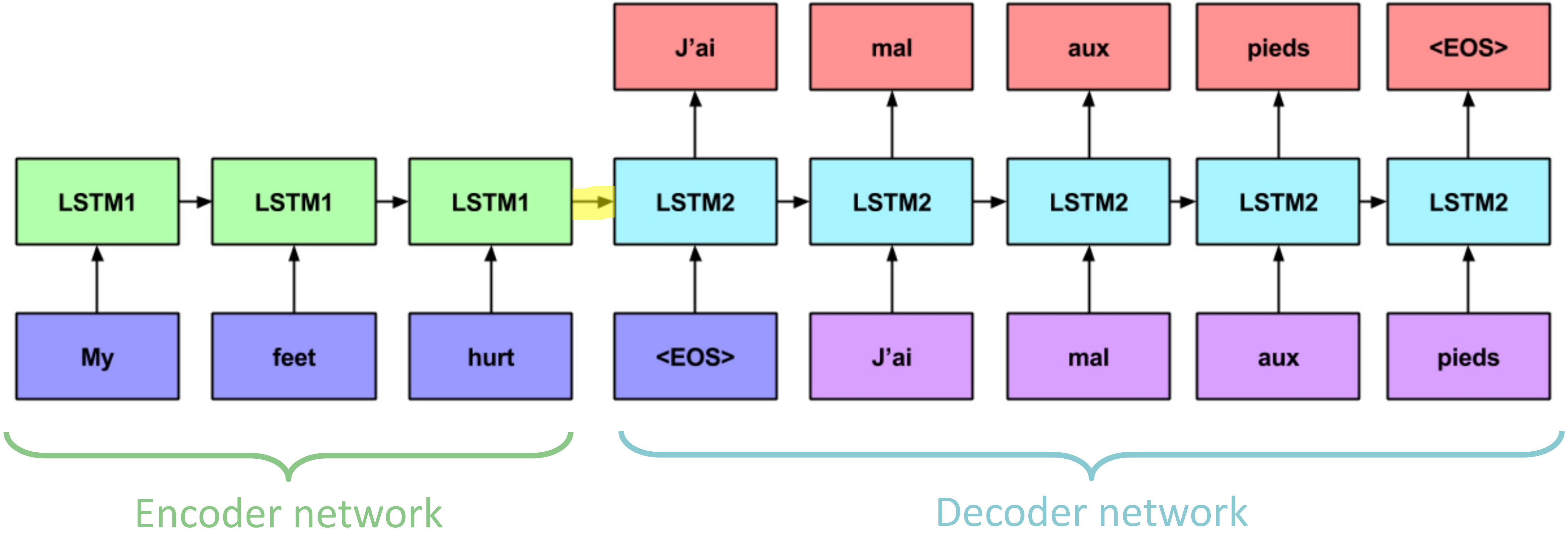
Input sequence

RNN Language Models: Pros & Cons

- Pros:
  - Can handle arbitrary sequence lengths without having to increase model size (i.e., # of learnable parameters)
  - Trainable via backpropagation
- Cons
  - Vanishing/exploding gradients
    - Can be addressed by LSTMs
  - Does not consider information from later timesteps
    - Can be addressed by bidirectional RNNs
  - Computation is inherently sequential
  - "You can't cram the meaning of a whole %&!$# sentence into a single $&!#* vector!" – Ray Mooney, UT Austin

RNN Language Models: Pros & Cons

- Pros:
    - Can handle arbitrary sequence lengths without having to increase model size (i.e., # of learnable parameters)
    - Trainable via backpropagation
- Cons
    - Vanishing/exploding gradients
        - Can be addressed by LSTMs
    - Does not consider information from later timesteps
        - Can be addressed by bidirectional RNNs
    - Computation is inherently sequential
    - The entire sequence up to some timestep is represented using just one vector (or two vectors in an LSTM)



Encoder-Decoder Architectures (Sutskever et al., 2014)

Source: https://arxiv.org/pdf/1506.00019.pdf

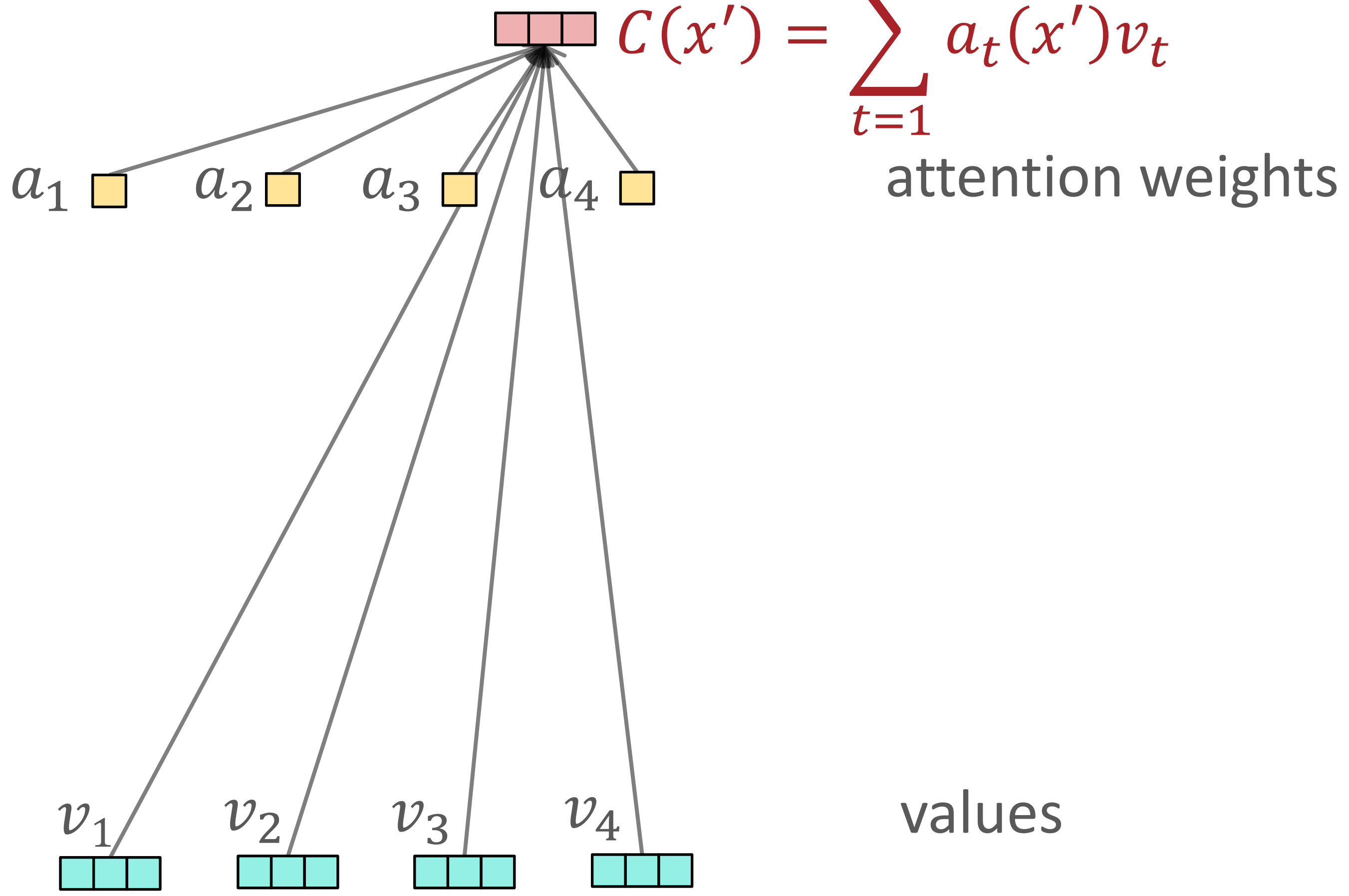
Attention

- Approach: compute a representation of the input sequence for each token x' in the decoder
- Idea: allow the decoder to learn which tokens in the input to "pay attention to" i.e., put more weight on

Attention

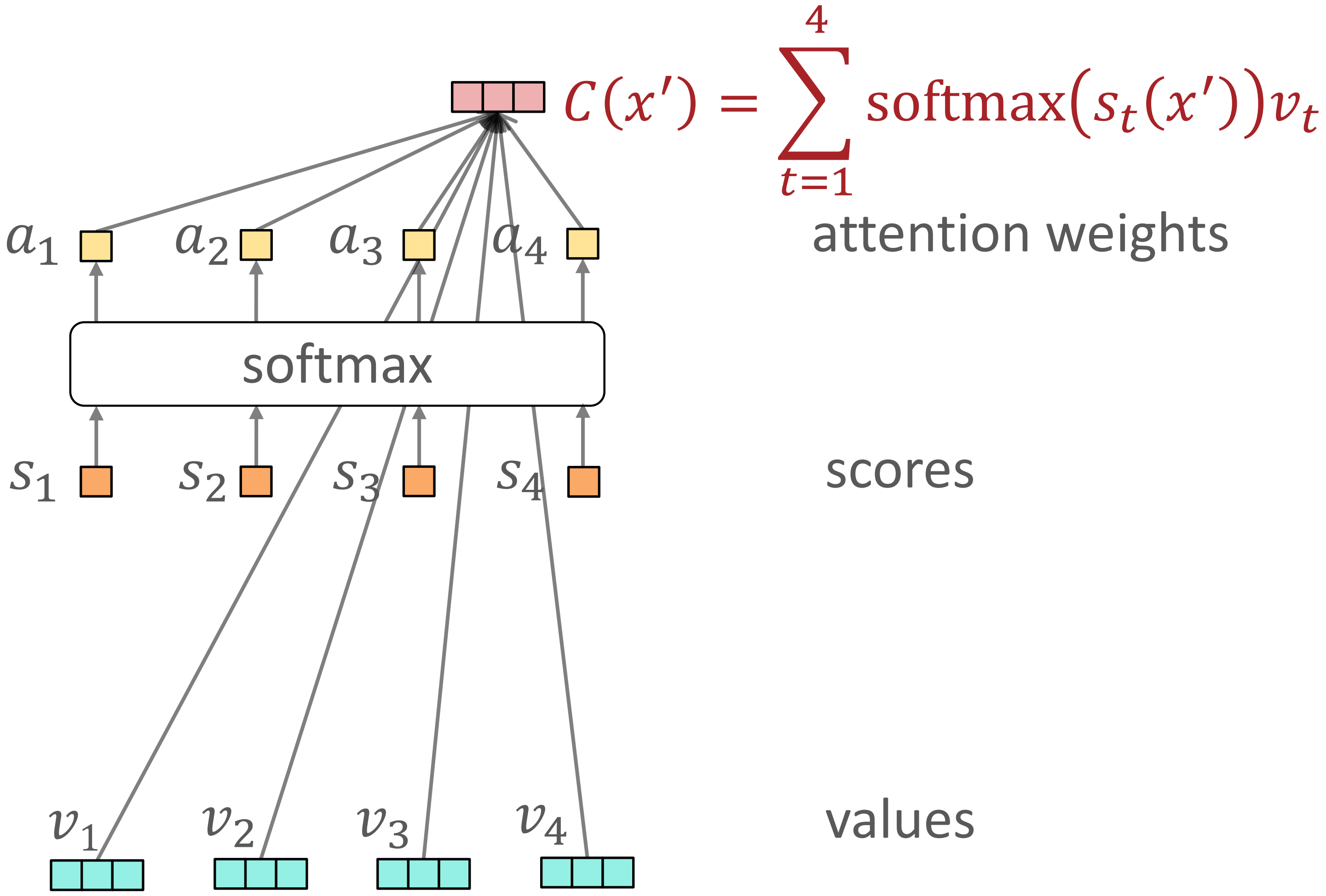
- Approach: compute a representation of the input sequence for each token x' in the decoder

$$C(x') = \sum_{t=1}^{4} a_t(x') v_t$$



Attention

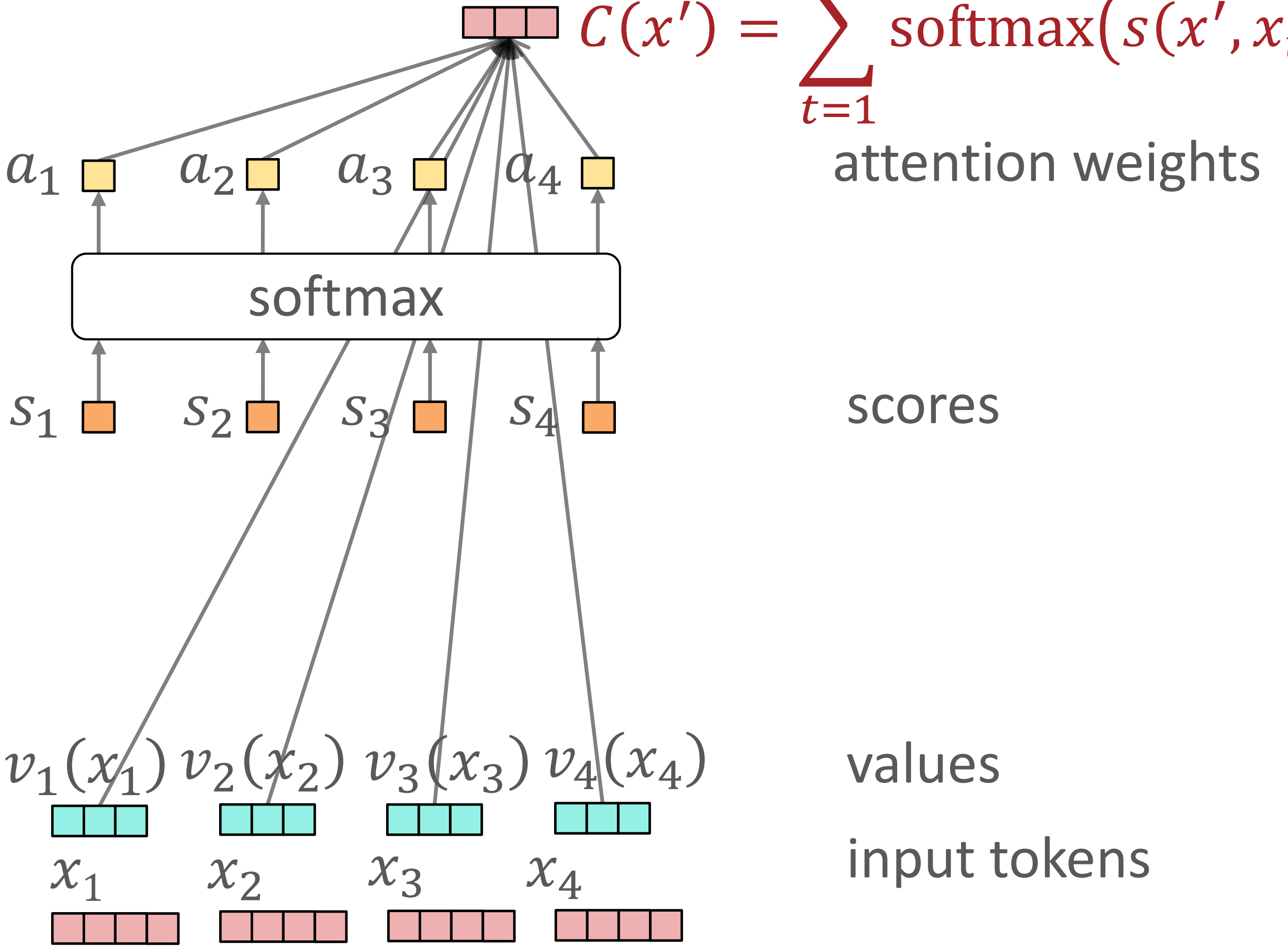
- Approach: compute a representation of the input sequence for each token x' in the decoder



Attention

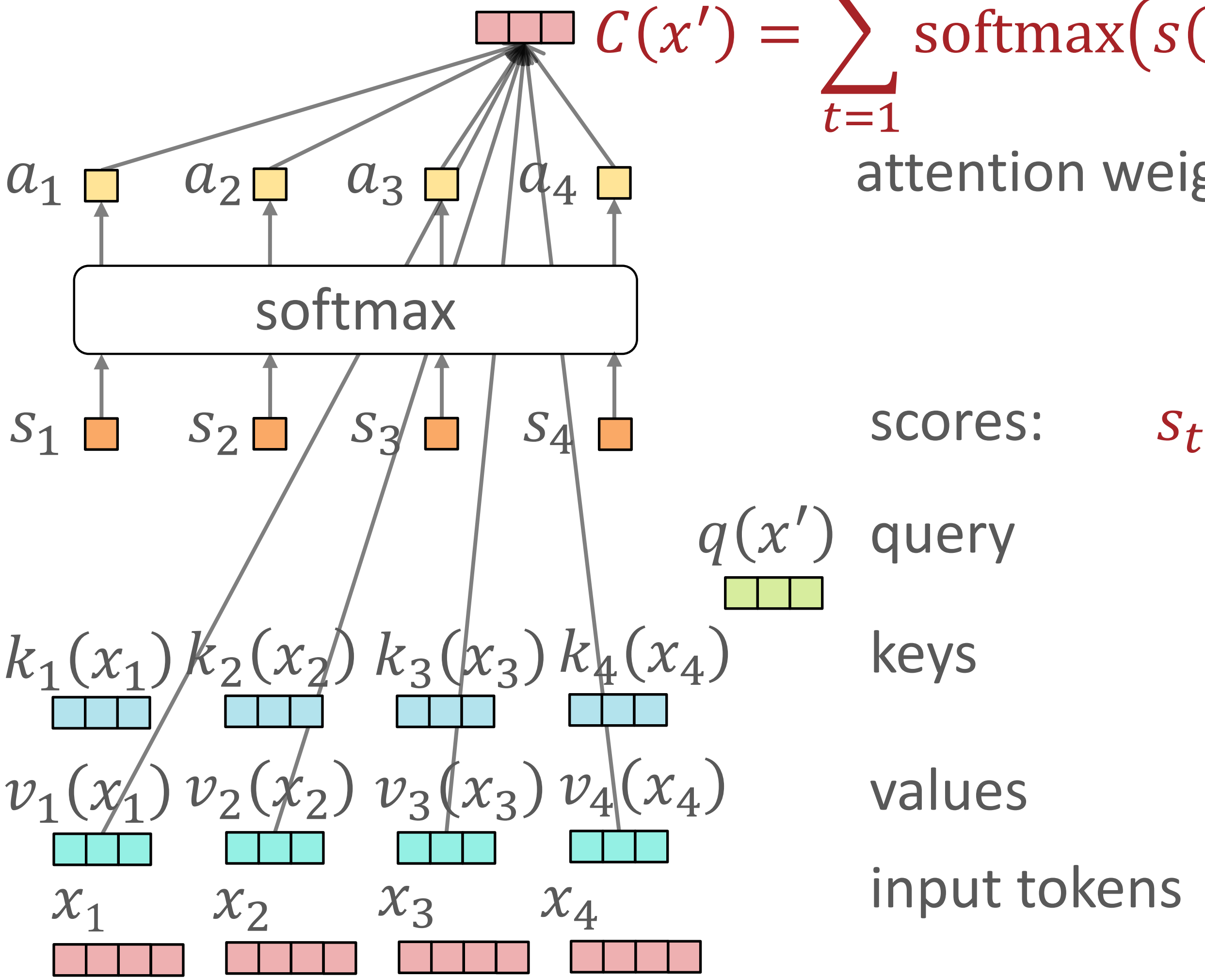
- Approach: compute a representation of the input sequence for each token x' in the decoder

$$C(x') = \sum_{t=1}^{4} \text{softmax}\big(s(x', x_t)\big) v(x_t)$$



Scaled Dot-product Attention

- Approach: compute a representation of the input sequence for each token x' in the decoder



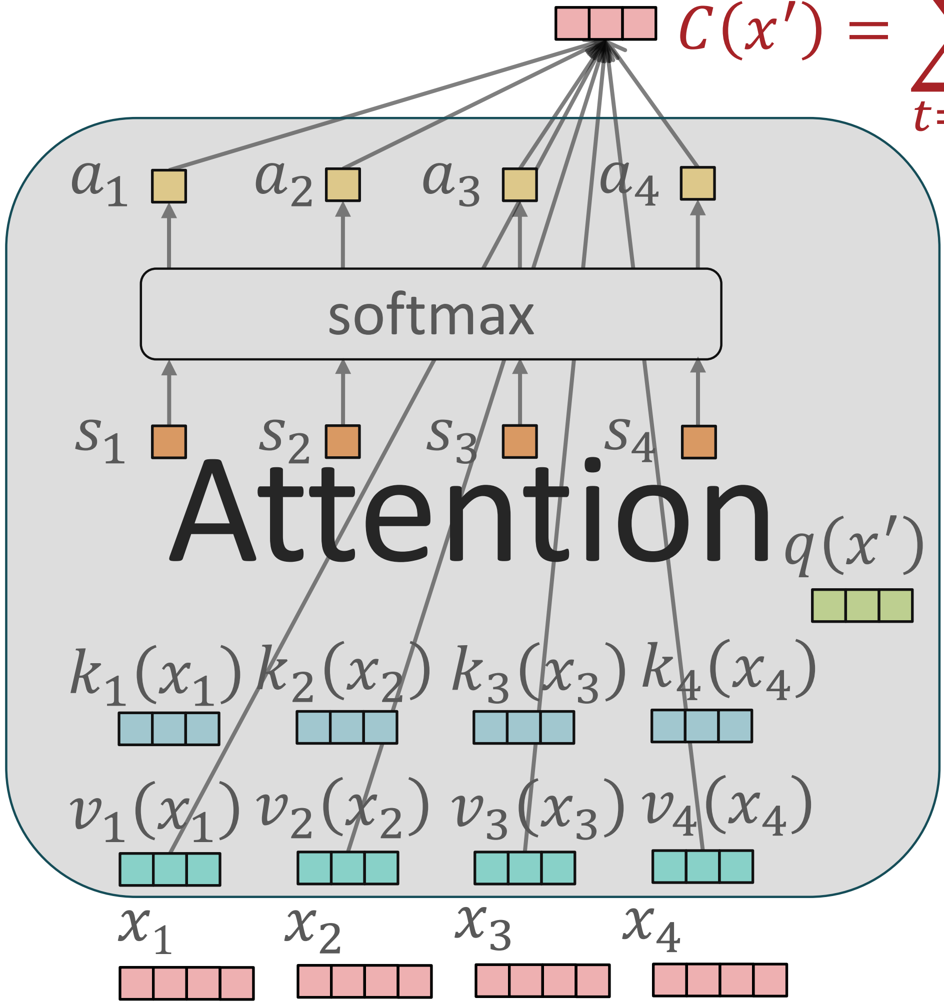
$$C(x') = \sum_{t=1}^{4} \text{softmax}\big(s(x', x_t)\big) v(x_t)$$

$$s_t = \frac{k_t^T q}{\sqrt{\text{length}(q)}}$$

Scaled Dot-product Attention

- Approach: compute a representation of the input sequence for each token x' in the decoder

$$C(x') = \sum_{t=1}^{4} \text{softmax}\big(s(x', x_t)\big) v(x_t)$$



attention weights

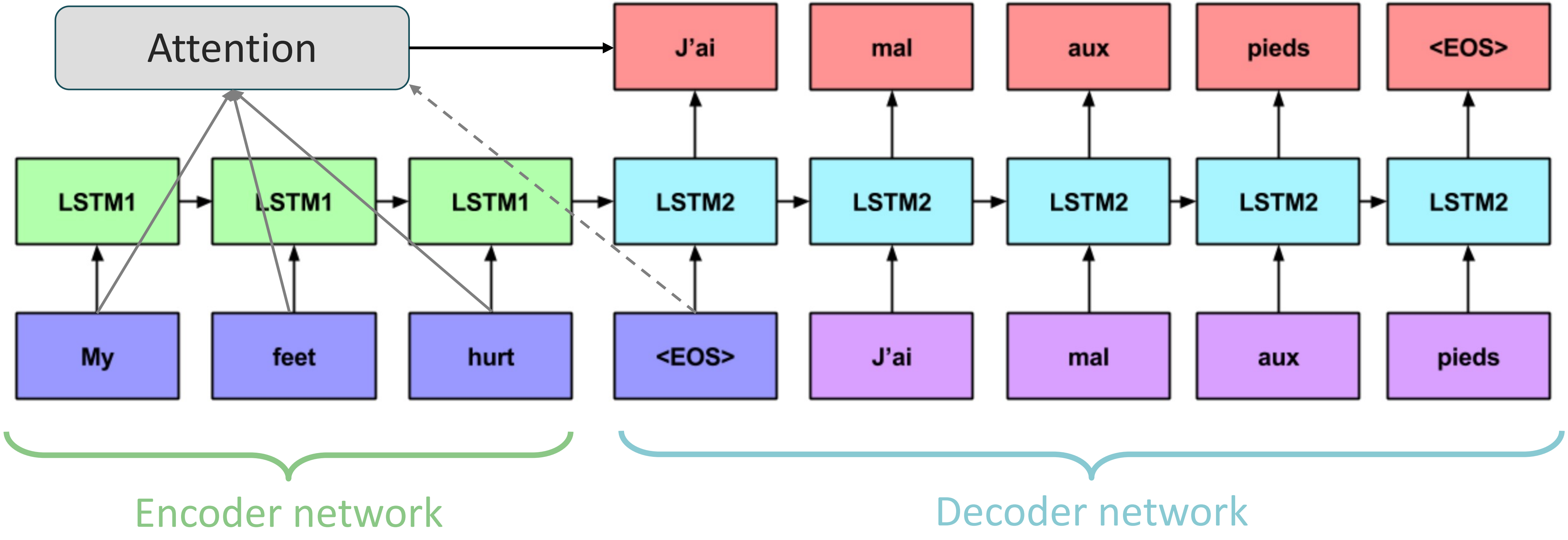
scores: $s_t = \frac{k_t^T q}{\sqrt{\text{length}(q)}}$

query: $q = w_Q^T x'$

keys: $k_t = W_K x_t$

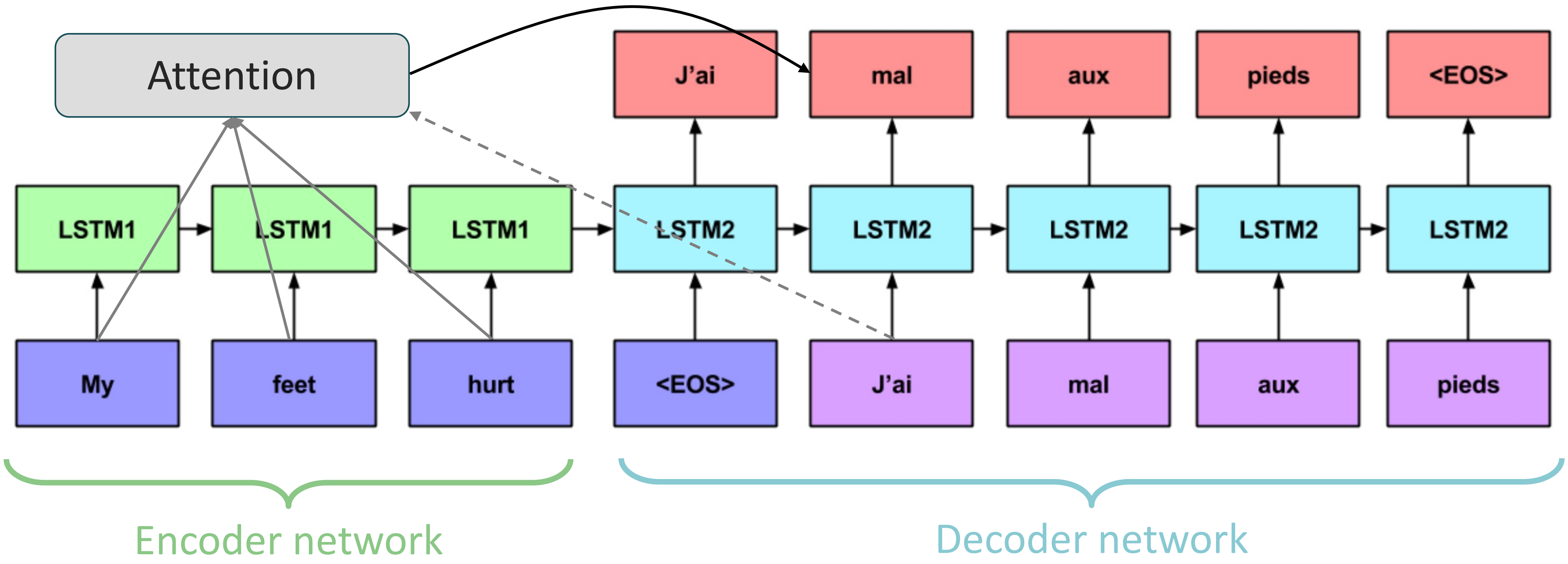
values: $v_t = W_V x_t$

input tokens



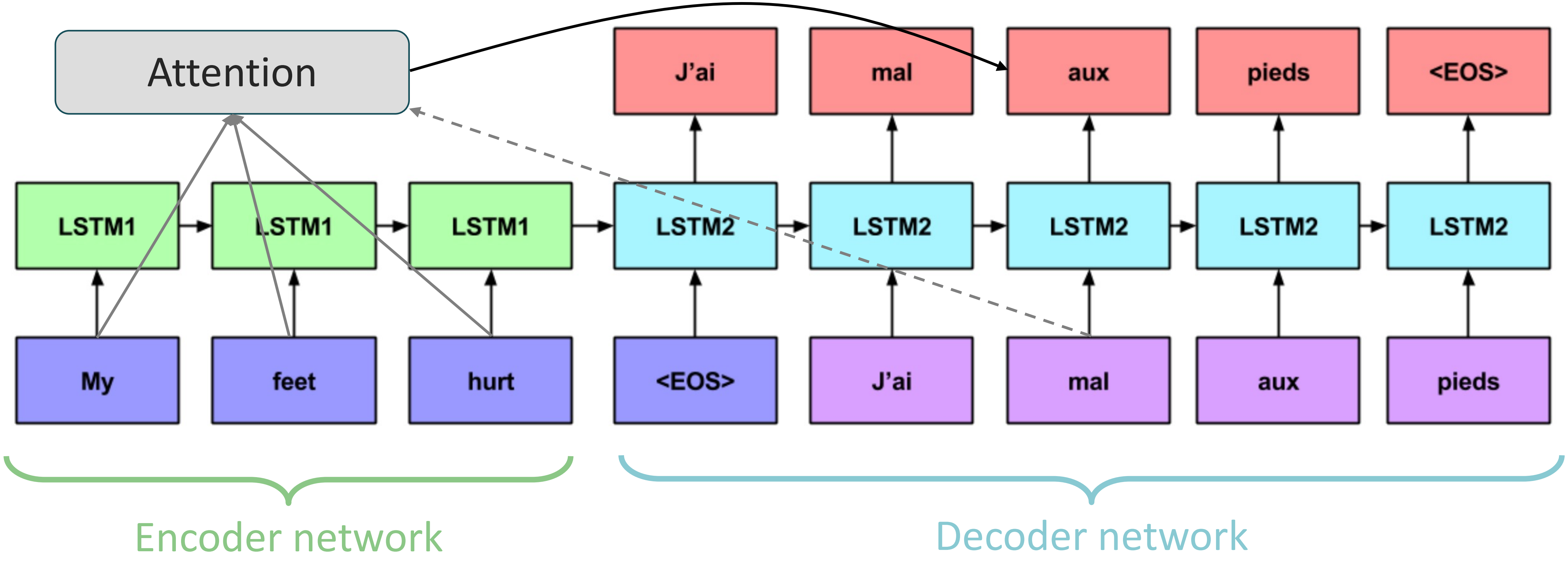
Encoder-Decoder Architectures with Attention

Source: https://arxiv.org/pdf/1506.00019.pdf



Encoder-Decoder Architectures with Attention

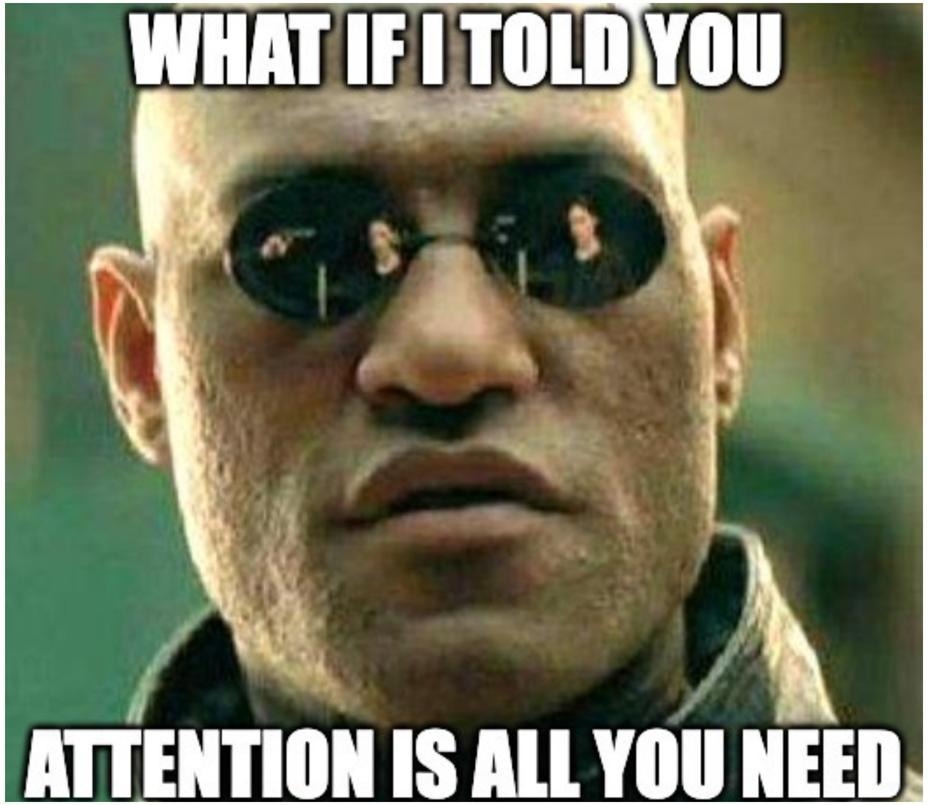
Source: https://arxiv.org/pdf/1506.00019.pdf



Encoder-Decoder Architectures with Attention

Source: https://arxiv.org/pdf/1506.00019.pdf

Attention

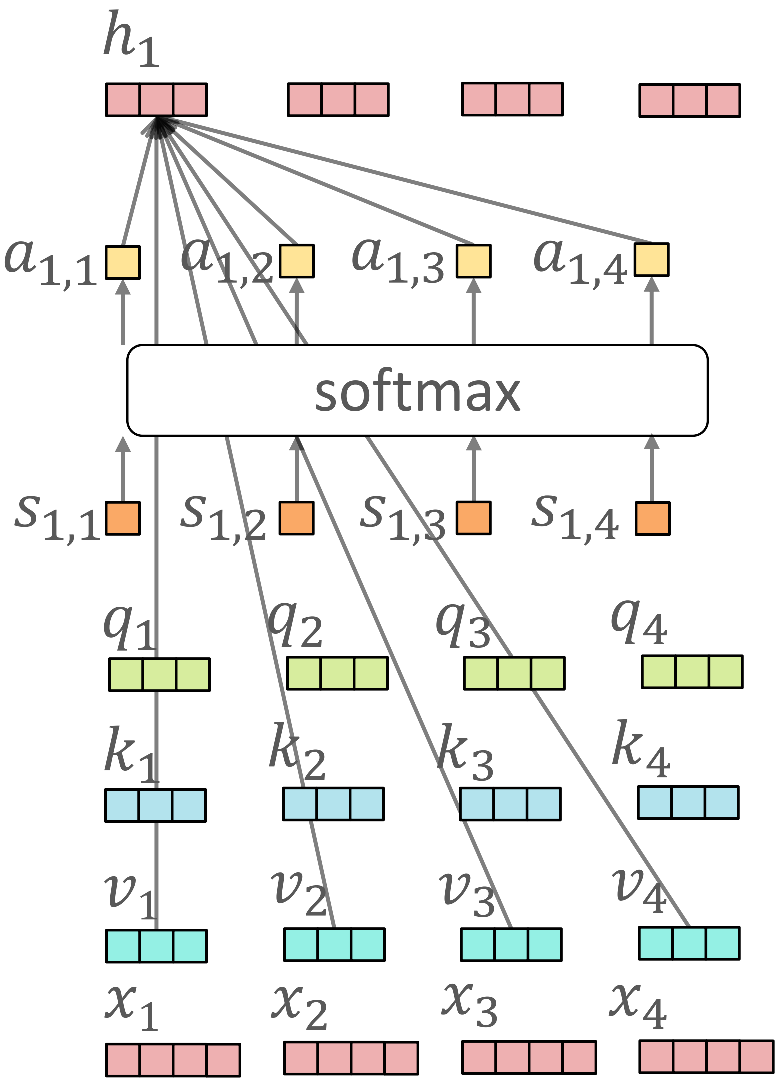


~~Encoder-Decoder Architectures with~~ Attention (Vaswani et al., 2017)

Source: https://arxiv.org/pdf/1706.03762.pdf

Scaled Dot-product Self-attention

- Approach: compute a representation for each token in the *input sequence* by attending to all the input tokens



$$h_1 = \sum_{j=1}^{4} \text{softmax}(s_{1,j}) v_j$$

attention weights

scores: $s_{1,j} = \frac{k_j^T q_1}{\sqrt{\text{length}(k_j)}}$

queries: $q_t = W_Q x_t$

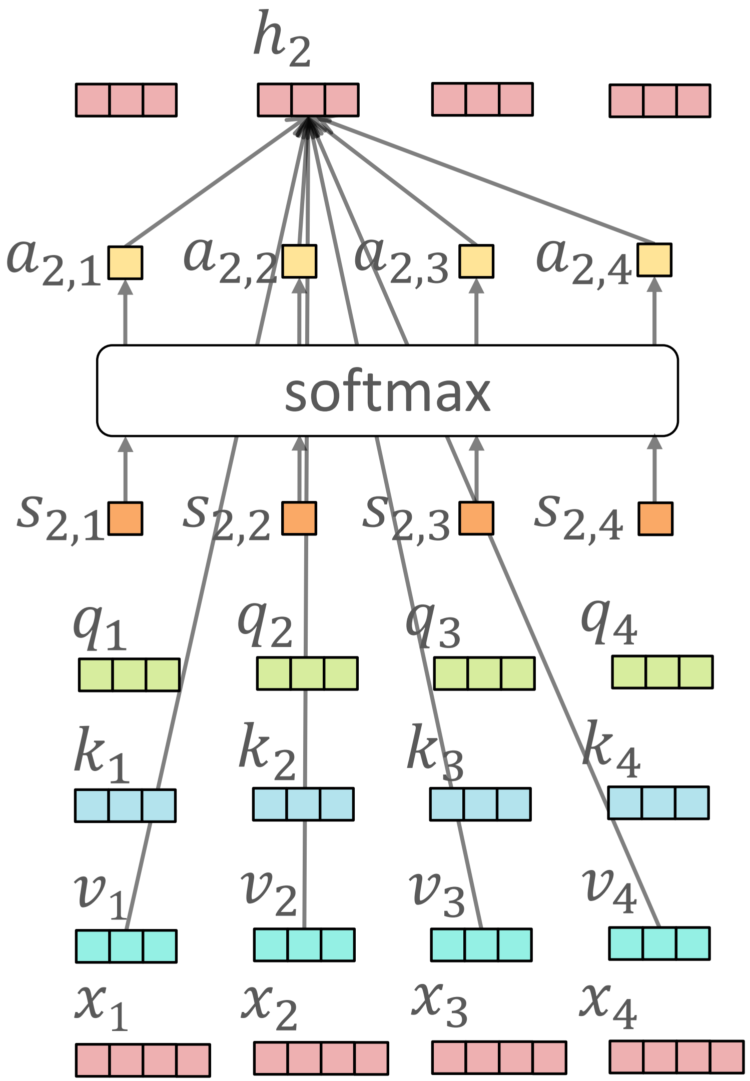
keys: $k_t = W_K x_t$

values: $v_t = W_V x_t$

input tokens

Scaled Dot-product Self-attention

- Approach: compute a representation for each token in the *input sequence* by attending to all the input tokens



$$h_2 = \sum_{j=1}^{4} \text{softmax}(s_{2,j}) v_j$$

attention weights

scores: $s_{2,j} = \frac{k_j^T q_2}{\sqrt{\text{length}(k_j)}}$

queries: $q_t = W_Q x_t$

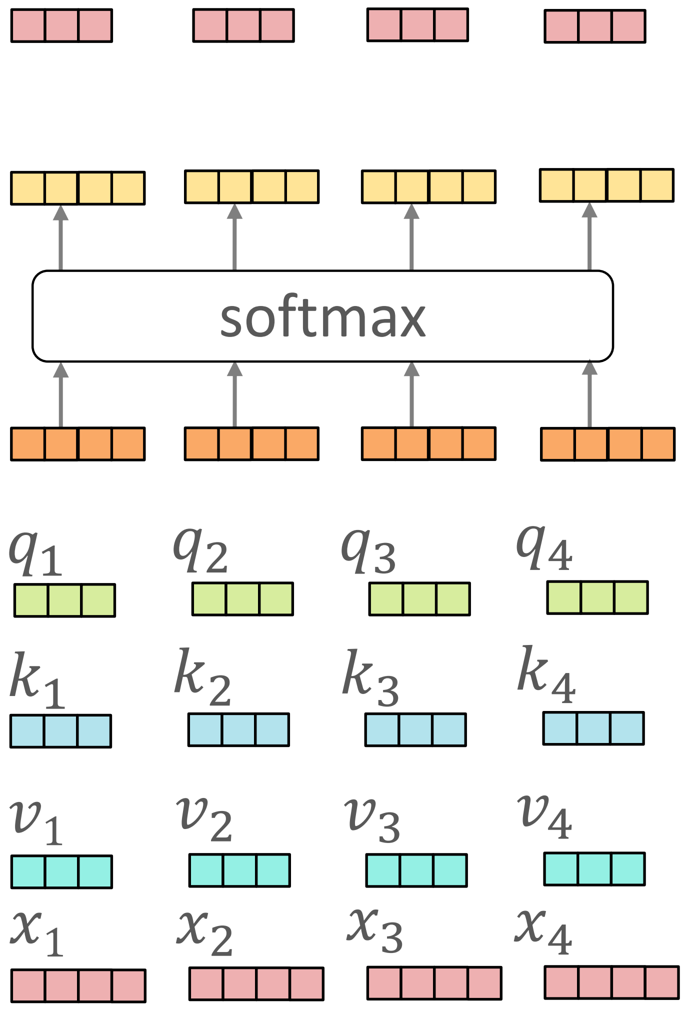
keys: $k_t = W_K x_t$

values: $v_t = W_V x_t$

input tokens

Scaled Dot-product Self-attention: Matrix Form

- Approach: compute a representation for each token in the *input sequence* by attending to all the input tokens



$H = \text{softmax}(S)V \in \mathbb{R}^{N \times d_v}$

attention weights

scores: $S = \frac{QK^T}{\sqrt{d_k}} \in \mathbb{R}^{N \times N}$

queries: $Q = XW_Q \in \mathbb{R}^{N \times d_k}$

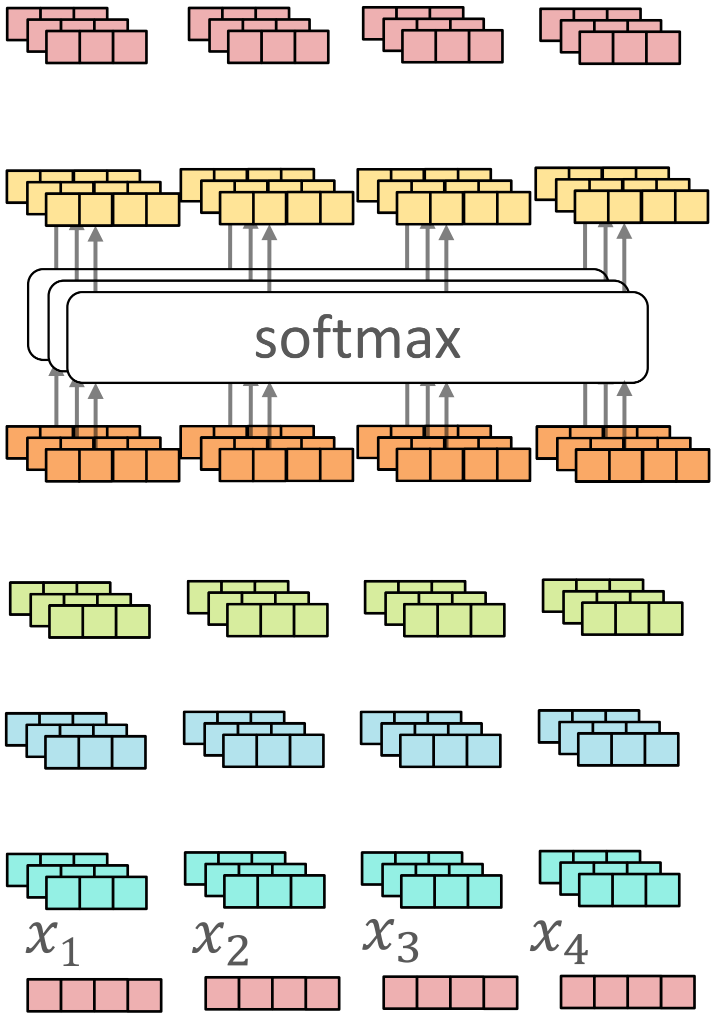
keys: $K = XW_K \in \mathbb{R}^{N \times d_k}$

values: $V = XW_V \in \mathbb{R}^{N \times d_v}$

design matrix: $X \in \mathbb{R}^{N \times D}$

Multi-head Scaled Dot-product Self-attention

- Idea: just like we might want multiple convolutional filters in a convolutional layer, we might want multiple attention weights to learn different relationships between tokens!



$H^{(h)} = \text{softmax}\left(S^{(h)}\right)V^{(h)}$

attention weights

scores: $S^{(h)} = \frac{Q^{(h)}{K^{(h)}}^T}{\sqrt{d_k^{(h)}}}$

queries: $Q^{(h)} = XW_Q^{(h)}$

keys: $K^{(h)} = XW_K^{(h)}$

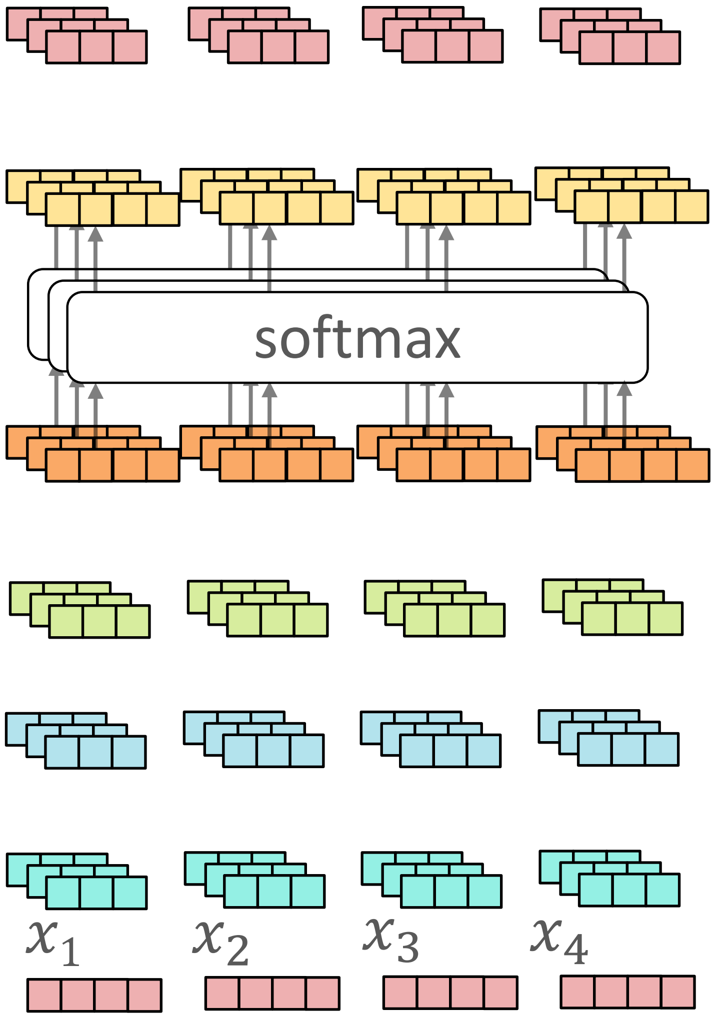
values: $V^{(h)} = XW_V^{(h)}$

design matrix: X

Key Takeaway: All of this computation is

1. differentiable
2. highly parallelizable!

- Idea: just like we might want multiple convolutional filters in a convolutional layer, we might want multiple attention weights to learn different relationships between tokens!



$$H^{(h)} = \text{softmax}\left(S^{(h)}\right)V^{(h)}$$

attention weights

scores: $S^{(h)} = \frac{Q^{(h)} {K^{(h)}}^T}{\sqrt{d_k^{(h)}}}$

queries: $Q^{(h)} = XW_Q^{(h)}$

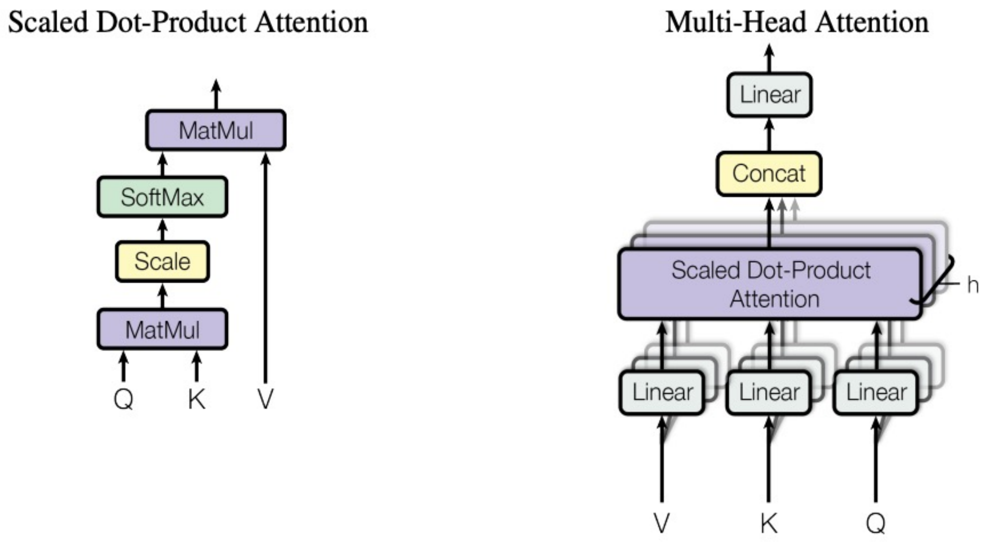
keys: $K^{(h)} = XW_K^{(h)}$

values: $V^{(h)} = XW_V^{(h)}$

design matrix: X

Multi-head Scaled Dot-product Self-attention

- Idea: just like we might want multiple convolutional filters in a convolutional layer, we might want multiple attention weights to learn different relationships between tokens!



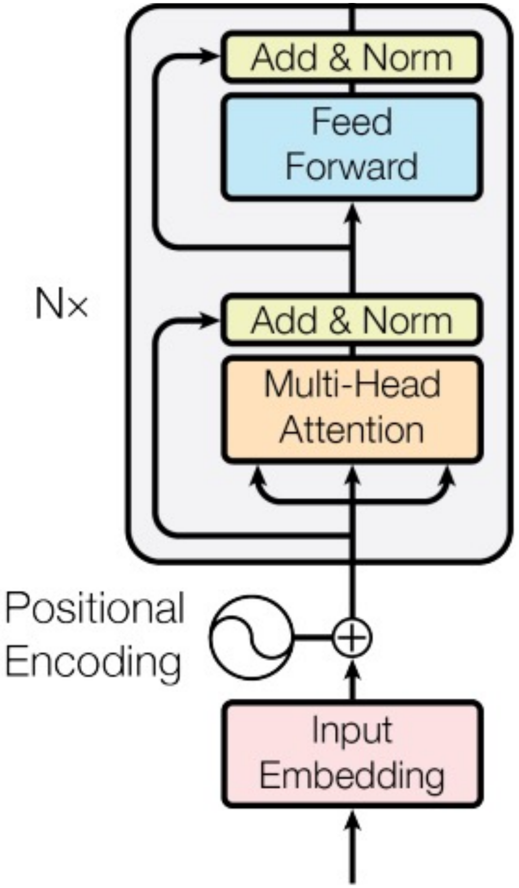
- The outputs from all the attention heads are concatenated together to get the final representation

$$H = \left[H^{(1)}, H^{(2)}, \dots, H^{(h)}\right]$$

- Common architectural choice: $d_v = {}^{D}/_{h} \rightarrow |H| = D$

Source: https://arxiv.org/pdf/1706.03762.pdf

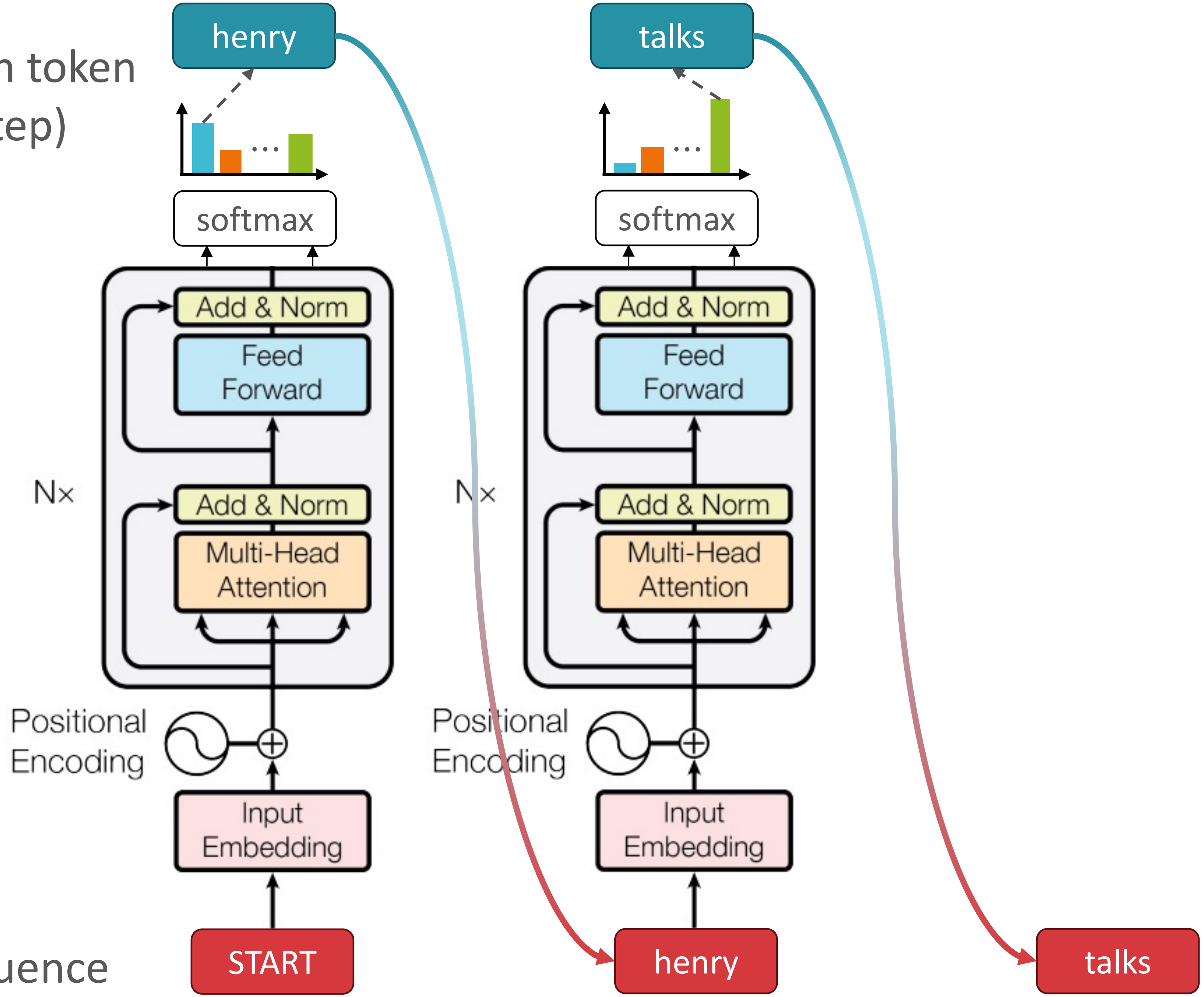
Transformers



Source: https://arxiv.org/pdf/1706.03762.pdf

Transformer Language Models

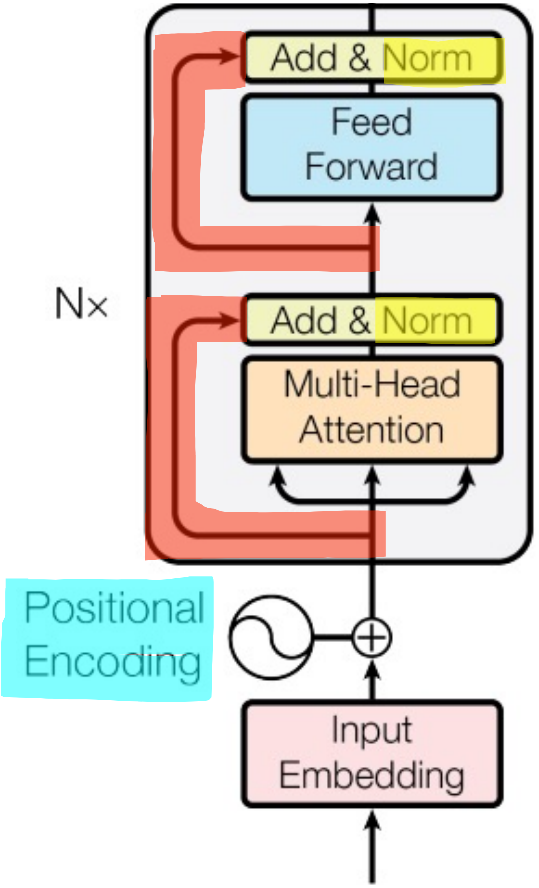
Generated sequence (use each token as the input to the next timestep)



Input sequence

Source: https://arxiv.org/pdf/1706.03762.pdf

Transformers

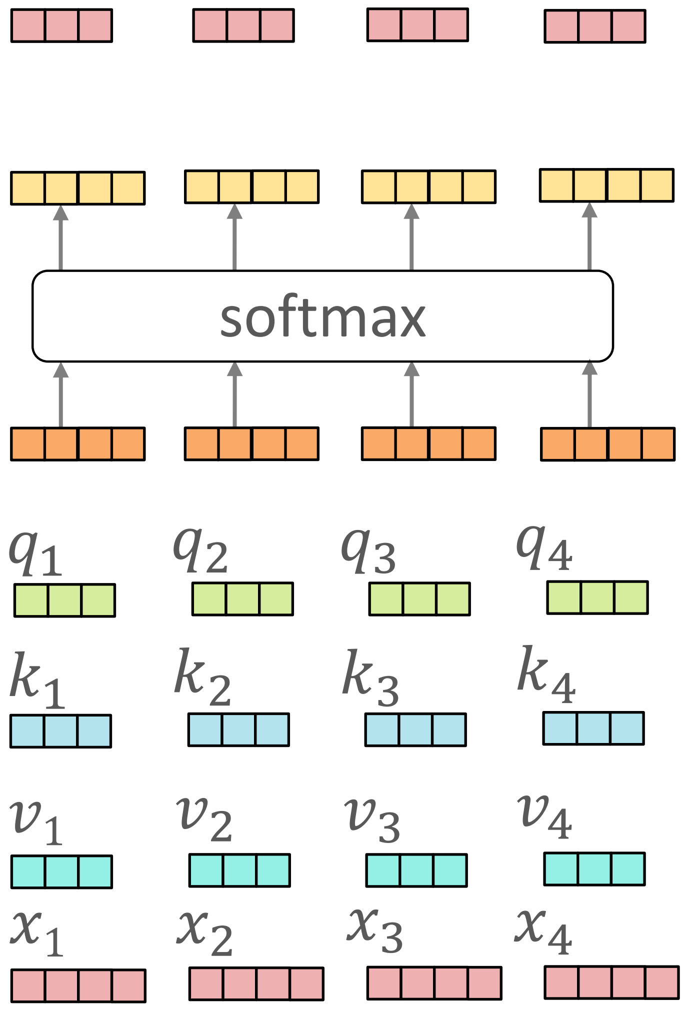


- In addition to multi-head attention, transformer architectures use
  1. Positional encodings
  2. Layer normalization
  3. Residual connections
  4. A fully-connected feed-forward network

Source: https://arxiv.org/pdf/1706.03762.pdf

Scaled Dot-product Self-attention: Matrix Form

- Issue: if all tokens attend to every token in the sequence, then how does the model infer the order of tokens?



$H = \text{softmax}(S)V \in \mathbb{R}^{N \times d_v}$

attention weights

scores: $S = \frac{QK^T}{\sqrt{d_k}} \in \mathbb{R}^{N \times N}$

queries: $Q = XW_Q \in \mathbb{R}^{N \times d_k}$

keys: $K = XW_K \in \mathbb{R}^{N \times d_k}$

values: $V = XW_V \in \mathbb{R}^{N \times d_v}$

design matrix: $X \in \mathbb{R}^{N \times D}$

Positional Encodings

- Issue: if all tokens attend to every token in the sequence, then how does the model infer the order of tokens?
- Idea: add a position-specific embedding p_t to the token embedding x_t

$$x_t' = x_t + p_t$$

- Positional encodings can be
  - *fixed* i.e., some predetermined function of t or *learned* alongside the token embeddings
  - *absolute* i.e., only dependent on the token's location in the sequence or *relative* to the query token's location

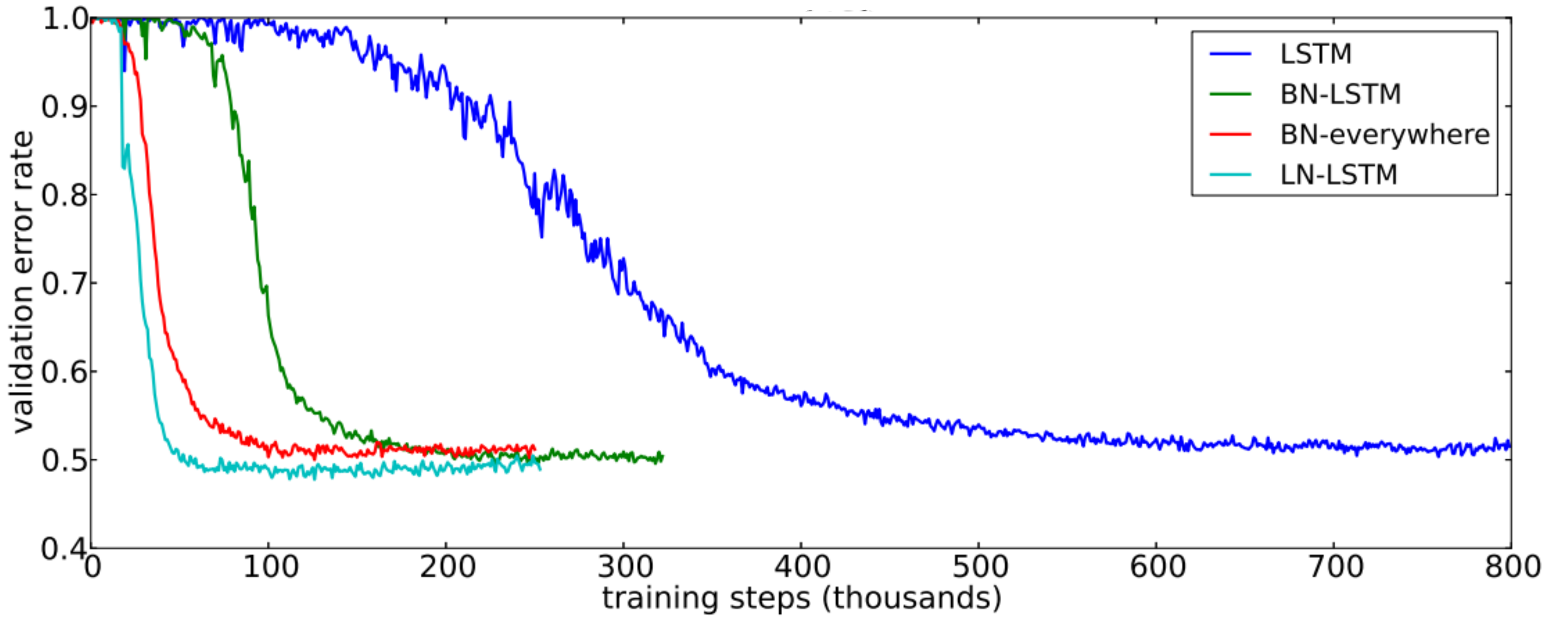
Layer Normalization

- Issue: for certain activation functions, the weights in later layers are **highly sensitive** to changes in the earlier layers
  - Small changes to weights in early layers are amplified so weights in deeper layers have to deal with massive dynamic ranges → slow optimization convergence
- Idea: normalize the output of a layer to always have the same (learnable) mean, β, and variance, γ^2

$$H' = \gamma\left(\frac{H - \mu}{\sigma}\right) + \beta$$

where μ is the mean and σ is the standard deviation of the values in the vector H

Layer Normalization



- Idea: normalize the output of a layer to always have the same (learnable) mean, β, and variance, γ^2

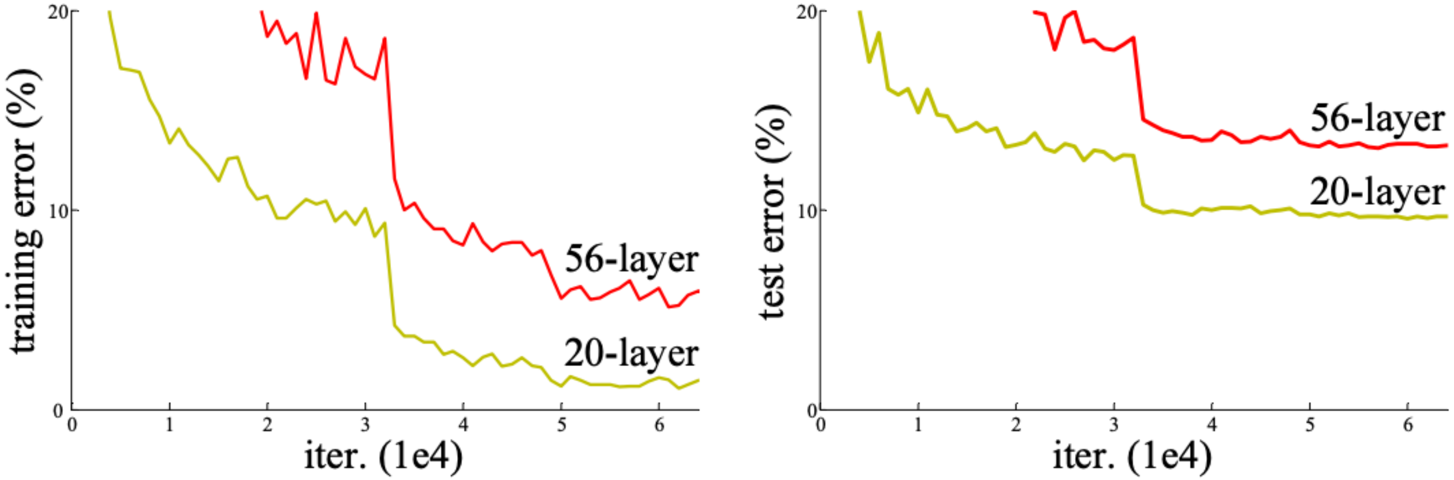
$$H' = \gamma\left(\frac{H-\mu}{\sigma}\right) + \beta$$

where μ is the mean and σ is the standard deviation of the values in the vector H

Source: https://arxiv.org/pdf/1607.06450.pdf

Residual Connections

- Observation: early deep neural networks suffered from the "degradation" problem where adding more layers actually made performance worse!



- Wait but this is ridiculous: if the later layers aren't helping, couldn't they just learn the identity transformation???
- Insight: neural network layers actually have a hard time learning the identity function

Source: https://arxiv.org/pdf/1512.03385.pdf

Residual Connections

- Observation: early deep neural networks suffered from the “degradation” problem where adding more layers actually made performance worse!
- Idea: add the input embedding back to the output of a layer

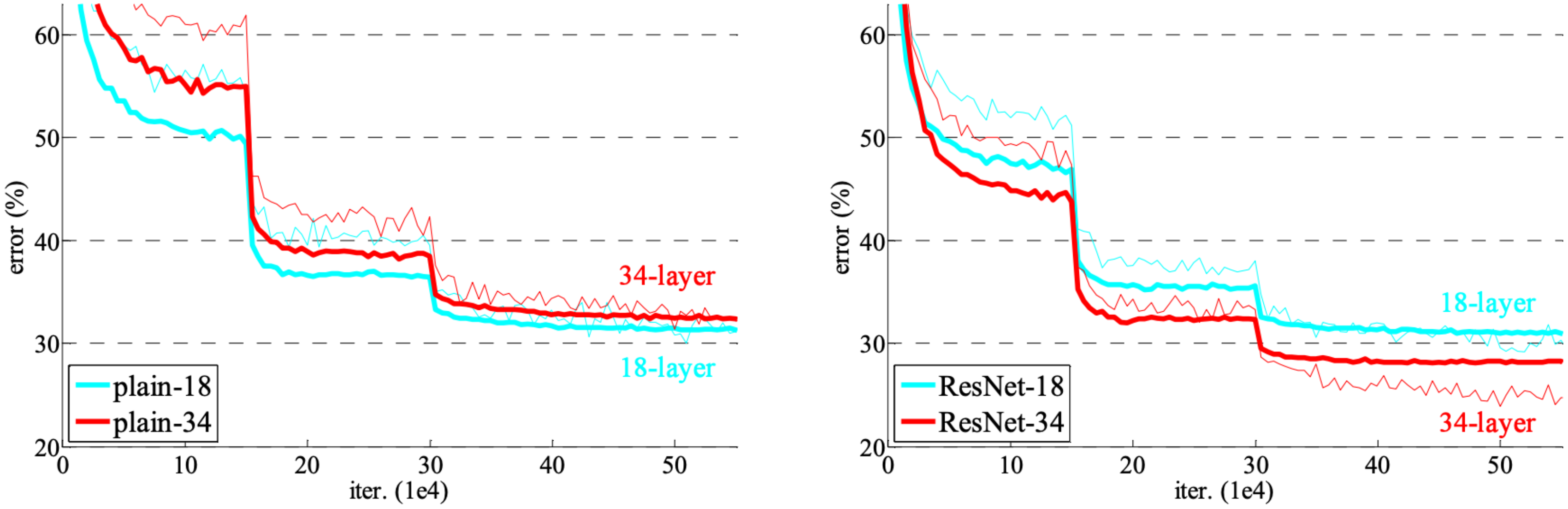
$$H' = H\left(x^{(i)}\right) + x^{(i)}$$

- Suppose the target function is f
  - Now instead of having to learn $f\left(x^{(i)}\right)$, the hidden layer just needs to learn the residual $r = f\left(x^{(i)}\right) - x^{(i)}$
  - If f is the identity function, then the hidden layer just needs to learn $r = 0$, which is easy for a neural network!

Residual Connections

- Observation: early deep neural networks suffered from the "degradation" problem where adding more layers actually made performance worse!
- Idea: add the input embedding back to the output of a layer

$$H' = H\left(x^{(i)}\right) + x^{(i)}$$



Source: https://arxiv.org/pdf/1512.03385.pdf

Key Takeaways

- Language models fit joint probability distributions to sequences of inputs
  - Can be sampled from to generate text
- Attention allows information to directly pass between every pair of tokens
  - Attention can be used in conjunction with RNNs/LSTMs
  - However, (self-)attention can also be used in isolation
- Transformers consist of multi-head attention layers with residual connections, layer normalization and fully-connected layers