10-301/601: Introduction to Machine Learning Lecture 22 – Attention & Transformers

Henry Chai

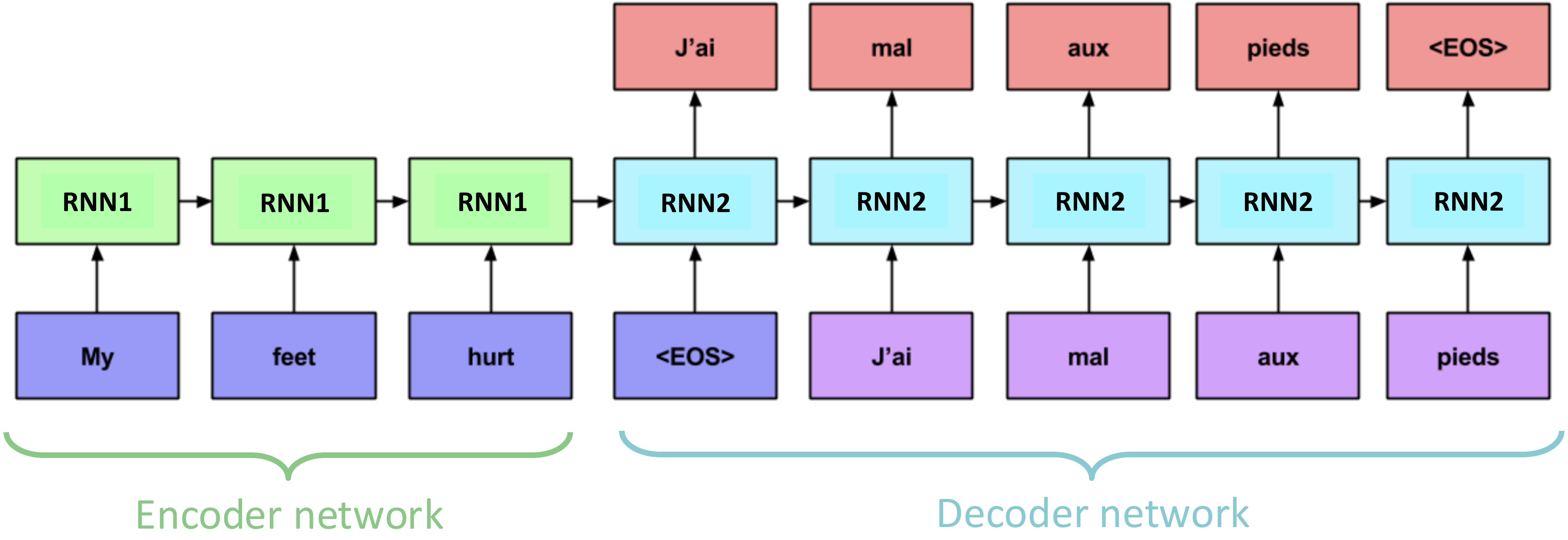
6/3/25

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
  - 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
  - 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



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

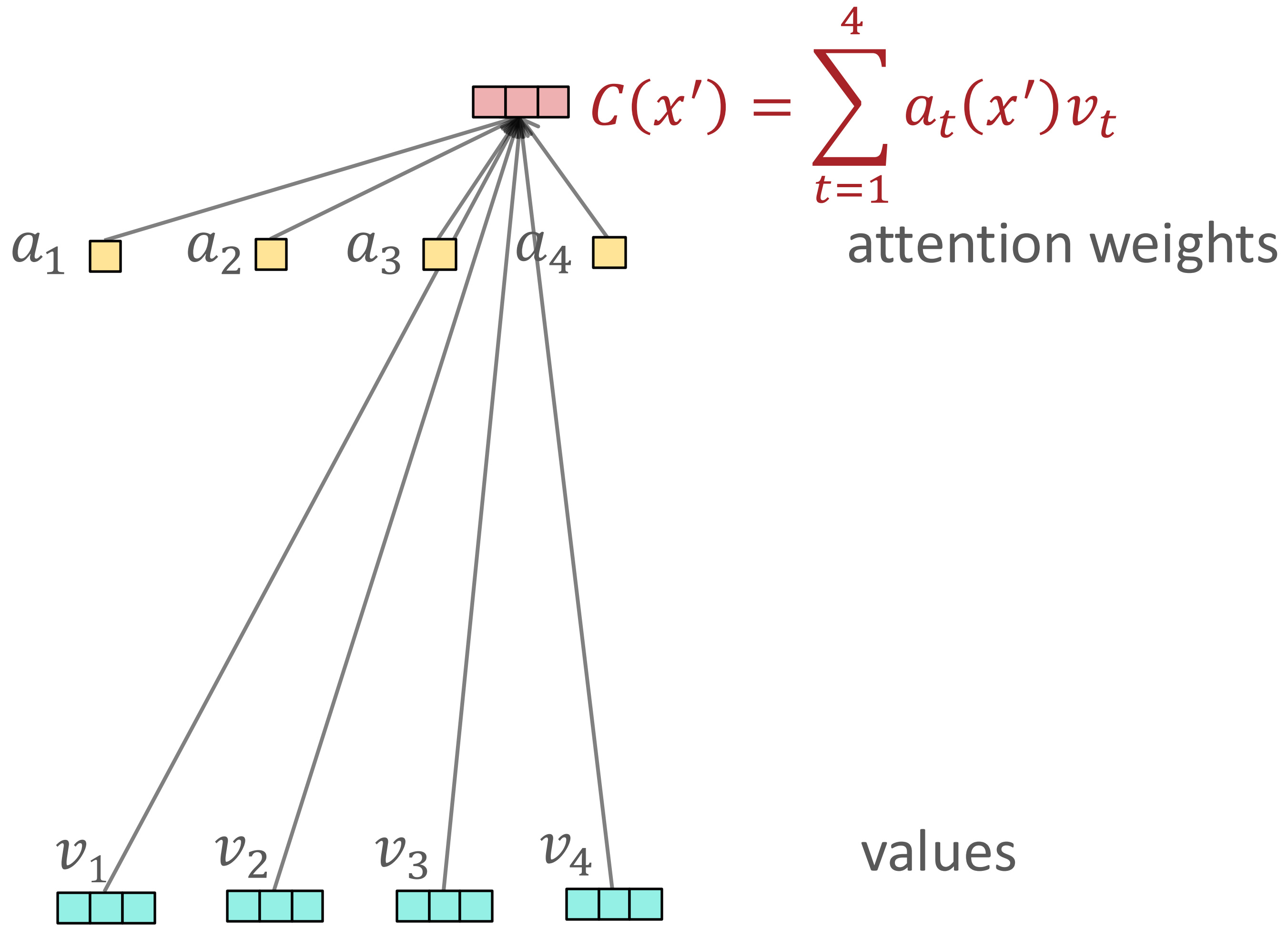
Source: https://arxiv.org/pdf/1506.00019.pdf

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

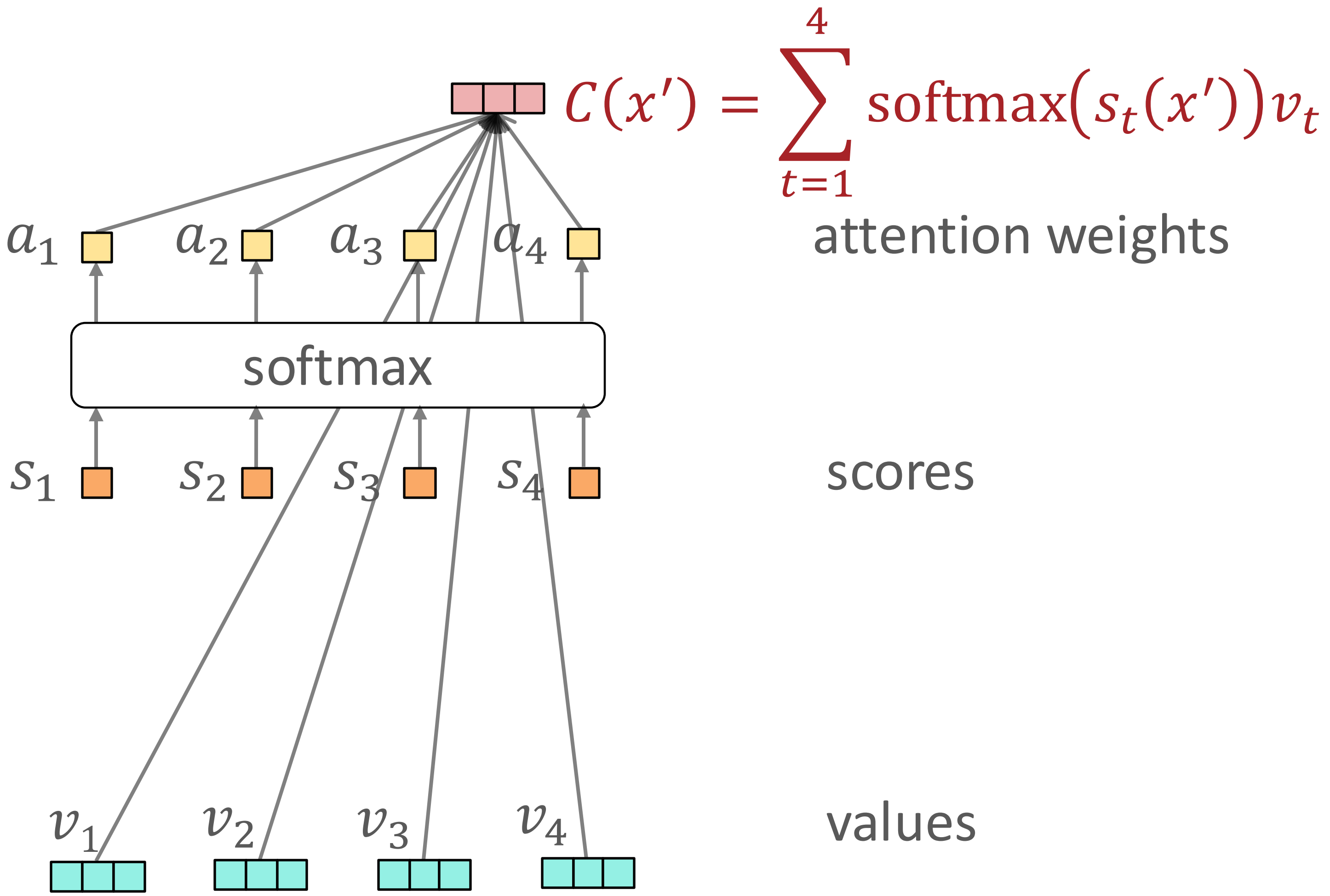
Attention

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



Attention

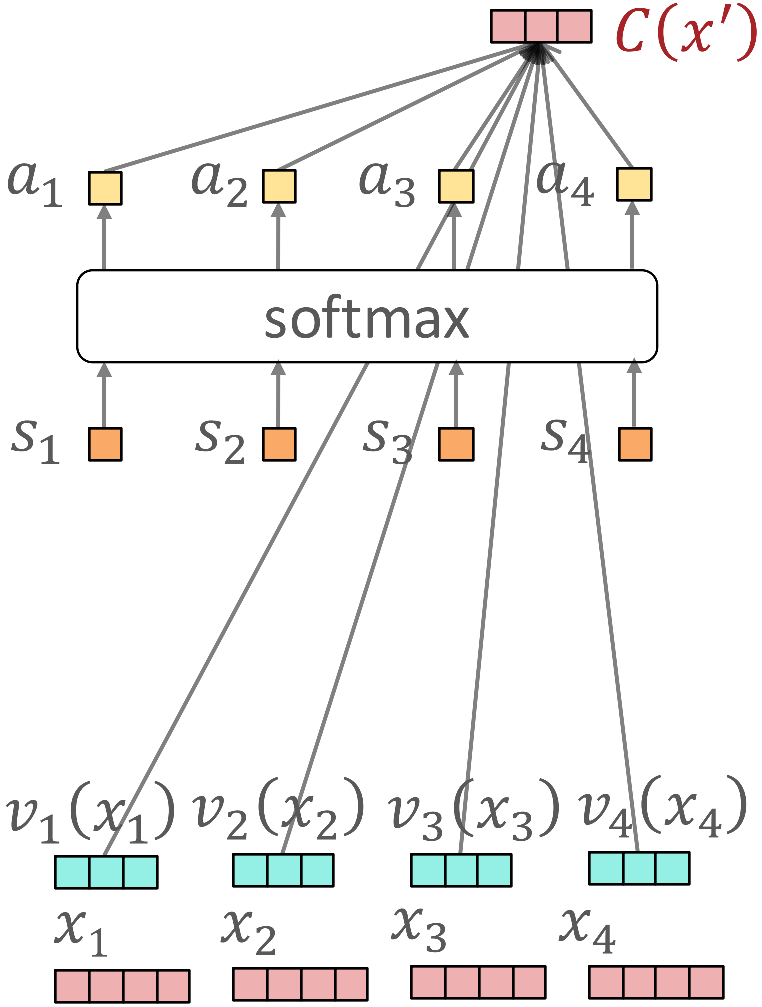
- Approach: compute a representation of the input sequence for each token x' in the decoder



Attention

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

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



attention weights

scores

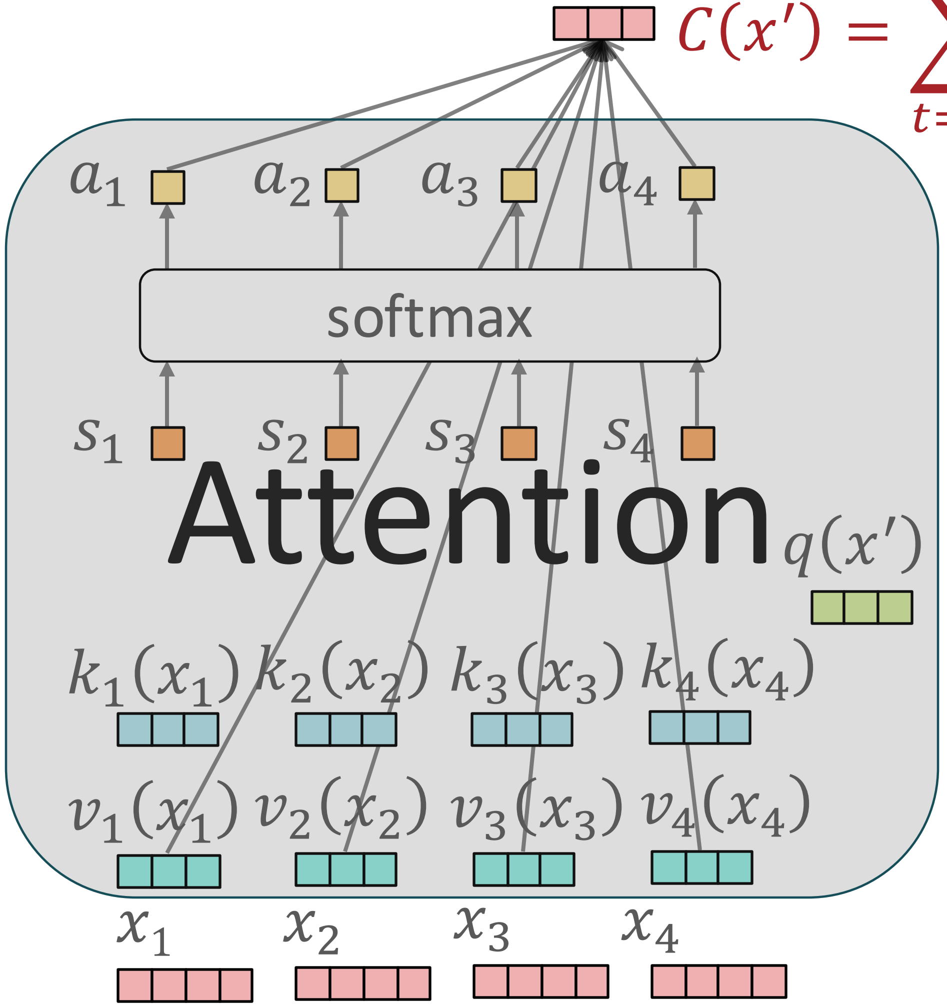
values

input tokens

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

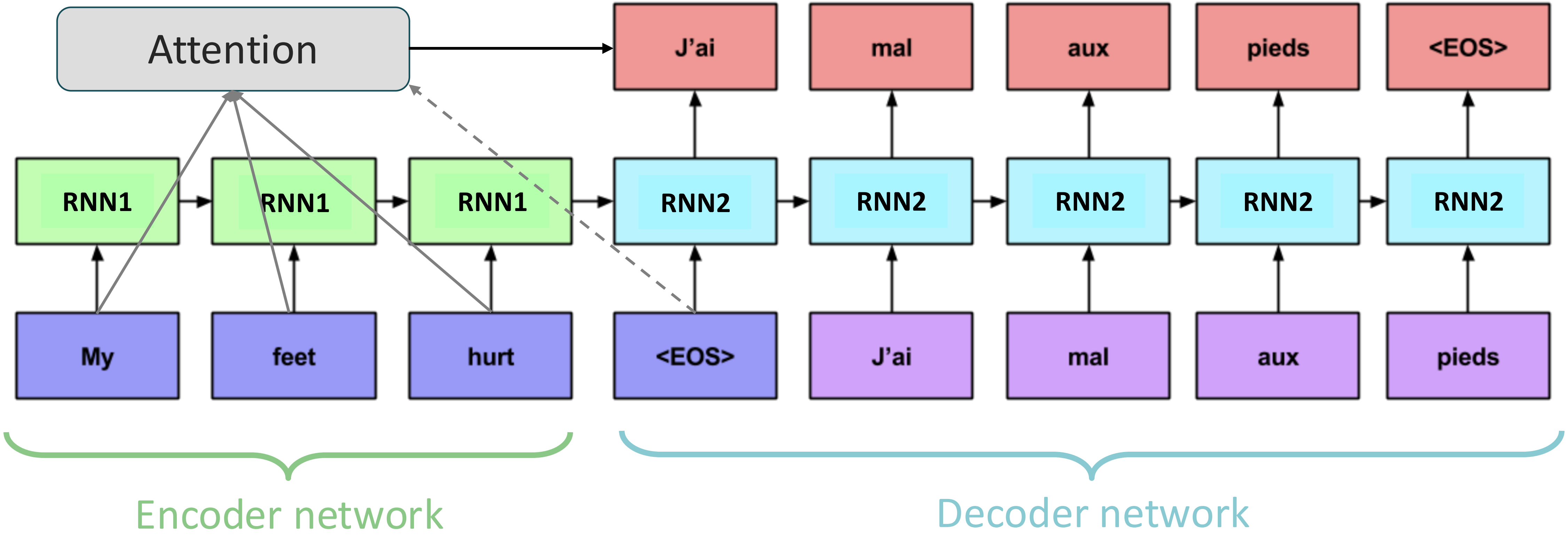
scores: $s_t = \frac{k_t^T q}{\sqrt{\text{length}(q)}}$

query: $q = w_Q^T x'$

keys: $k_t = W_K x_t$

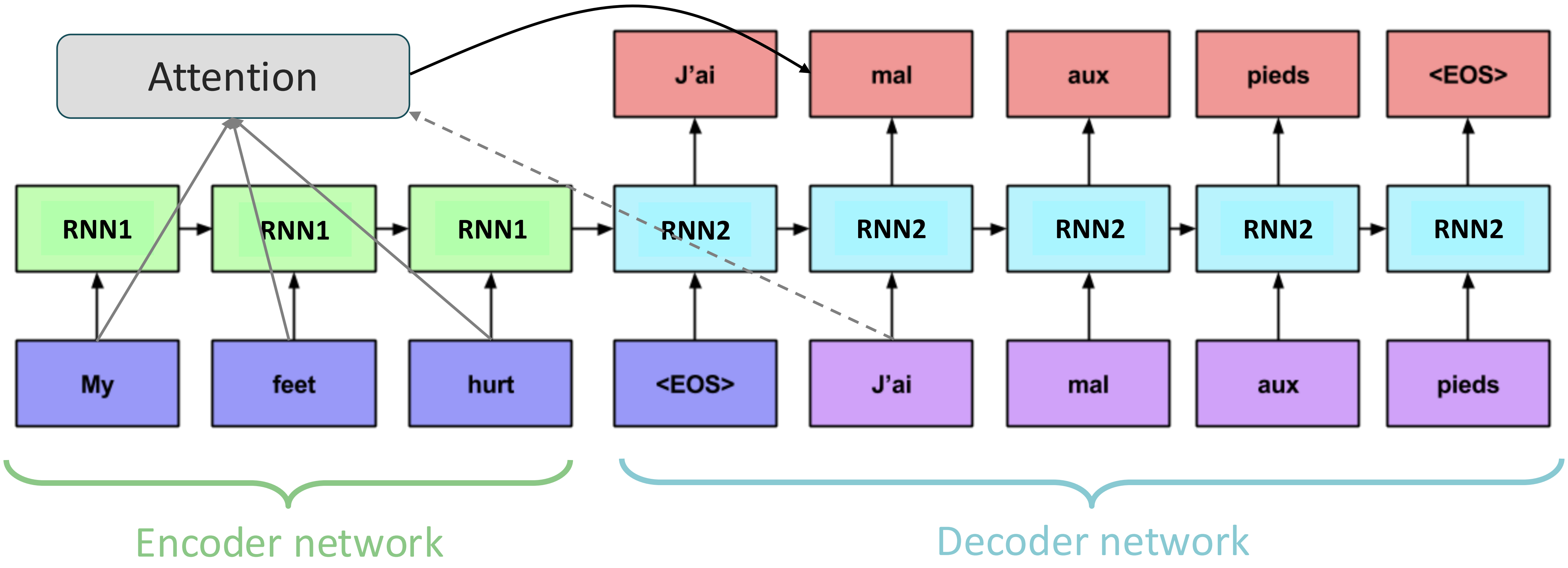
values: $v_t = W_V x_t$

input tokens



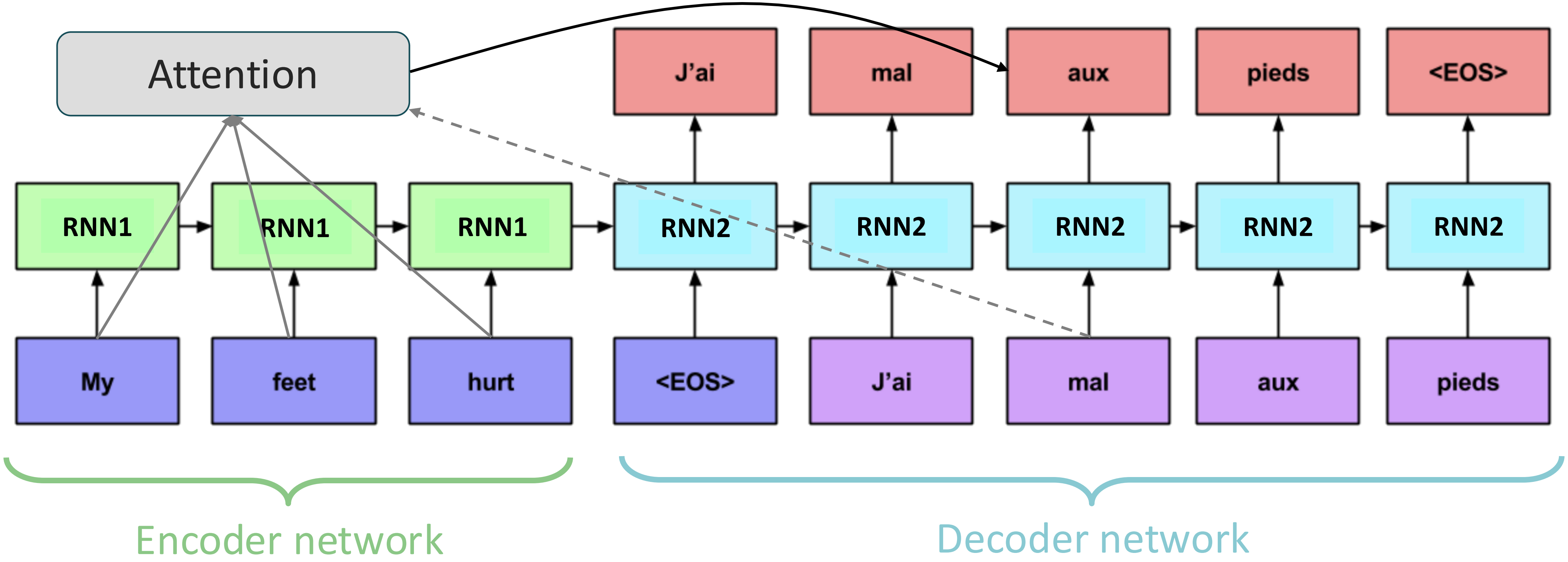
Encoder-Decoder Architectures with Attention

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



Encoder-Decoder Architectures with Attention

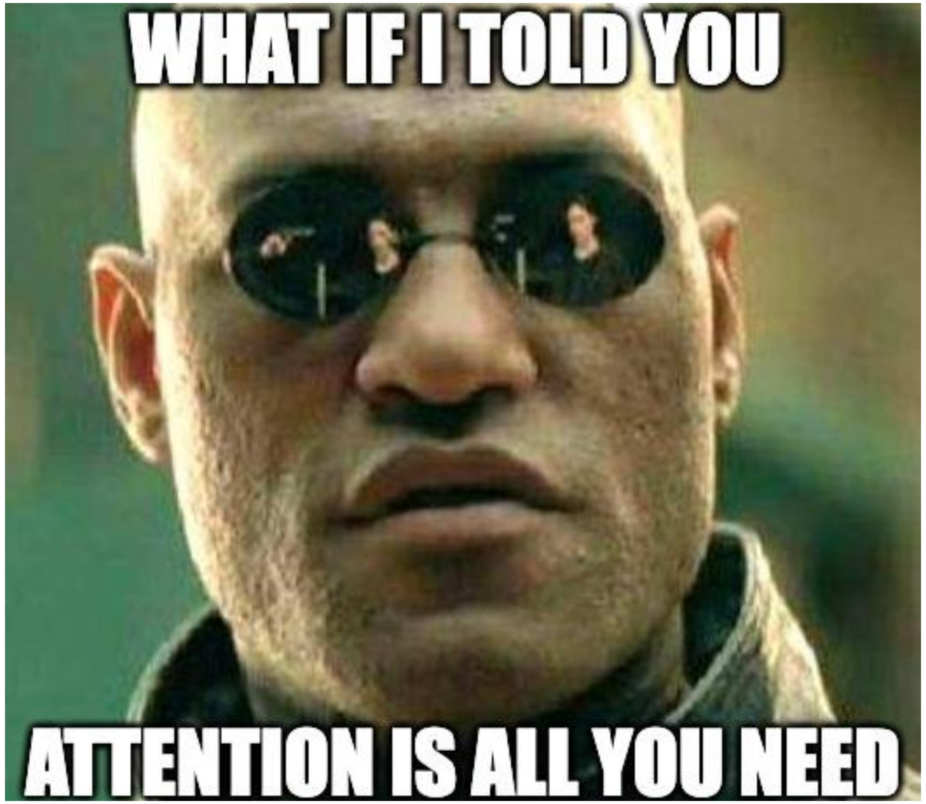
Source: https://arxiv.org/pdf/1506.00019.pdf



Encoder-Decoder Architectures with Attention

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

Attention

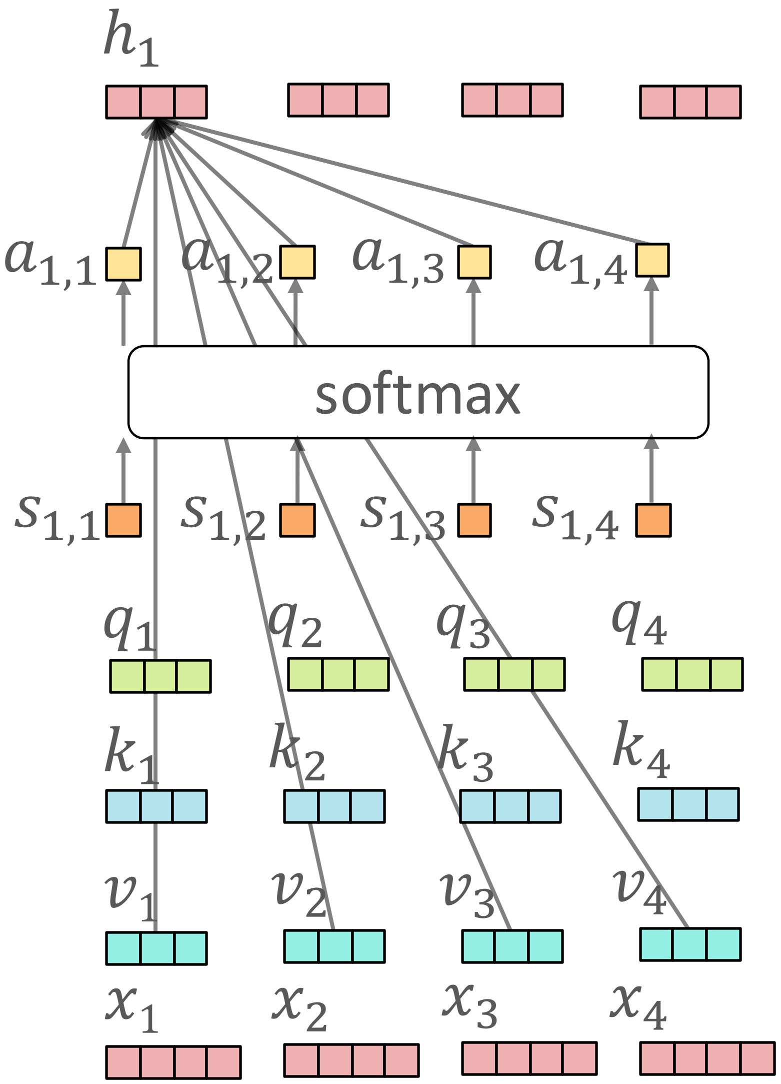


~~Encoder-Decoder Architectures with~~ Attention (Vaswani et al., 2017)

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

queries: $q_t = W_Q x_t$

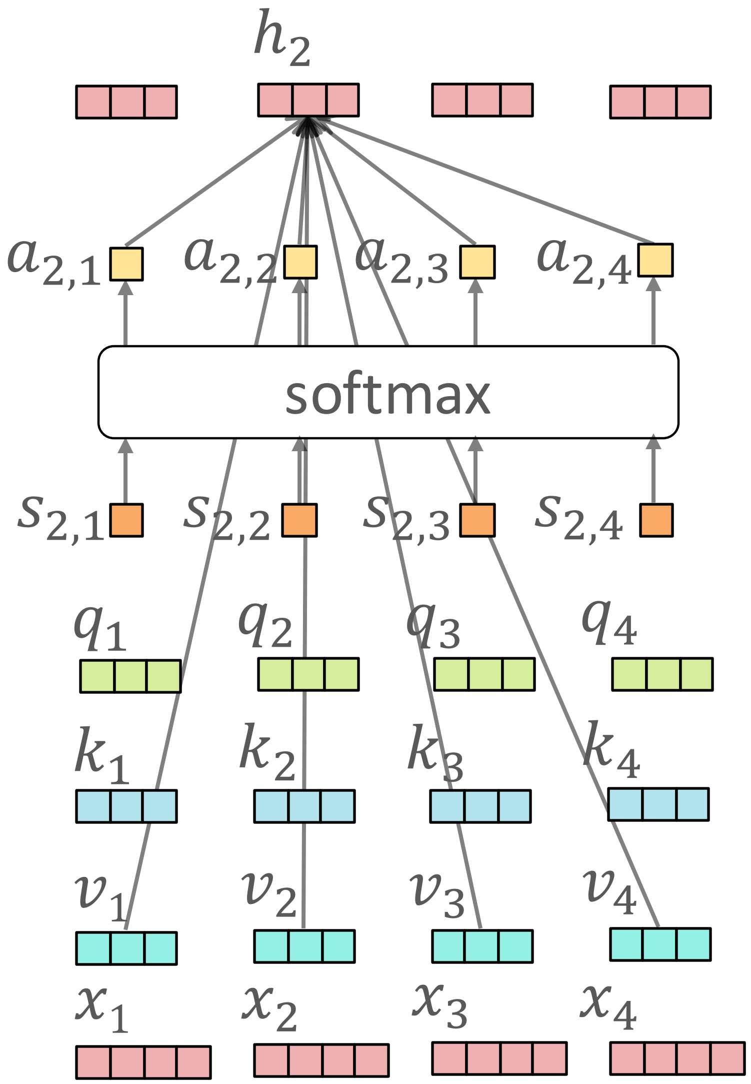
keys: $k_t = W_K x_t$

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

queries: $q_t = W_Q x_t$

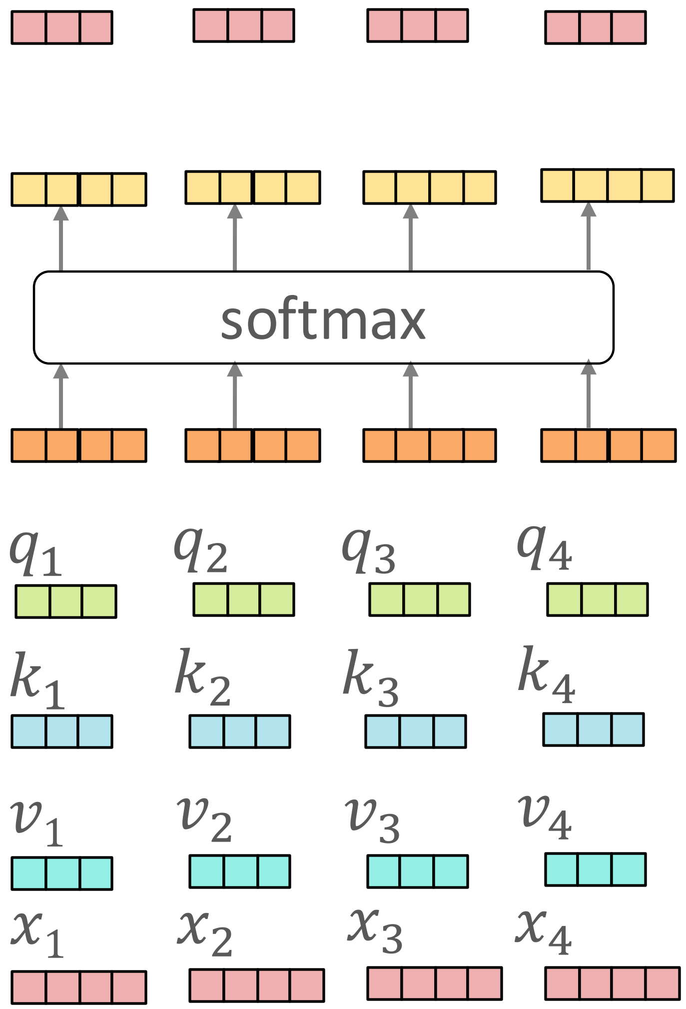
keys: $k_t = W_K x_t$

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



$H = \text{softmax}(S)V$

attention weights

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

queries: $Q = XW_Q$

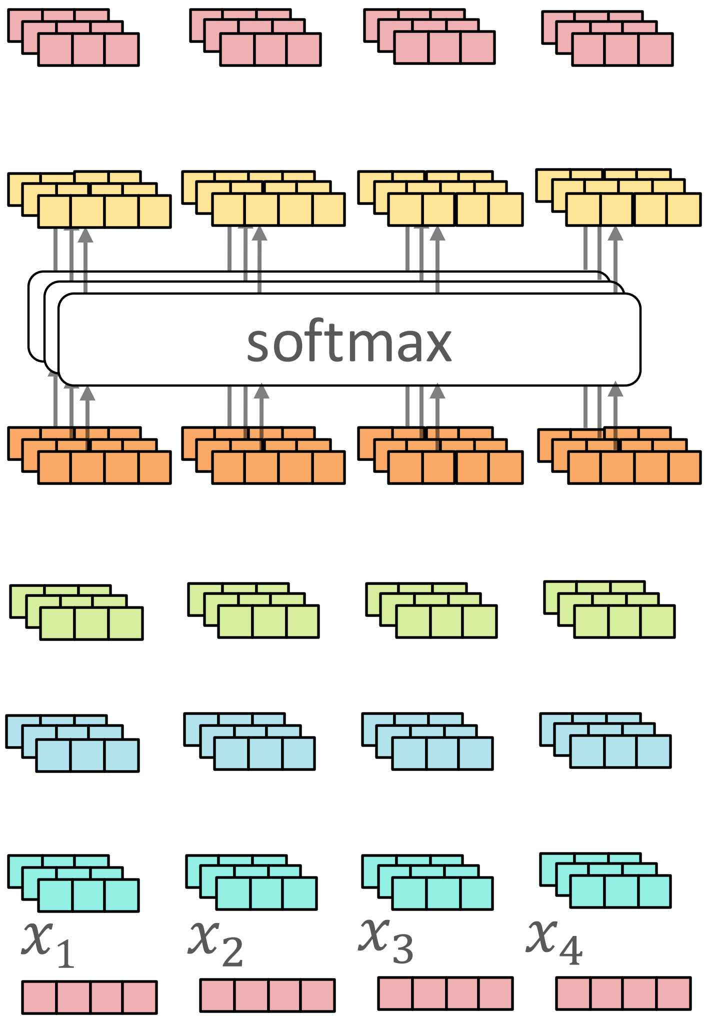
keys: $K = XW_K$

values: $V = XW_V$

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!



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

attention weights

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

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

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

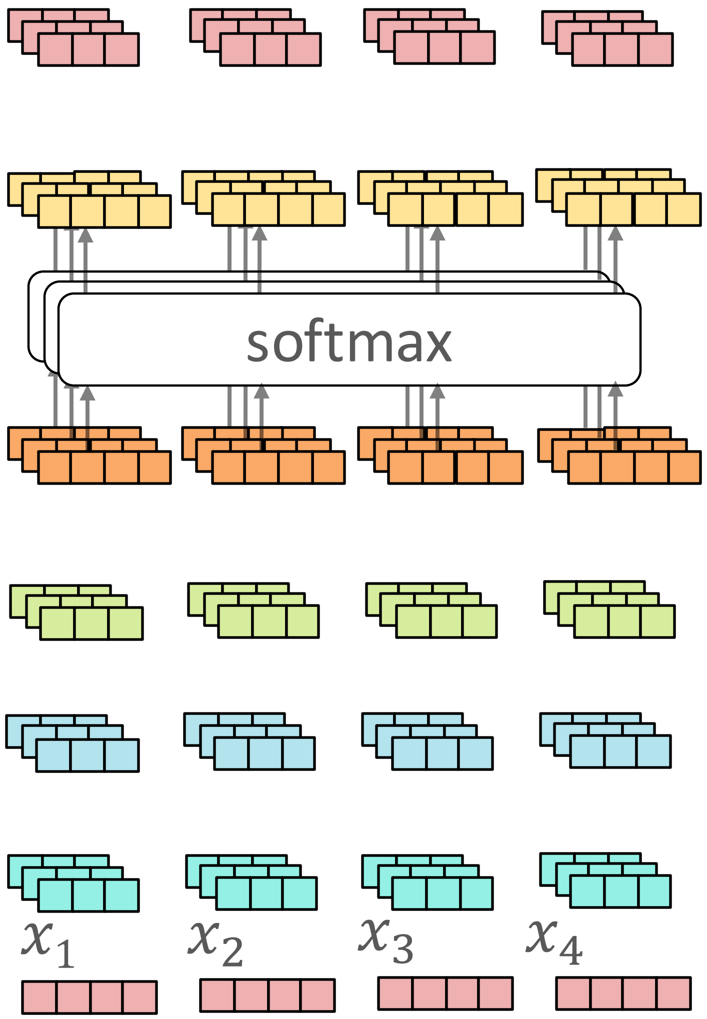
values: $V^{(h)} = XW_V^{(h)}$

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

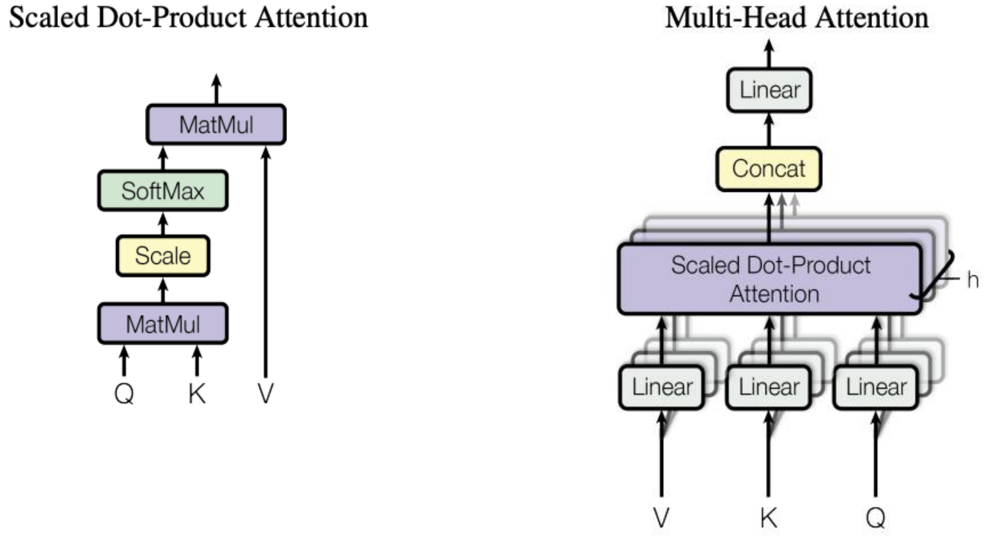
keys: $K^{(h)} = XW_K^{(h)}$

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!



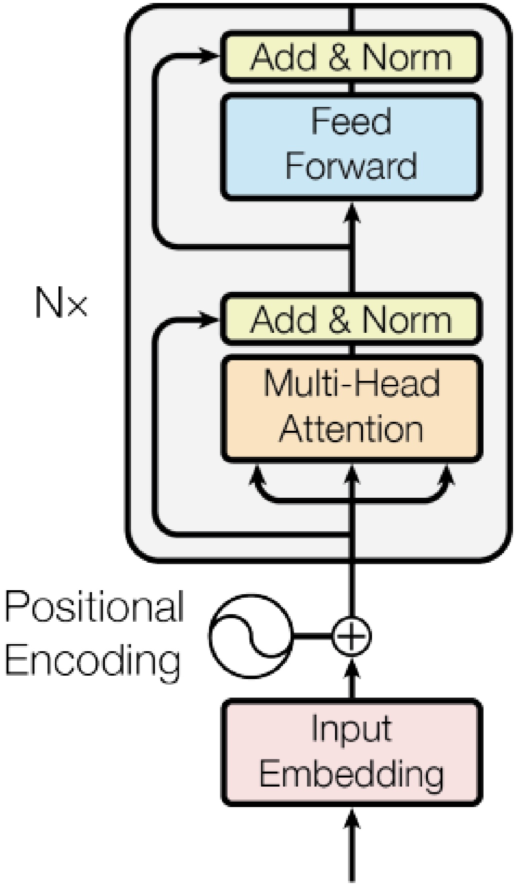
- The outputs from all the attention heads are concatenated together to get the final representation

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

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

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

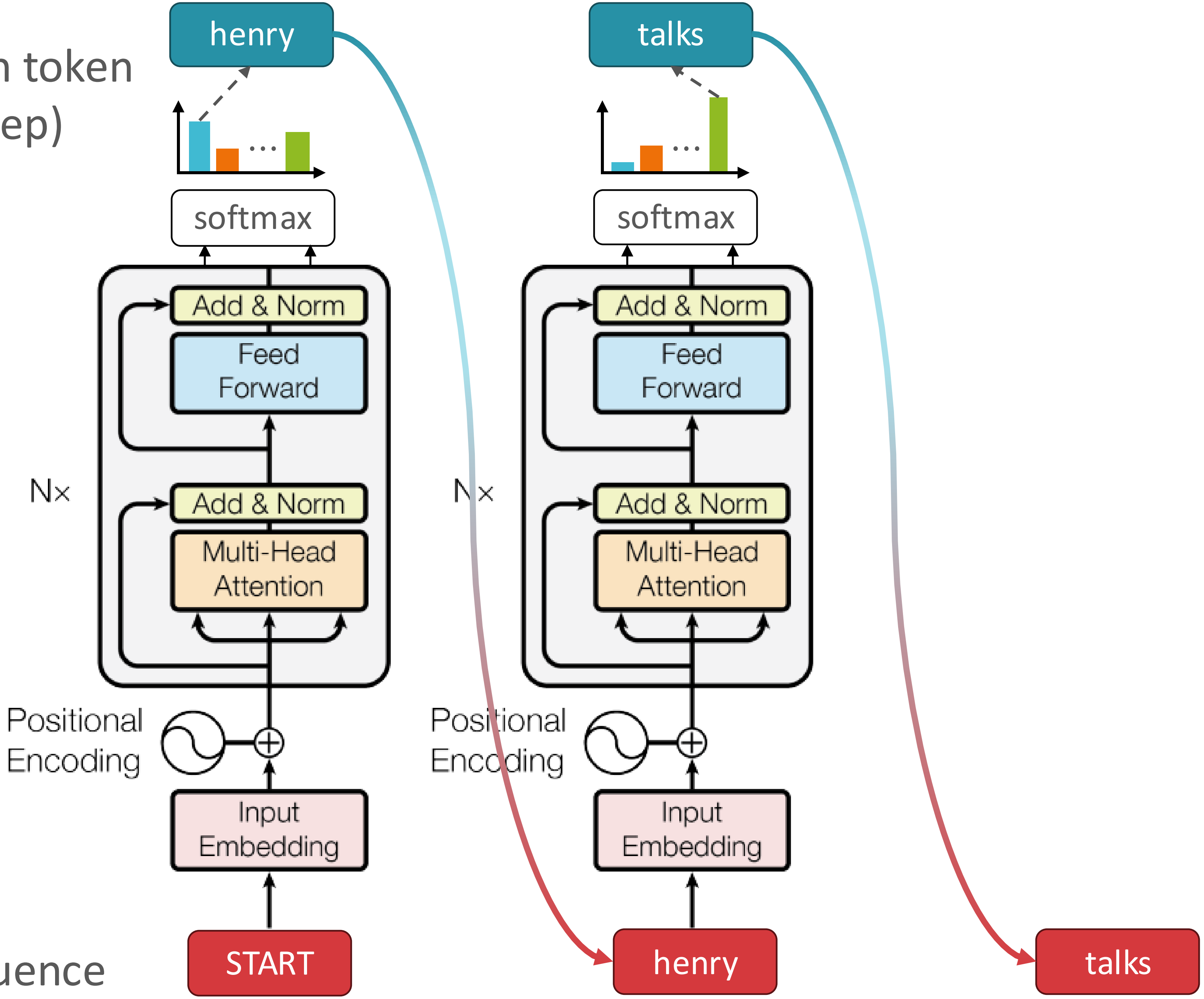
Transformers



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

Transformer Language Models

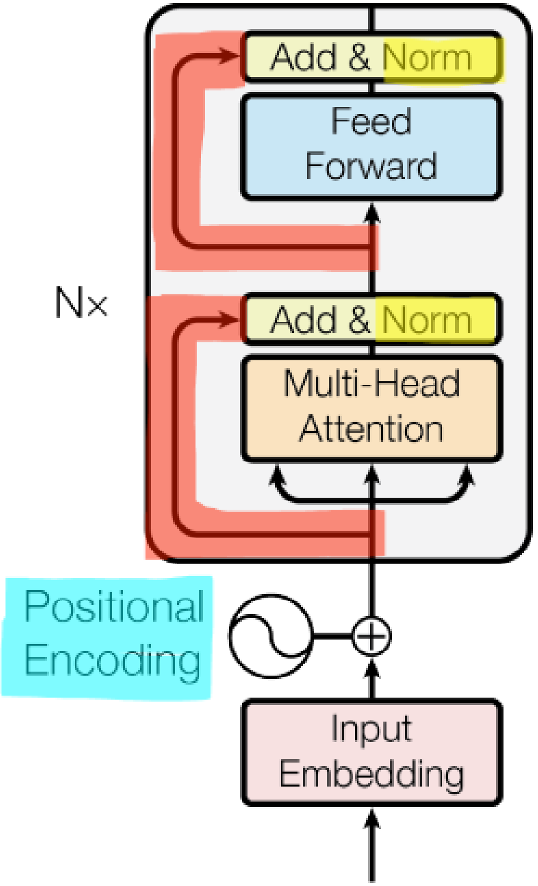
Generated sequence (use each token as the input to the next timestep)



Input sequence

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

Transformers

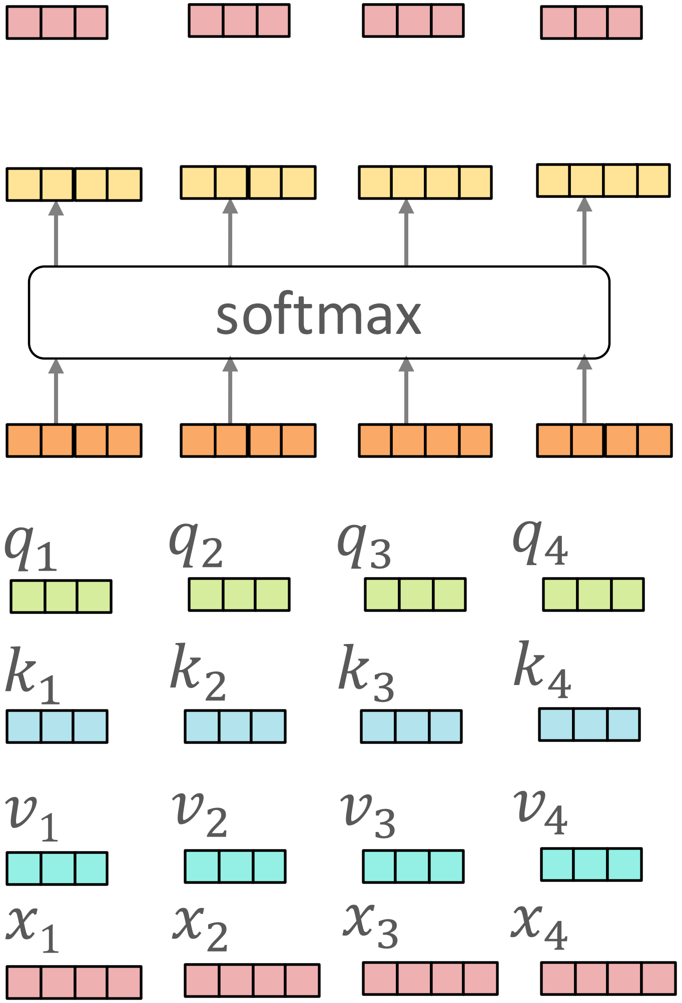


- In addition to multi-head attention, transformer architectures use
  1. Positional encodings
  2. Layer normalization
  3. Residual connections
  4. A fully-connected feed-forward network

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

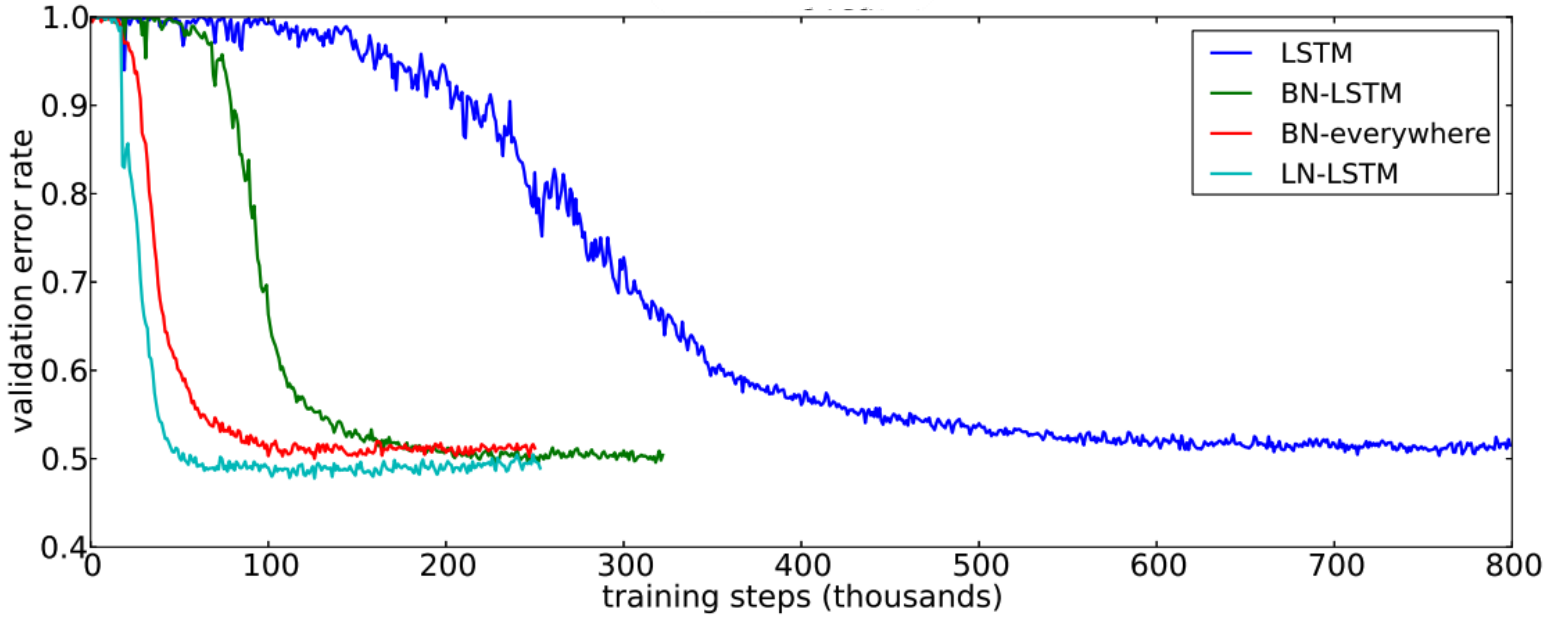
Layer Normalization

- 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

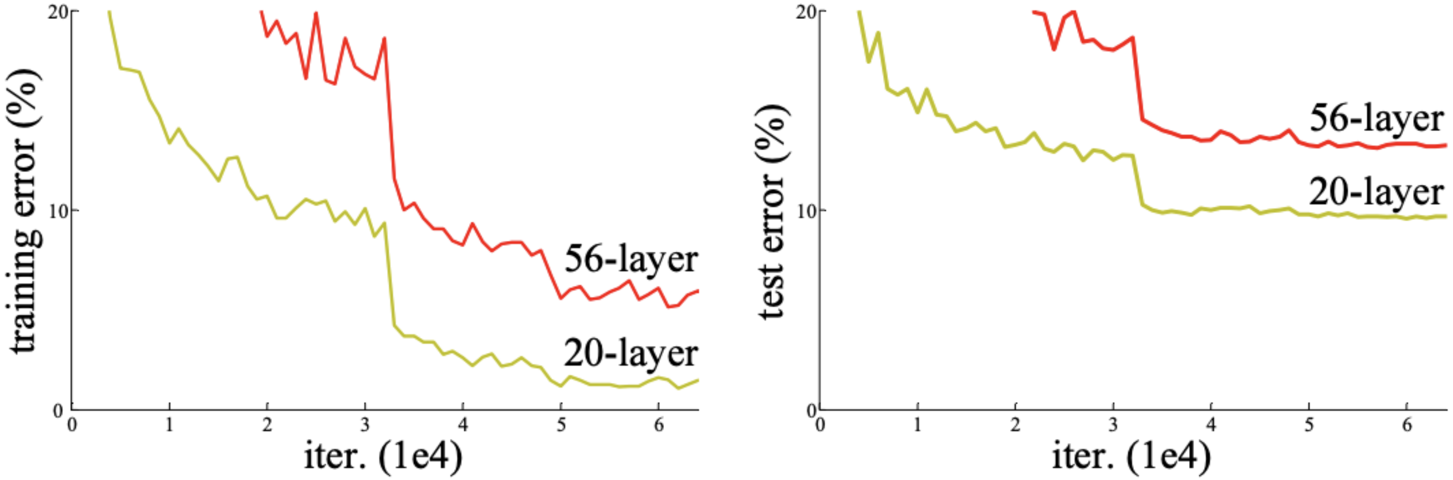
$$H' = \gamma\left(\frac{H-\mu}{\sigma}\right) + \beta$$

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

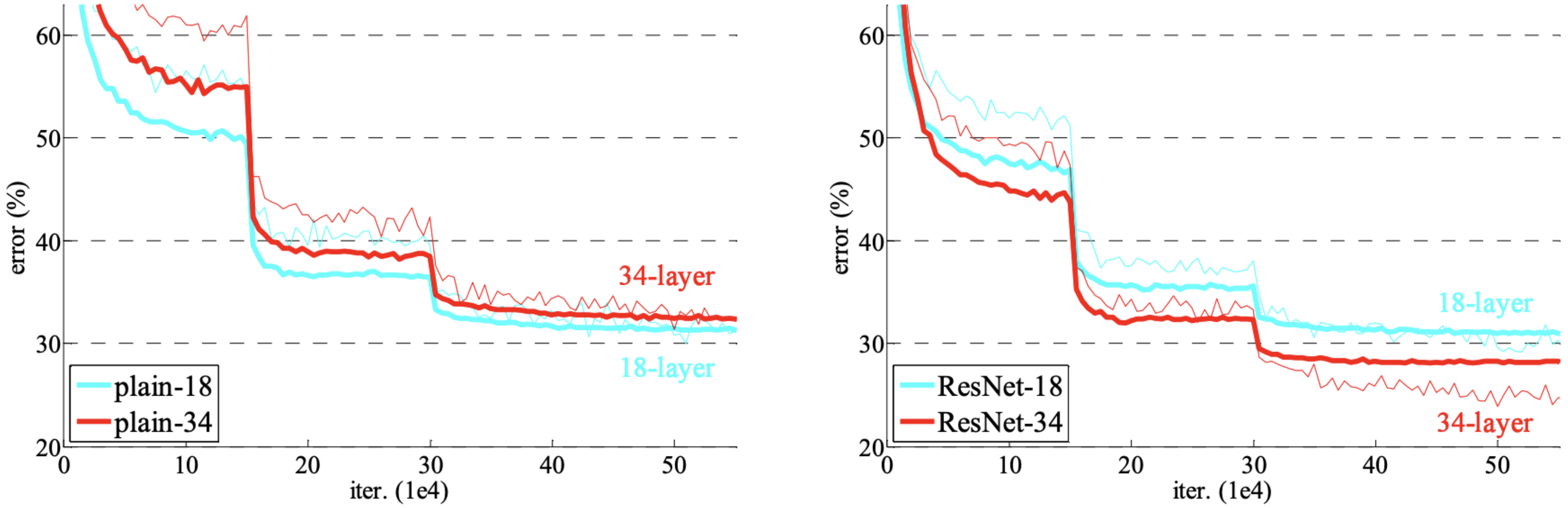
$$H' = H\left(x^{(i)}\right) + x^{(i)}$$

- 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

- 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