10-701: Introduction to Machine Learning

Lecture 13 - Attention & Transformers

Hoda Heidari

* Slides adopted from F24 offering of 10701 by Henry Chai.

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 representation from earlier hidden layers into later ones.

Recurrent Neural Networks

- Data points consists of (input **sequence**, label **sequence**) pairs, potentially of **varying lengths**

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

$x^{(i)}$

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

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

Recurrent Neural Networks

- RNNs process inputs one time step at a time, using **recurrence**:

hidden state/ representation for $x^{(i)}$

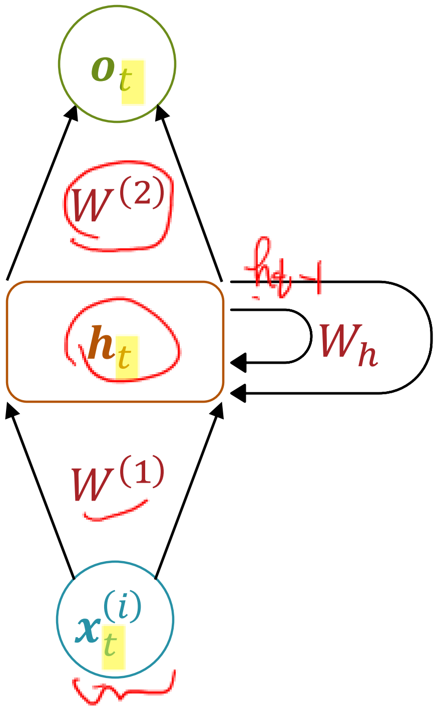
$\boldsymbol{h}_t = \left[1, \theta\left(W^{(1)}\boldsymbol{x}_t^{(i)} + W_h\boldsymbol{h}_{t-1}\right)\right]^T$ and $\boldsymbol{o}_t = \hat{y}_t^{(i)} = \theta\left(W^{(2)}\boldsymbol{h}_t\right)$

Where $\boldsymbol{h}_t$ serves as a summary or latent representation of the sequence up to time t.

- The same parameters $W^{(1)}$, W_h and $W^{(2)}$ are reused at every step.
- We can unroll the RNN for as many time steps as the sequence requires.
- So, at training and inference time, the RNN can run for different numbers of steps depending on the input length.

Recurrent Neural Networks

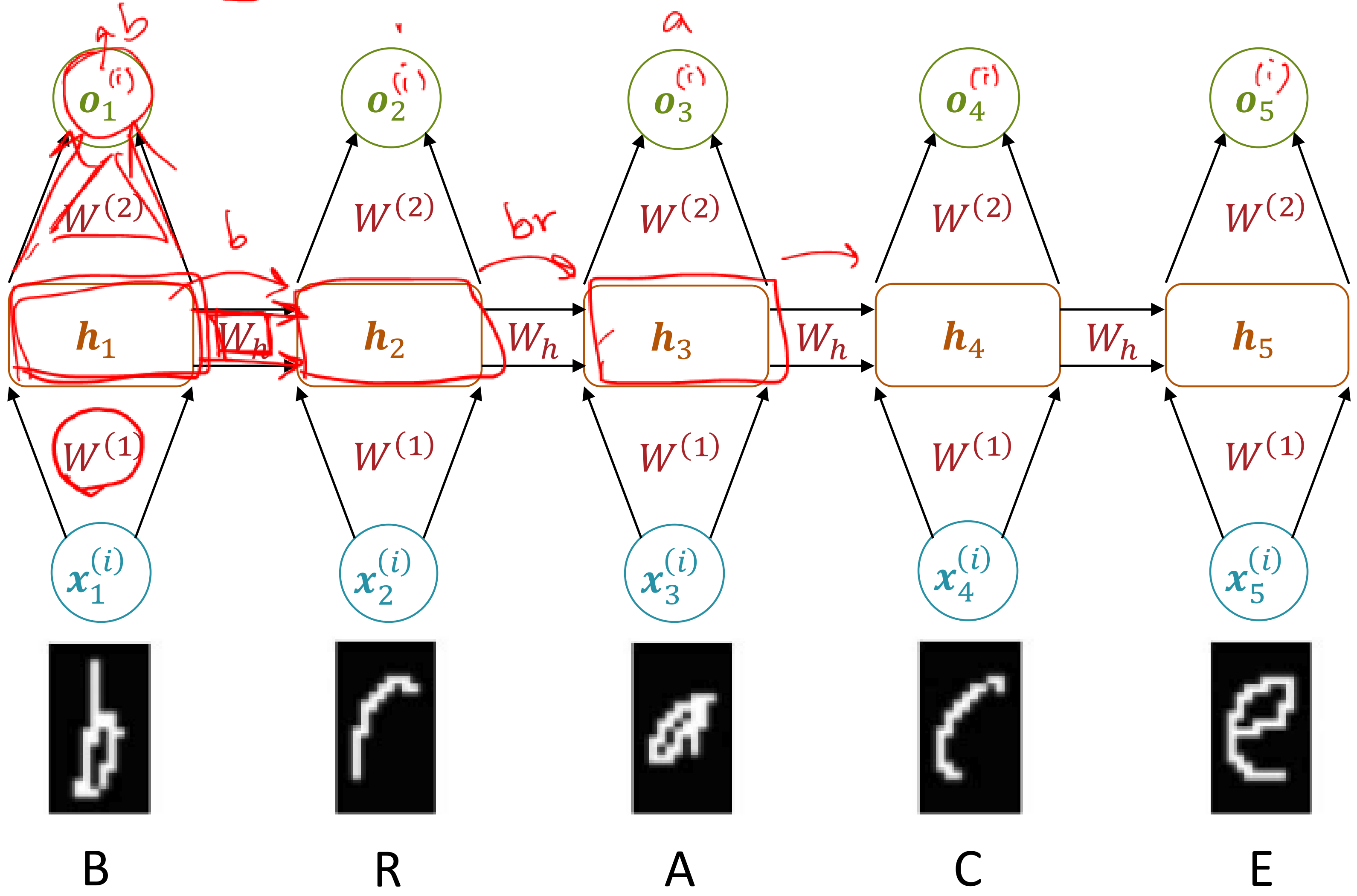
$$\boldsymbol{h}_t = \left[1, \theta\left(W^{(1)}\boldsymbol{x}_t^{(i)} + W_h\boldsymbol{h}_{t-1}\right)\right]^T \text{ and } \boldsymbol{o}_t = \hat{y}_t^{(i)} = \theta\left(W^{(2)}\boldsymbol{h}_t\right)$$



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

Unrolling Recurrent Neural Networks

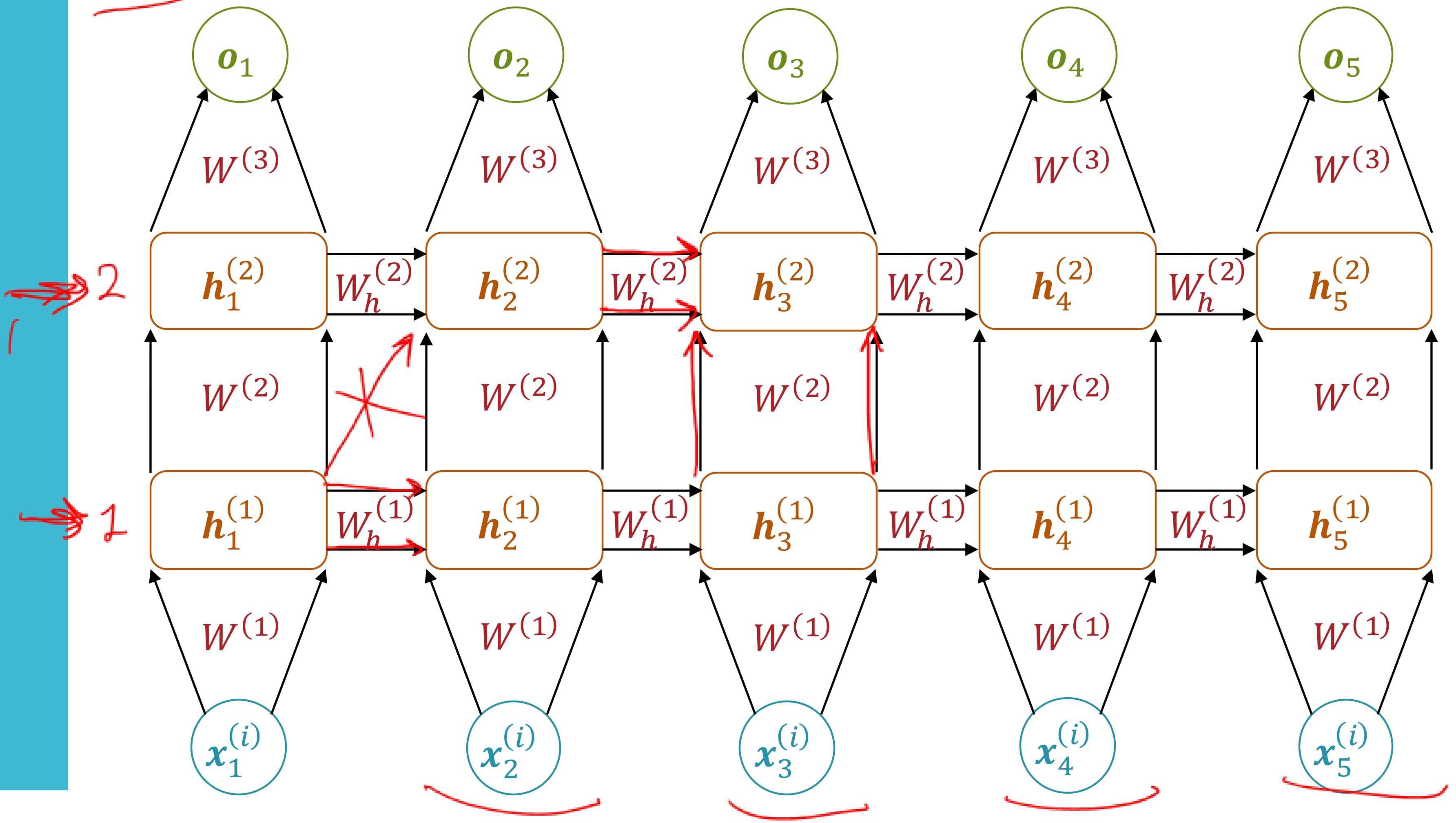
$$\boldsymbol{h}_t = \left[1, \theta\left(W^{(1)}\boldsymbol{x}_t^{(i)} + \boxed{W_h}\boldsymbol{h}_{t-1}\right)\right]^T \text{ and } \boldsymbol{o}_t = \hat{y}_t^{(i)} = \theta\left(W^{(2)}\boldsymbol{h}_t\right)$$



Source: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6319312

Deep Recurrent Neural Networks

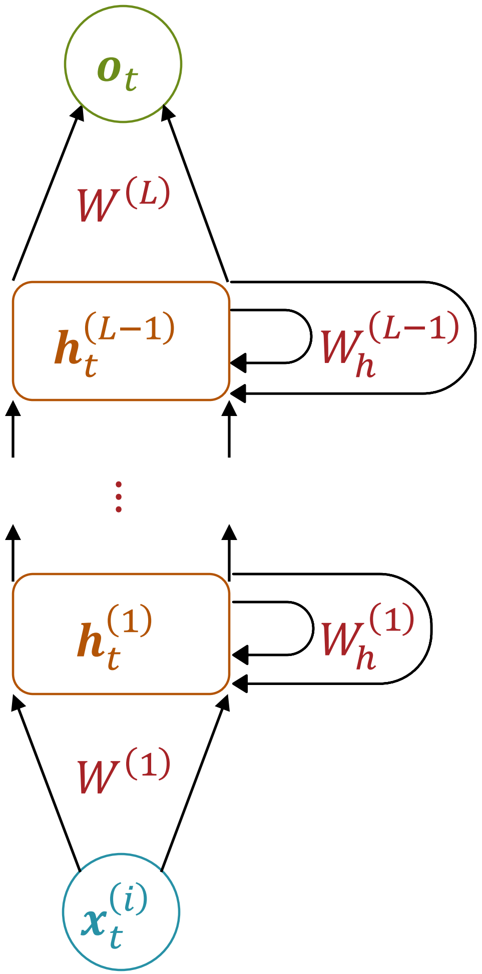
$$\boldsymbol{h}_t^{(l)} = \left[1, \theta\left(W^{(l)}\boldsymbol{h}_t^{(l-1)} + W_h^{(l)}\boldsymbol{h}_{t-1}^{(l)}\right)\right]^T \text{ and } \boldsymbol{o}_t = \hat{y}_t^{(i)} = \theta\left(W^{(L)}\boldsymbol{h}_t^{(L-1)}\right)$$



Source: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6319312

Deep Recurrent Neural Networks

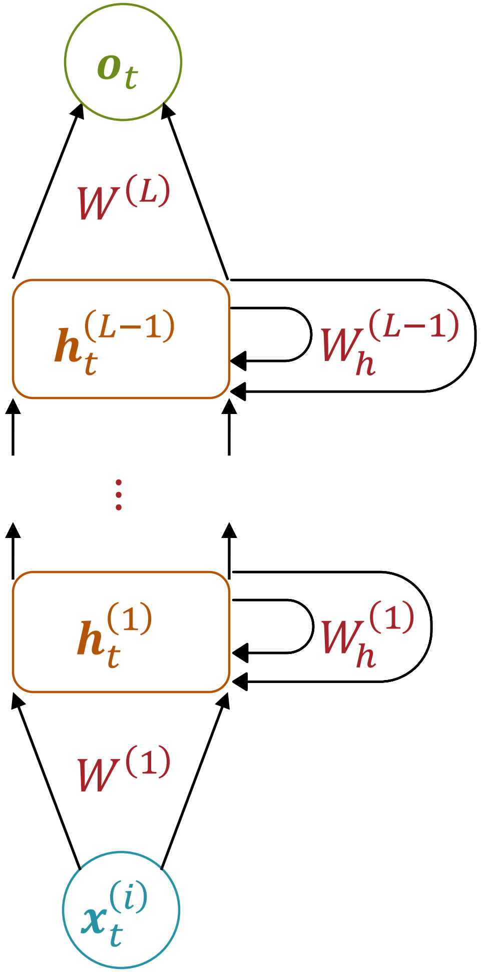
$$\boldsymbol{h}_t^{(l)} = \left[1, \theta\left(W^{(l)}\boldsymbol{h}_t^{(l-1)} + W_h^{(l)}\boldsymbol{h}_{t-1}^{(l)}\right)\right]^T \text{ and } \boldsymbol{o}_t = \hat{y}_t^{(i)} = \theta\left(W^{(L)}\boldsymbol{h}_t^{(L-1)}\right)$$



Source: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6319312

But why do we only pass information forward? What if later time steps have useful information as well?

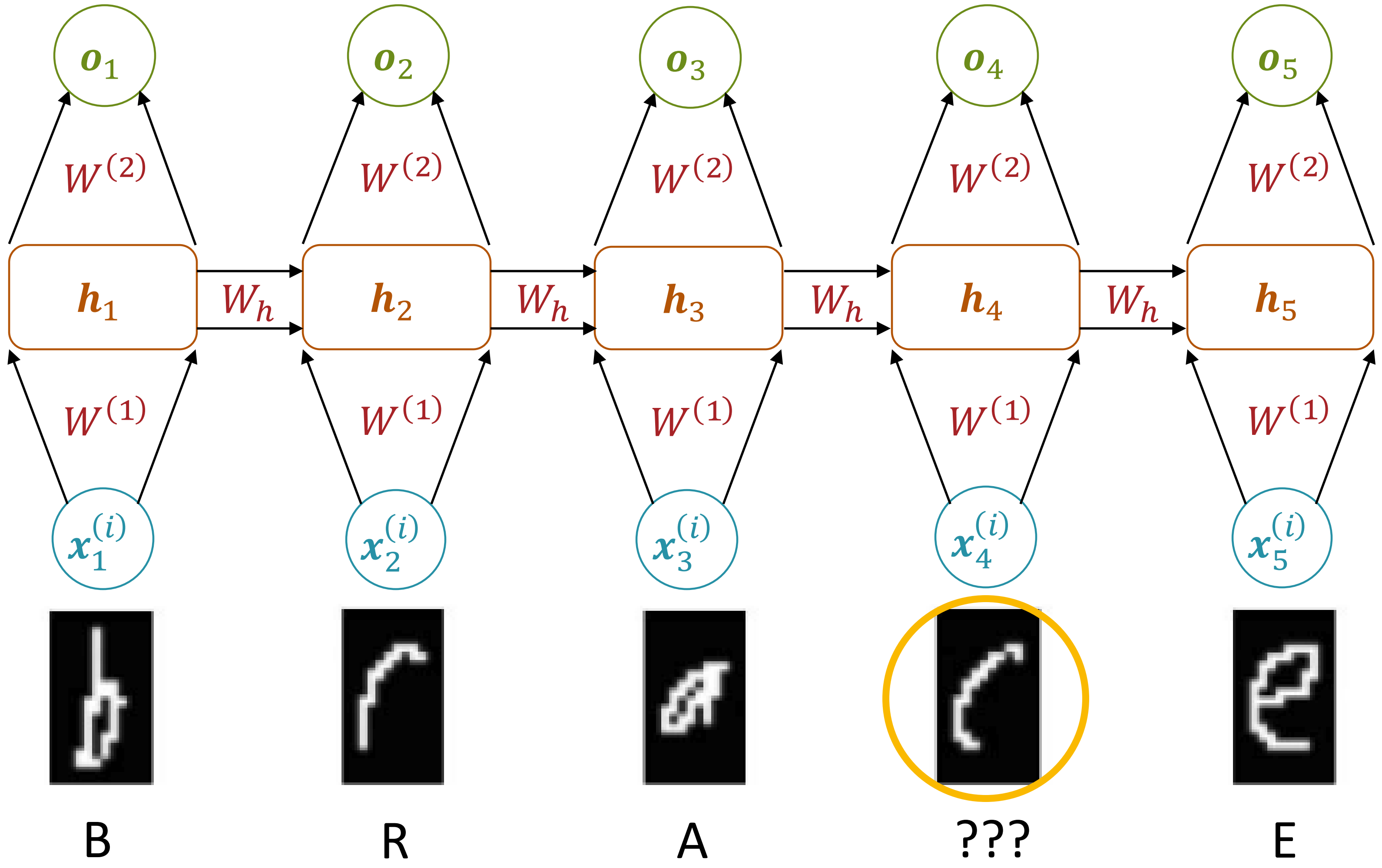
$$\boldsymbol{h}_t^{(l)} = \left[1, \theta\left(W^{(l)}\boldsymbol{h}_t^{(l-1)} + W_h^{(l)}\boldsymbol{h}_{t-1}^{(l)}\right)\right]^T \text{ and } \boldsymbol{o}_t = \hat{y}_t^{(i)} = \theta\left(W^{(L)}\boldsymbol{h}_t^{(L-1)}\right)$$



Source: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6319312

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

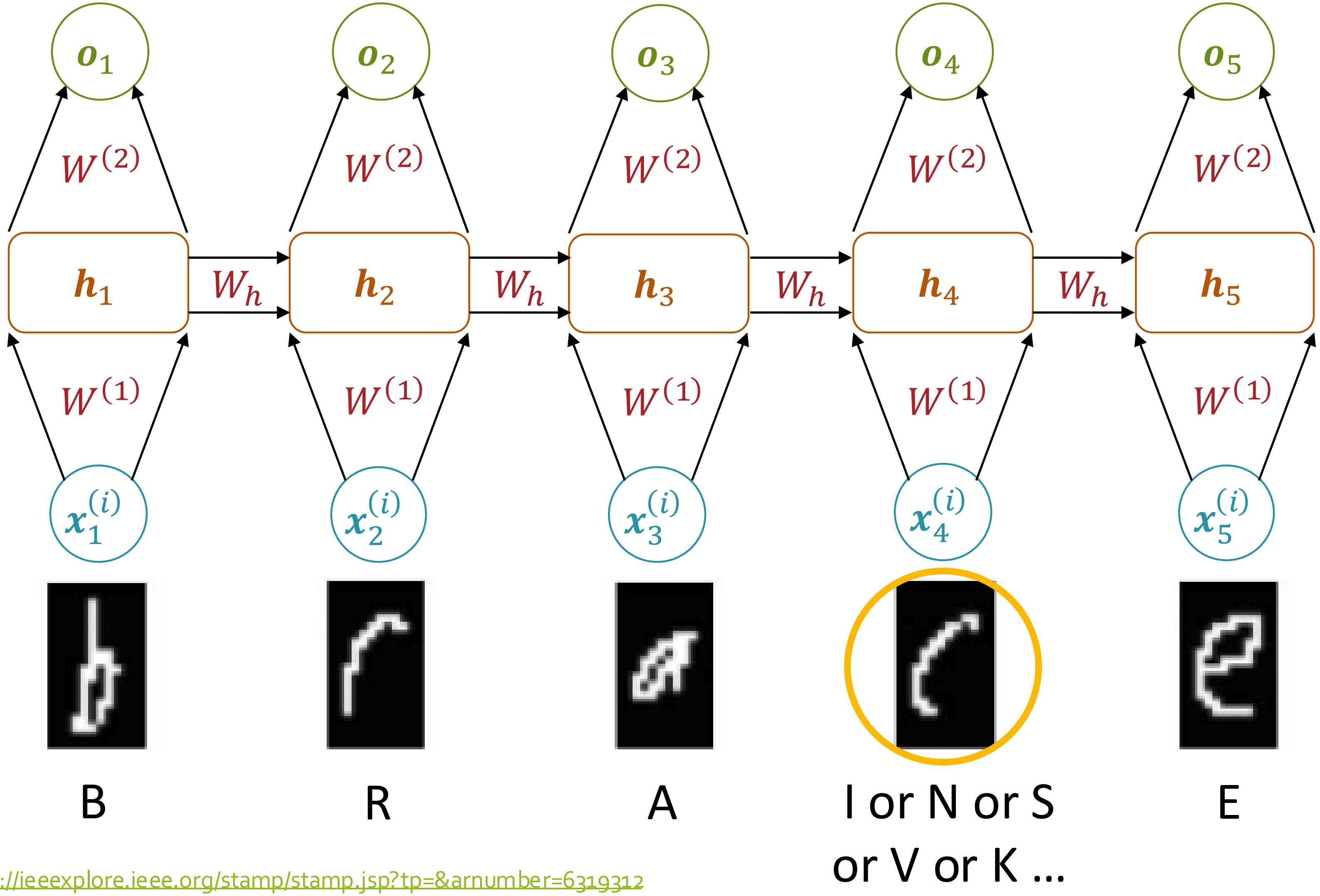
But why do we only pass information forward? What if later time steps have useful information as well?



Source: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6319312

But why do we only pass information forward? What if later time steps have useful information as well?

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



Source: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6319312

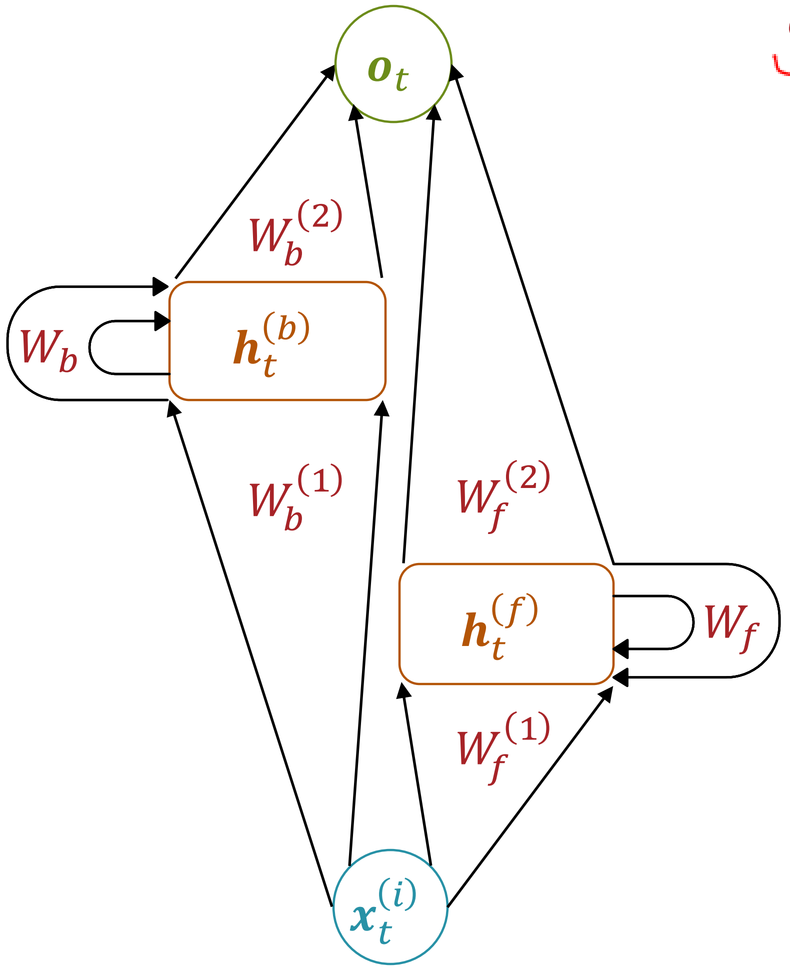
Bidirectional Recurrent Neural Networks

- Bidirectional Recurrent Neural Networks (BiRNNs) capture **context from both the past and the future** of a sequence.
- A BiRNN has two RNNs:
  - one $\boldsymbol{h}_t^{(f)}$ processes the sequence **forward in time**
  - one $\boldsymbol{h}_t^{(b)}$ processes it **backward in time**
  - The combination contains information from the entire sequence centered around position t.

Bidirectional Recurrent Neural Networks

$$\boldsymbol{h}_t^{(f)} = \left[1, \theta\left(W_f^{(1)} \boldsymbol{x}_t^{(i)} + W_f \boldsymbol{h}_{t-1}^{(f)}\right)\right]^T \text{ and } \boldsymbol{h}_t^{(b)} = \left[1, \theta\left(W_b^{(1)} \boldsymbol{x}_t^{(i)} + W_b \boldsymbol{h}_{t+1}^{(b)}\right)\right]^T$$

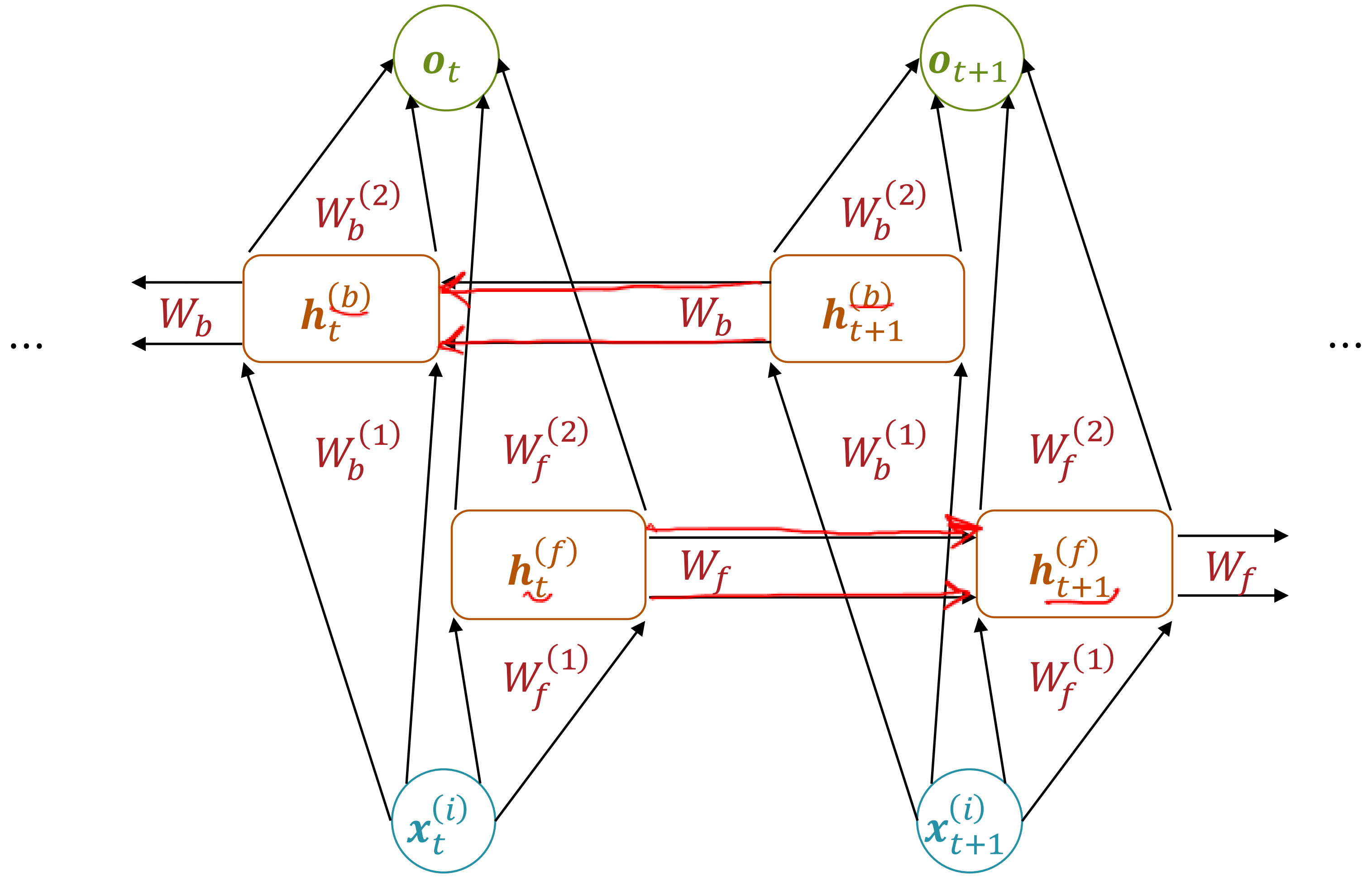
$$\boldsymbol{o}_t = \hat{y}_t^{(i)} = \theta\left(W_f^{(2)} \boldsymbol{h}_t^{(f)} + W_b^{(2)} \boldsymbol{h}_t^{(b)}\right)$$



Unrolling Bidirectional Recurrent Neural Networks

$$\boldsymbol{o}_t = \hat{y}_t^{(i)} = \theta\left(W_f^{(2)}\boldsymbol{h}_t^{(f)} + W_b^{(2)}\boldsymbol{h}_t^{(b)}\right)$$

$\boldsymbol{h}_t^{(f)} = \left[1, \theta\left(W_f^{(1)}\boldsymbol{x}_t^{(i)} + W_f\boldsymbol{h}_{t-1}\right)\right]^T$ and $\boldsymbol{h}_t^{(b)} = \left[1, \theta\left(W_b^{(1)}\boldsymbol{x}_t^{(i)} + W_b\boldsymbol{h}_{t+1}\right)\right]^T$



Inference via Bidirectional Recurrent Neural Networks

- Inference when the entire sequence is available (e.g., sequence labeling, sentence-level classification)
  1. run forward RNN on the full sequence;
  2. Run backward RNN on the reversed sequence;
  3. Combine representations at time t: $[\boldsymbol{h}_t^{(f)}, \boldsymbol{h}_t^{(b)}]$
  4. Feed the combination to a classifier

$$\hat{y}_t^{(i)} = \theta\left(W_f^{(2)}\boldsymbol{h}_t^{(f)} + W_b^{(2)}\boldsymbol{h}_t^{(b)}\right)$$

Training RNNs

- A (deep/bidirectional) RNN simply represents a (somewhat complicated) computation graph
  - Weights ($W^{(1)}$, W_h and $W^{(2)}$) are shared between different timesteps, significantly reducing the number of parameters to be learned!
- Can be trained using (stochastic) gradient descent/ backpropagation → "backpropagation through time"

$x_1^{(i)}, x_2^{(i)}, \ldots, x_t^{(i)}, \ldots, x_T^{(i)}$ predictions: $\hat{y}_1^{(i)}, \ldots, \hat{y}_t^{(i)}, \ldots, \hat{y}_T^{(i)}$

$y_1^{(i)}, y_2^{(i)}, \ldots, y_t^{(i)}, \ldots, y_T^{(i)}$

Loss Functions for RNNs

- **Sequence-to-sequence prediction:** Token-wise cross-entropy averaged across time steps

$$\mathcal{L} = \frac{1}{T}\sum_t CE(y_t, softmax(W h_t))$$

- **Regression over sequences** (e.g., forecasting, speech features): MSE between predicted and target sequence

$$\mathcal{L} = \frac{1}{T}\sum_t \| y_t - \hat{y}_t \|^2$$

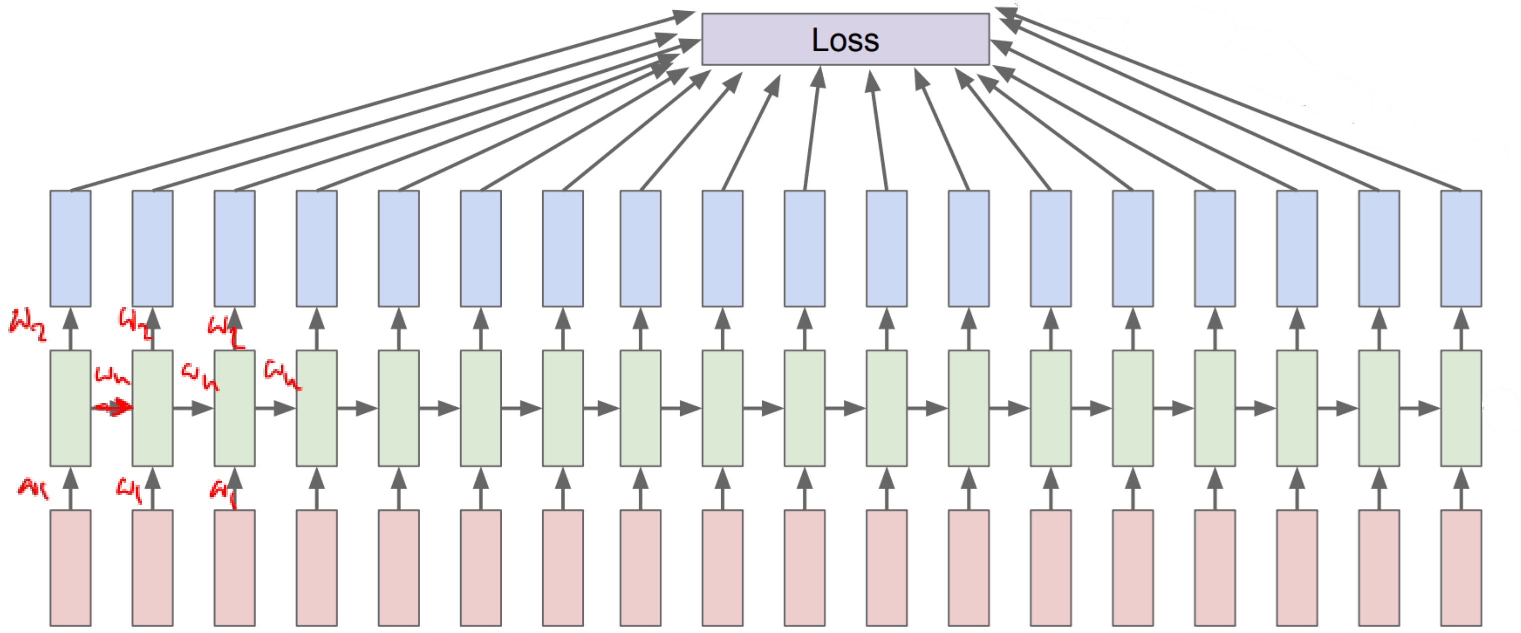
- **Sequence classification** (one label per sequence, e.g., sentiment classification): Cross-entropy on the final (or pooled) hidden state

$$\mathcal{L} = CE(y, softmax(W h_T))$$

Backprop Through Time

- Each hidden state h_t influences not only its immediate output y_t ,but also all future hidden states $h_{t+1}, h_{t+2}, \ldots.$
- Thus, each parameter ($W^{(1)}$, W_h and $W^{(2)}$) affects the loss indirectly **through time**.
- So, during training, need to propagate the gradient back through all those time steps.
- To train the RNN, we unroll it over the sequence, treating it like a deep feed-forward network with T layers — one per time step — all sharing the same parameters. length of sequence
- Then, we apply standard backpropagation over this unrolled network.

Training RNNs

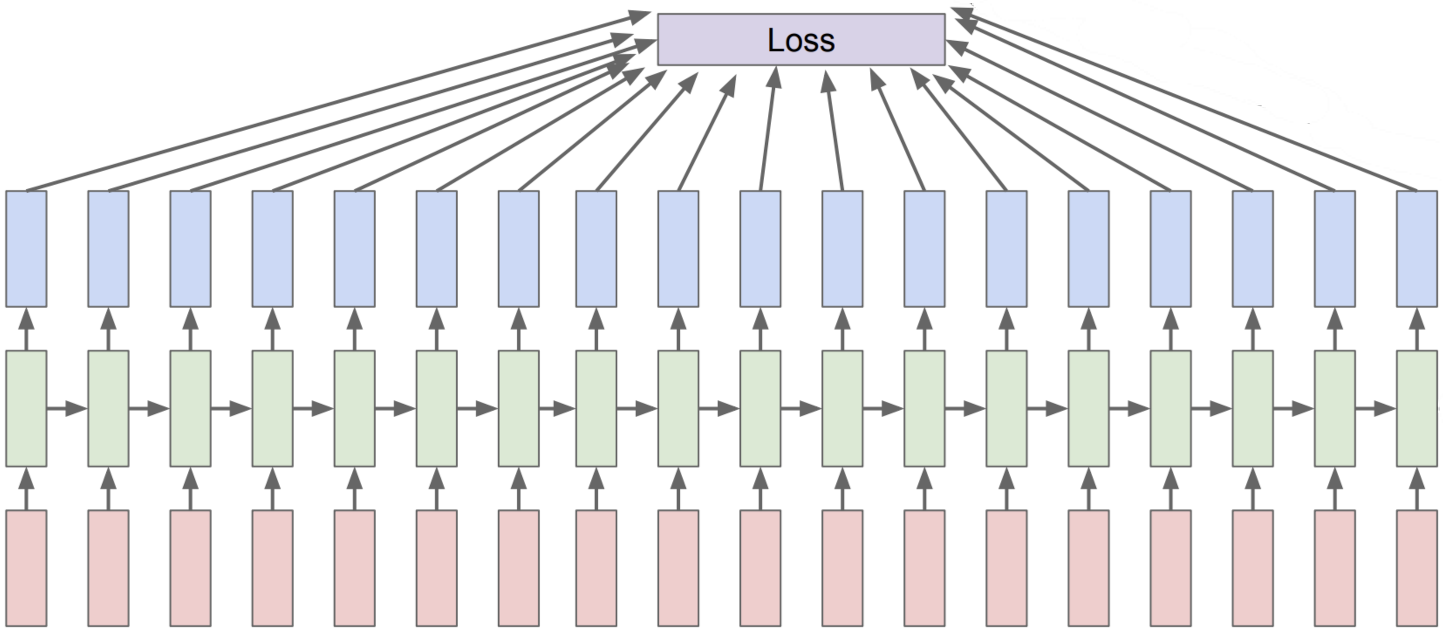


Forward pass to compute outputs and hidden representations

Backward pass to compute gradients

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

Training RNNs: Challenges



Forward pass to compute outputs and hidden representations

Backward pass to compute gradients

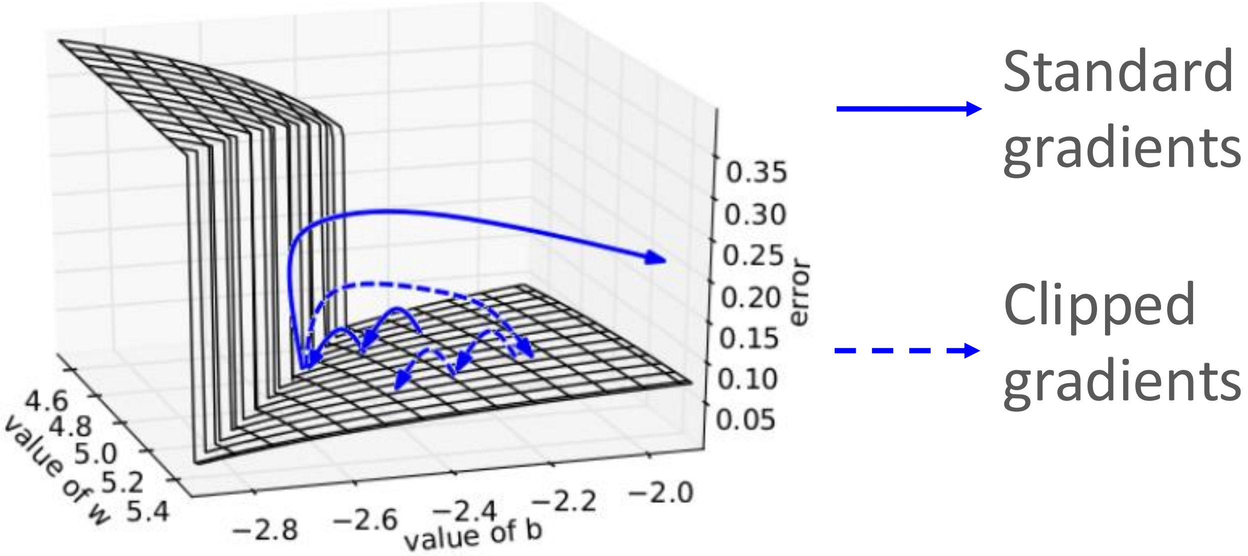
- Issue: as the sequence length grows, the gradient is more likely to explode or vanish

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

Gradient Clipping (Pascanu et al., 2013)

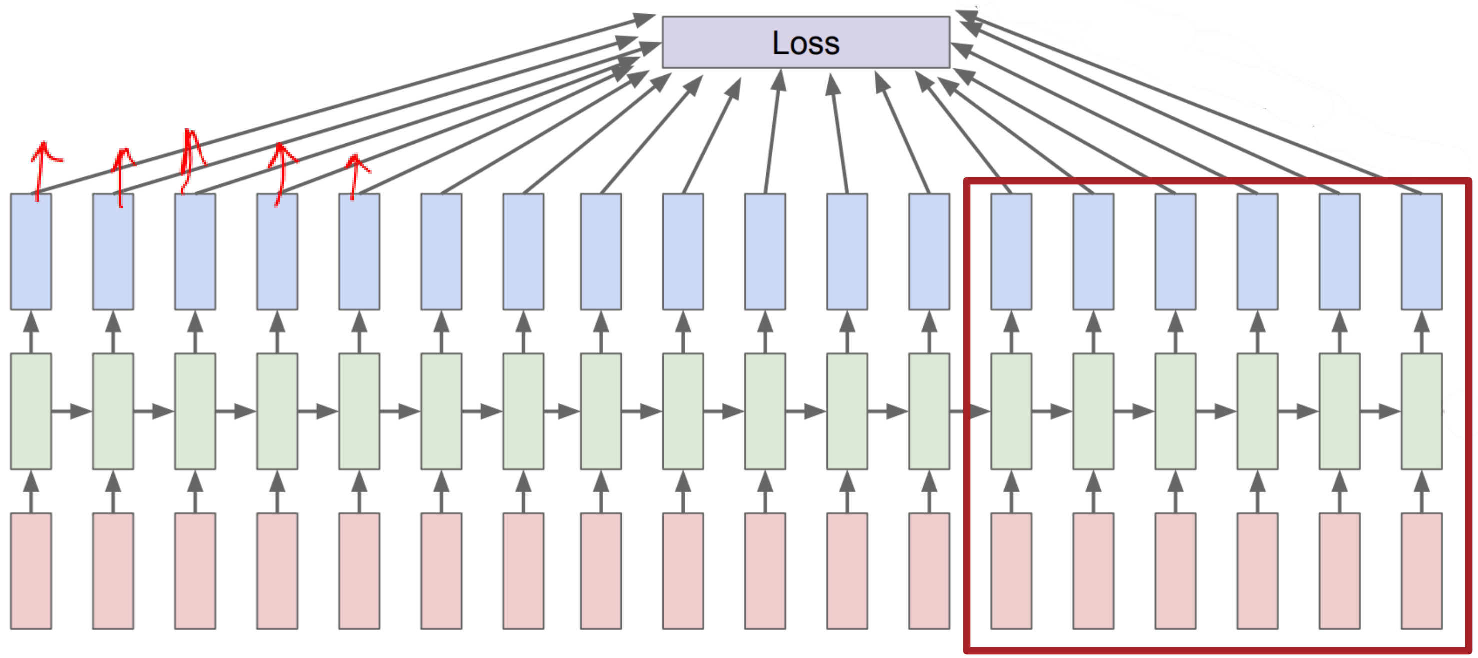
- Common strategy to deal with exploding gradients: if the magnitude of the gradient ever exceeds some threshold, simply scale it down to the threshold

$$G = \begin{cases} \nabla_W \ell^{(i)} & \text{if } \left\|\nabla_W \ell^{(i)}\right\|_2 \leq \tau \\ \left(\frac{\tau}{\left\|\nabla_W \ell^{(i)}\right\|_2}\right) \nabla_W \ell^{(i)} & \text{otherwise} \end{cases}$$



Source: https://arxiv.org/pdf/1211.5063.pdf

Truncated Backpropagation Through Time



Forward pass to compute outputs and hidden representations

Backward pass through a subsequence

- Idea: limit the number of time steps to backprop through

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 $\boldsymbol{h}_t$ 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 $\boldsymbol{h}_t$ but also maintains a separate internal *state*, C_t
- Gates are implemented as **sigmoids**: a value of 0 would be a fully *closed* gate and 1 would be fully *open*

$$I_t = \sigma\left(W_{ix}\boldsymbol{x}_t^{(i)} + W_{ih}\boldsymbol{h}_{t-1}\right)$$

$$O_t = \sigma\left(W_{ox}\boldsymbol{x}_t^{(i)} + W_{oh}\boldsymbol{h}_{t-1}\right)$$

$$F_t = \sigma\left(W_{fx}\boldsymbol{x}_t^{(i)} + W_{fh}\boldsymbol{h}_{t-1}\right)$$

$\tilde{C}_t$ candidate

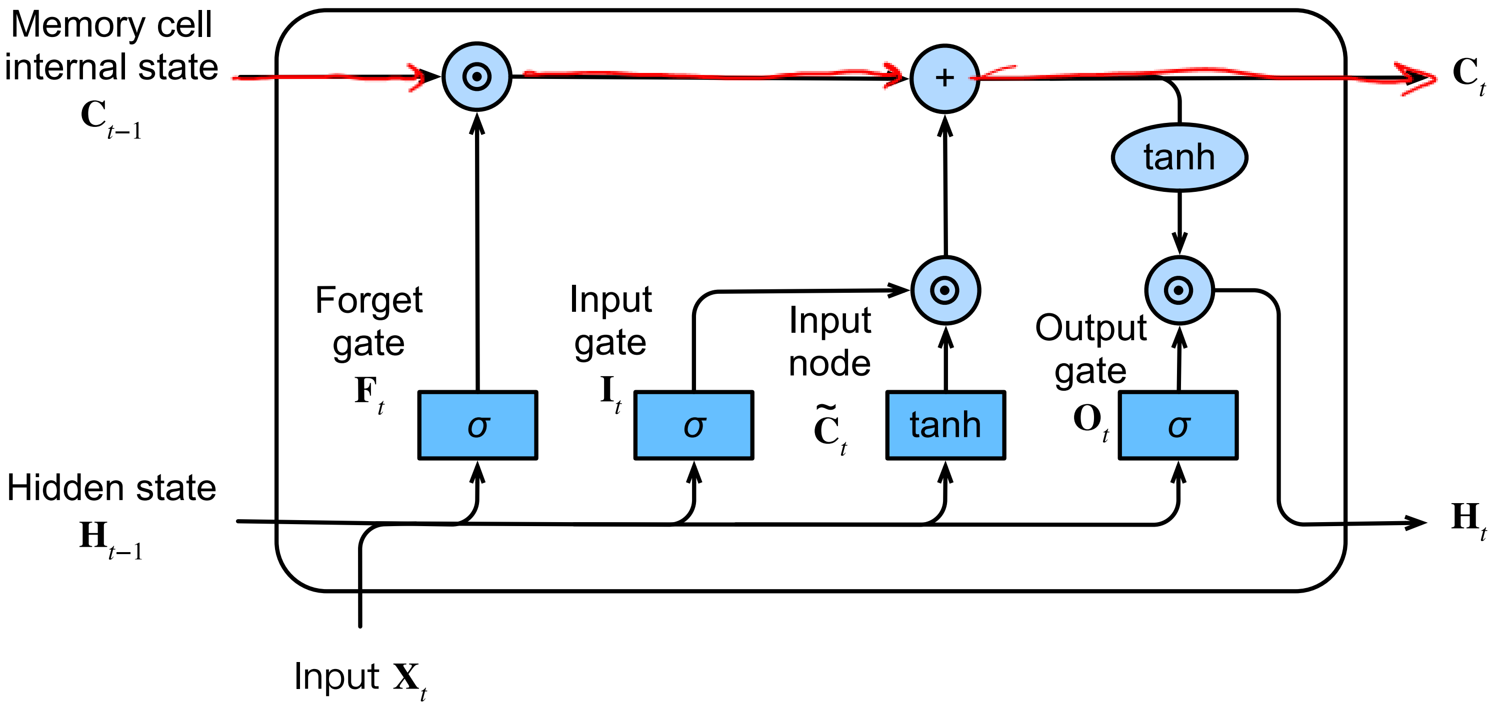
internal state $C_t = F_t \odot C_{t-1} + I_t \odot \theta\left(\overbrace{W^{(1)}\boldsymbol{x}_t^{(i)} + W_h\boldsymbol{h}_{t-1}}\right)$

candidate representation at time t

hidden state $\boldsymbol{h}_t = C_t \odot O_t$

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

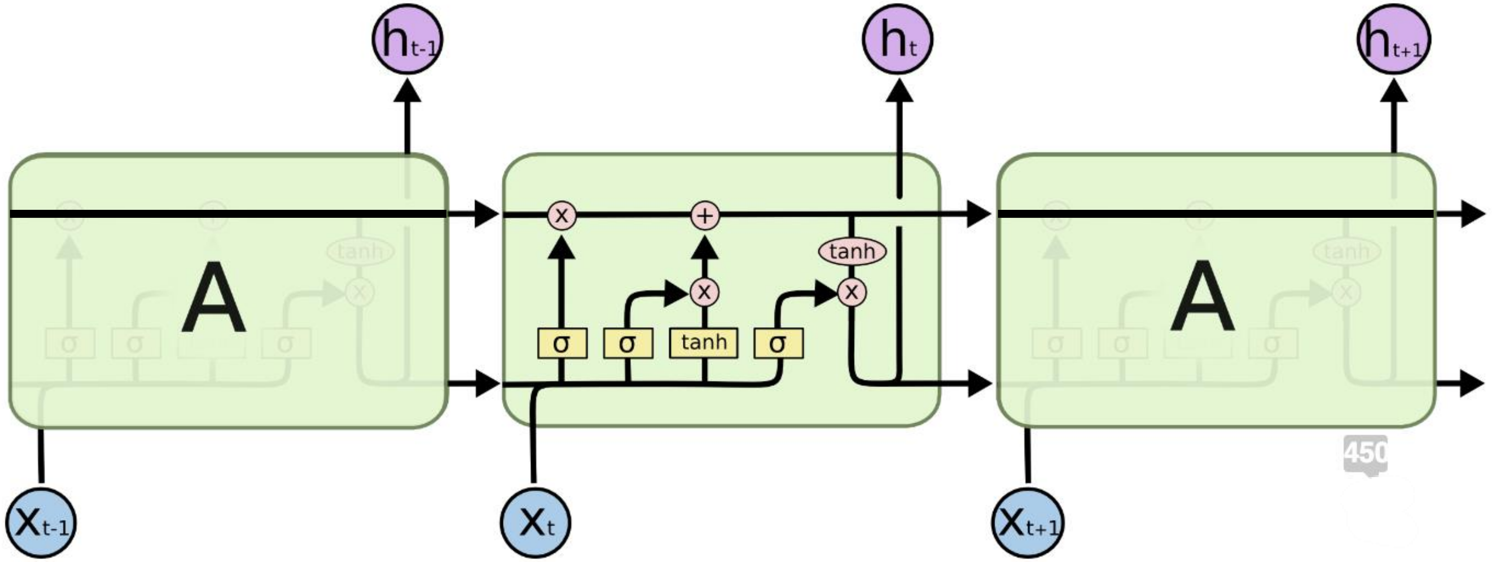
- LSTM networks address the vanishing gradient problem by replacing hidden layers with *memory cells*
- Each cell still computes a hidden representation H_t but also maintains a separate internal *state*, C_t



Source: https://d2l.ai/chapter_recurrent-modern/lstm.html

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 $\boldsymbol{h}_t$ 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/

LSTM and Gradients

- In a plain RNN, $\boldsymbol{h}_t = \left[1, \theta\left(W^{(1)}\boldsymbol{x}_t^{(i)} + W_h\boldsymbol{h}_{t-1}\right)\right]^T$ so the gradient w.r.t. an old state involves repeated multiplication by W_h and by θ. Over long time, that product tends to 0 (vanish) or ∞ (explode).
- Instead, the LSTM keeps a *side channel* (additive update + gate values near 1)

$$C_t = F_t \odot C_{t-1} + I_t \odot \theta\left(W^{(1)}\boldsymbol{x}_t^{(i)} + W_h\boldsymbol{h}_{t-1}\right)$$

- The old memory contributes **linearly,** thus gradients are less fragile.
- The backprop signal for something at time T reaching back to time $t \ll T$ can travel mostly along the **cell-state skip path**.
- If gates permit it (learned behavior), that path is close to identity — so the gradient can survive many steps.

Applications of LSTMs

AlphaGo

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 $\rightarrow$ 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.

Key Takeaways

- Recurrent neural networks use contextual information to reason about sequential data.
  - Can still be l... ...ation →
  - LSTMs allow ... to reach later timesteps without directly affecting intermediate hidden representations.

Transformers replaced LSTM in most large-scale NLP because attention scales better and captures long-range dependency without recurrence.

Language Models

1. Convert raw text into *embeddings*

$$\rightarrow \boldsymbol{x}^{(i)} = \left[\underbrace{\boldsymbol{x}_1^{(i)}}_{\text{words/token}}, \ldots, \boldsymbol{x}_{T_i}^{(i)}\right]$$

2. Learn or approximate a joint probability distribution over sequences

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

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

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

Language Models

1. Convert raw text into *embeddings*

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

2. Learn or approximate a joint probability distribution over sequences

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

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

$$P\left(\boldsymbol{x}^{(i)}\right) = \underbrace{P\left(\boldsymbol{x}_1^{(i)}\right)} * P\left(\boldsymbol{x}_2^{(i)} \mid \boldsymbol{x}_1^{(i)}\right) * P\left(x_3^{(i)} \mid x_1^{(i)}, x_1^{(i)}\right) \cdots * P\left(\boldsymbol{x}_{T_i}^{(i)} \mid \underbrace{\boldsymbol{x}_{T_i-1}^{(i)}, \ldots, \boldsymbol{x}_1^{(i)}}\right)$$

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~~ Just throw an RNN at it!

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

RNN 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

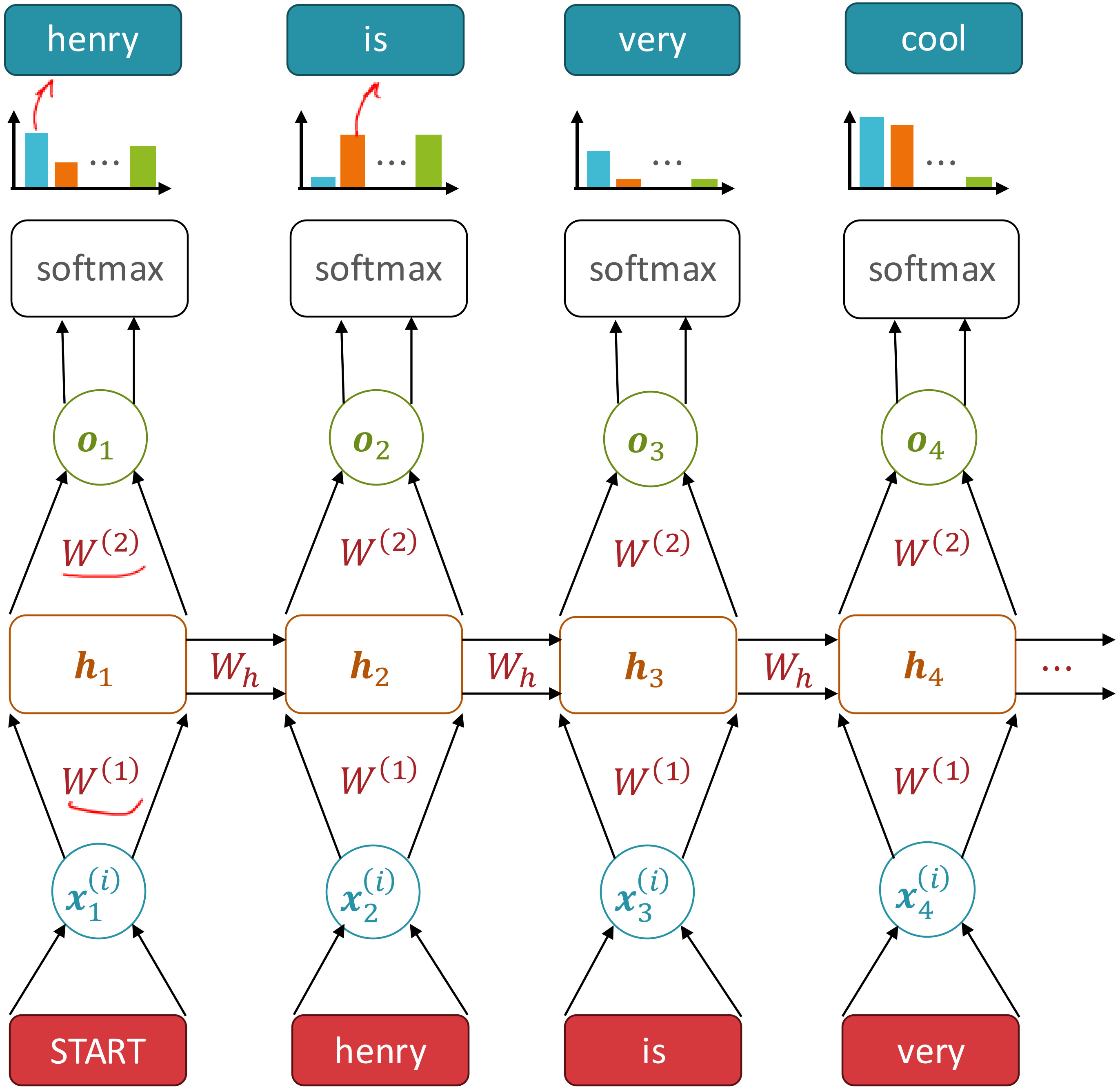
$$P\left(\boldsymbol{x}^{(i)}\right) = P\left(\boldsymbol{x}_1^{(i)}, \dots, \boldsymbol{x}_{T_i}^{(i)}\right)$$

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

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

RNN Language Models: Training

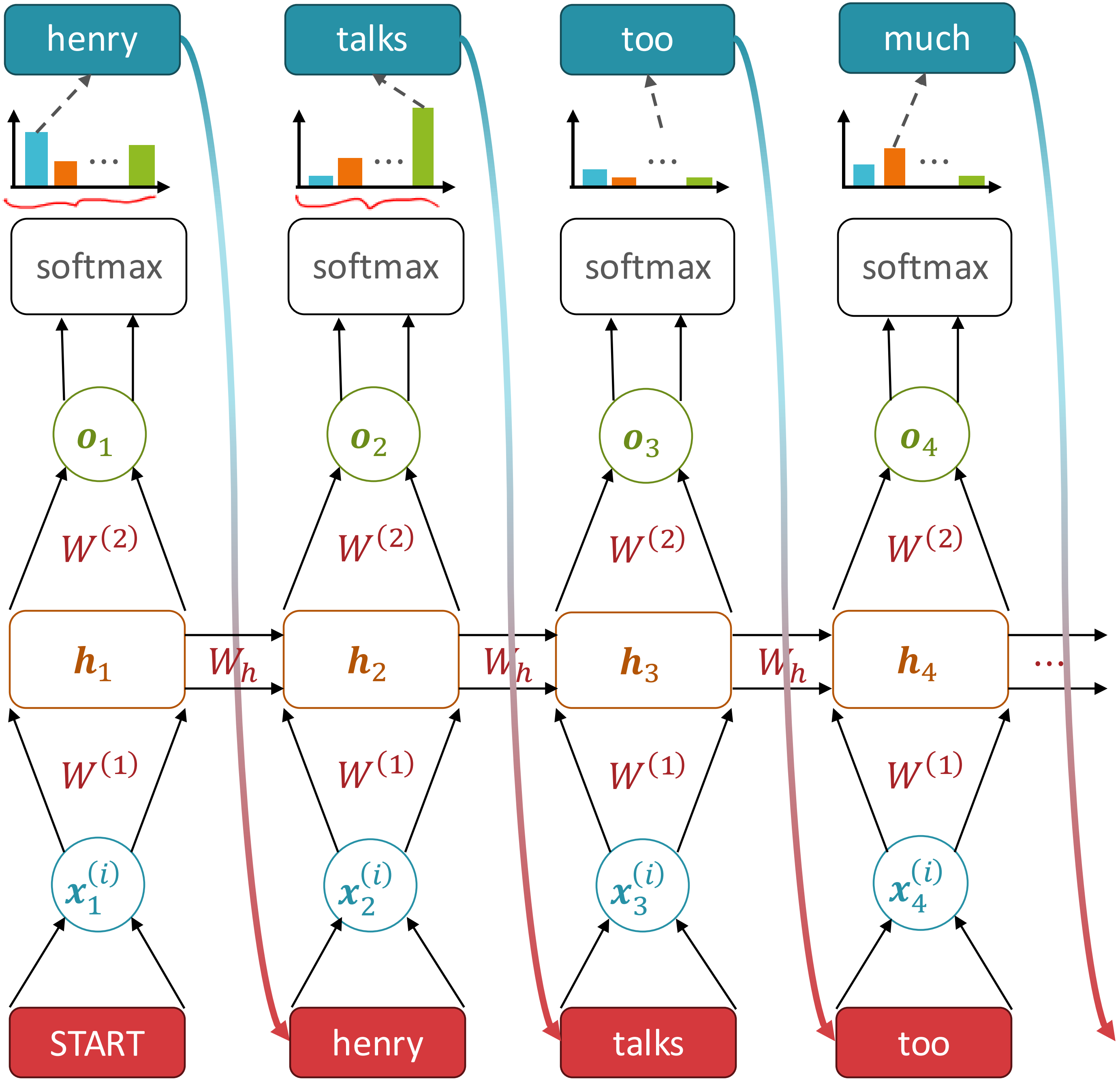
Target sequence (try to predict the next word)



Input sequence

RNN Language Models: Sampling

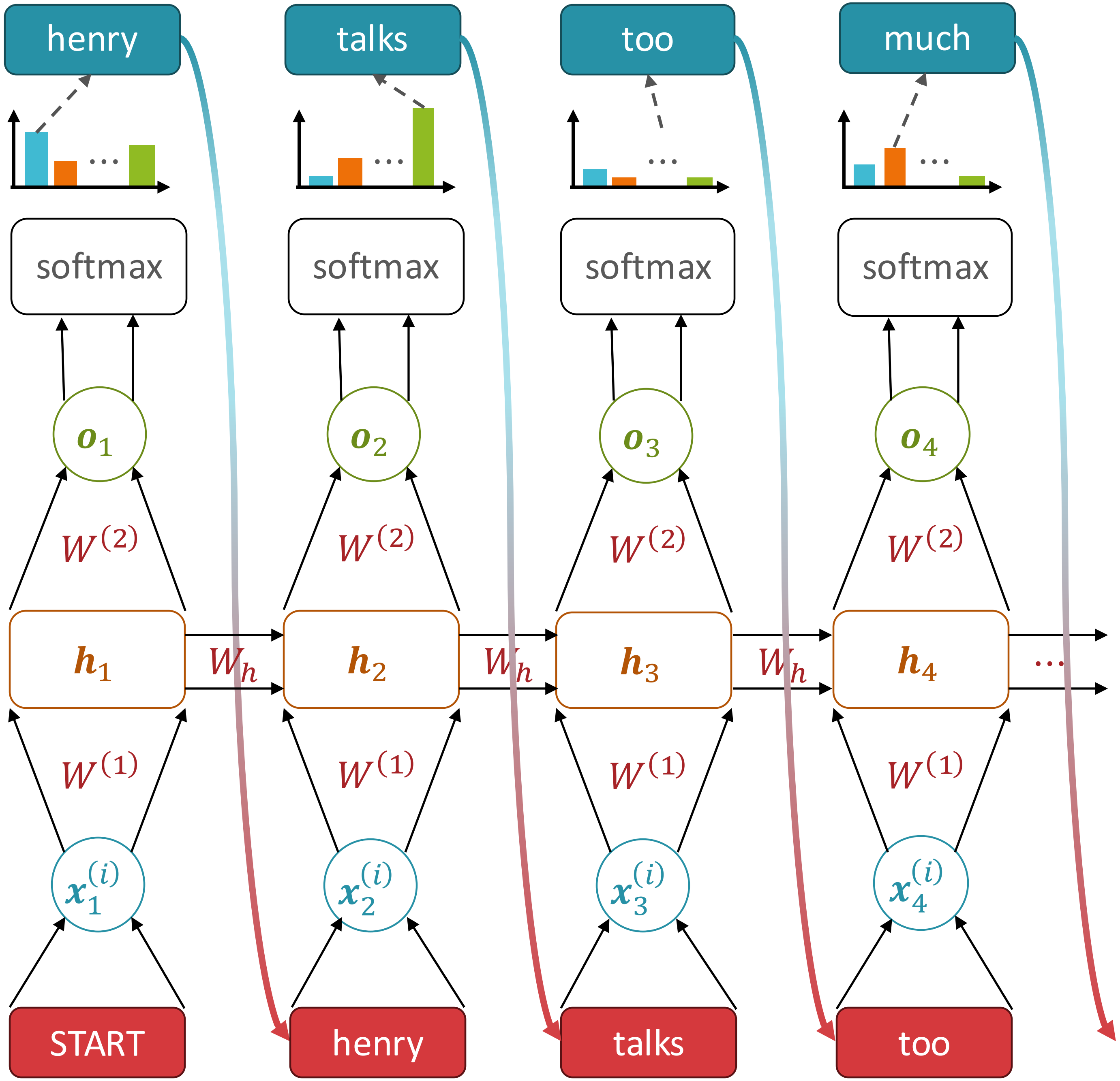
Generated sequence (use each token as the input to the next timestep)



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

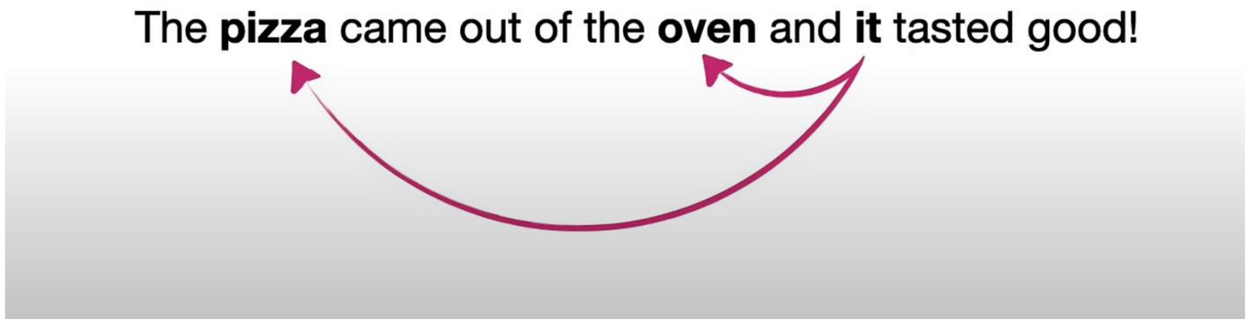
- Pros:
  - Can handle arbitrary sequence lengths without having to increase model size (i.e., # of learnable parameters)
  - Trainable via backpropagation
- Cons
  - Vanishing/exploding gradients
    - Can be addressed by LSTMs
  - Does not consider information from later timesteps
    - Can be addressed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

Transformers

- Instead of reading text word-by-word like an RNN/LSTM, a transformer looks at the entire sentence at once and asks:

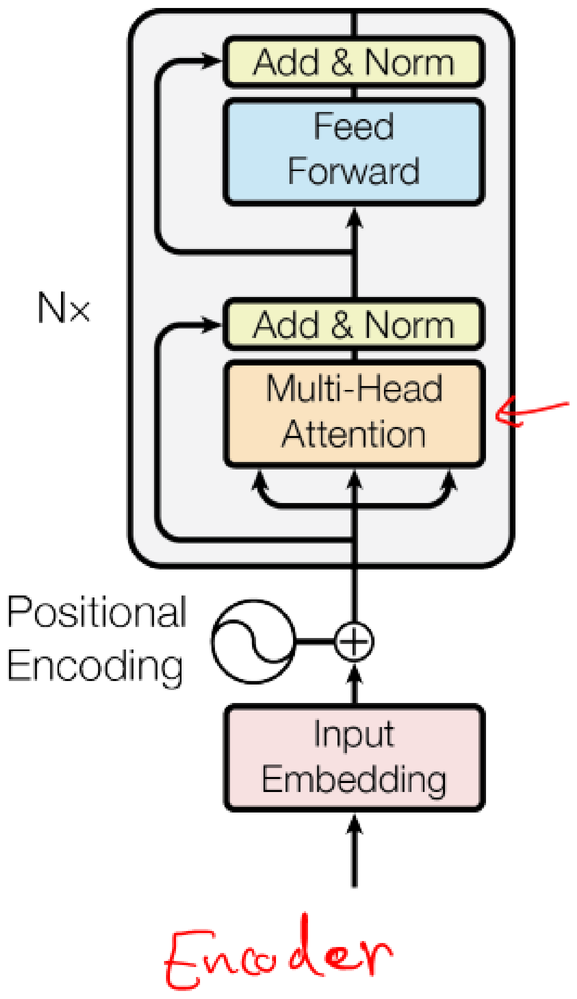
  “For this word, which other words in the sentence matter right now, and by how much?”

- This mechanism is called **self-attention:** every word computes links to other words and assigns strengths to them.



Transformers

1. **Embed words:** Words are turned into vectors (points in a space that roughly encode semantic meaning).
2. **Attention Scoring:** Each word looks at all other words and asks: “How relevant are you to me given the task at hand?”
3. **Blend information:** Each word builds a new representation by taking a weighted mix of the other words, weighted by their scores.
4. **Repeat:** Stack that many times--layers repeat the same pattern, so deeper layers see richer relational structure/representations.

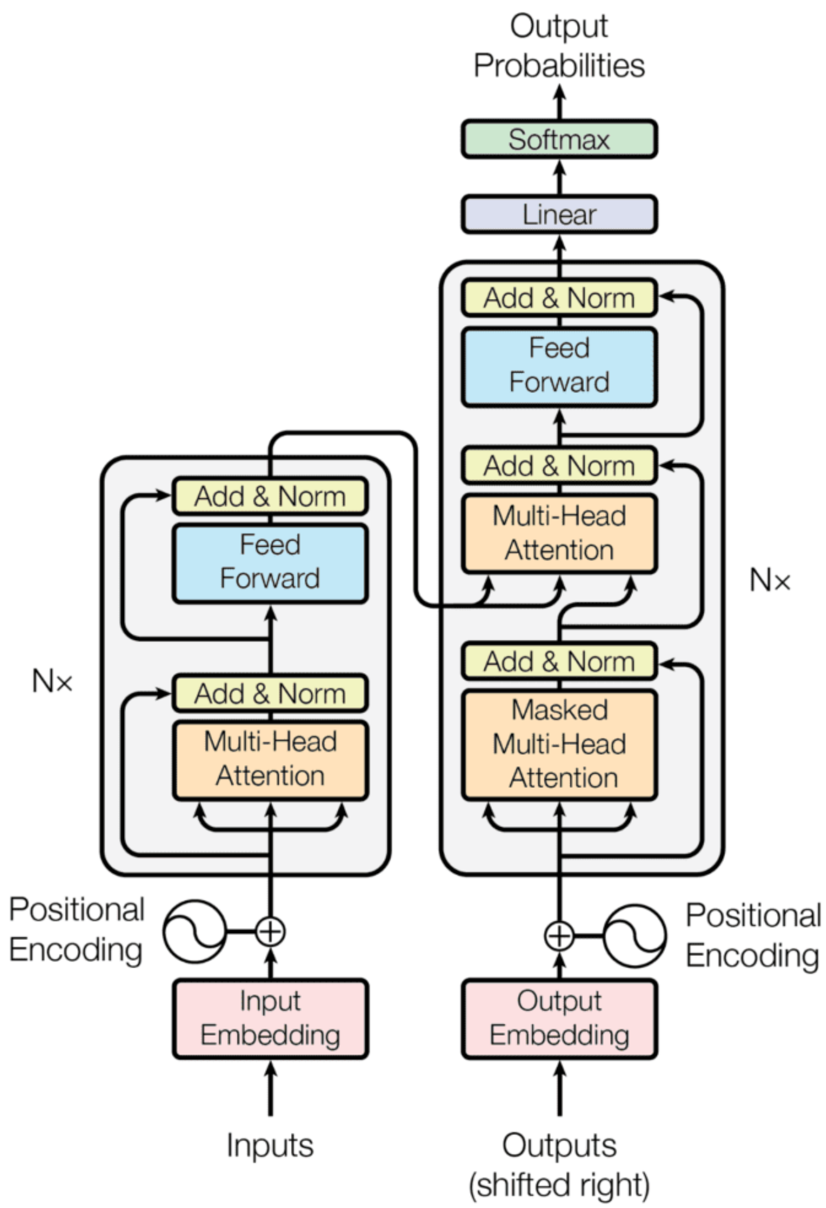


Source: https://arxiv.org/pdf/1706.03762.pdf

Encoder vs. Decoder

5. Generate/understand text: A decoder side produces next tokens by repeatedly applying the same attention process.

- **Encoder:** map an input sequence to representations, fed into a decoder.
- **Decoder:** receives the output of the encoder and the decoder output at the previous time step to generate new output.



Decoder

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

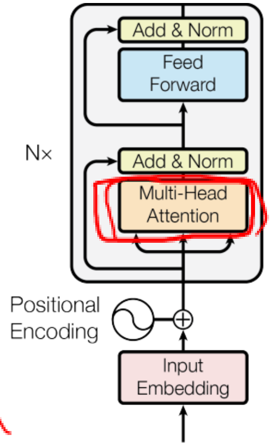
Transformers vs. RNNs

RNNs	Transformers
RNNs only get **cumulative context** from the past; they do not have easy access to distant words unless encoded in hidden state.	In a transformer, at each layer, every token sees every other token (**global context**).
RNNs can forget over long distances even with LSTMs.	Transformers use **self-attention** to directly connect any word to any other word
RNNs/LSTMs read sequences *step-by-step* ($x_1 \rightarrow x_2 \rightarrow x_3$ …), so they can't parallelize across time.	Transformers consume the whole sequence at once and compute attention in parallel → efficiency.

The Attention Mechanism

Attention head

Analogy: attention as soft lookup in a dictionary



1. The token x' broadcasts its **query** $q = w_Q^T x'$
2. Every other token x_t offers up its **key** $k_t = \boxed{W_K} x_t$
3. The model computes a *similarity* between the query and keys--scoring how relevant each token is to the query.

attention $$s_t(x', x_t) = \frac{k_t^T q}{\sqrt{\text{length}(q)}}$$

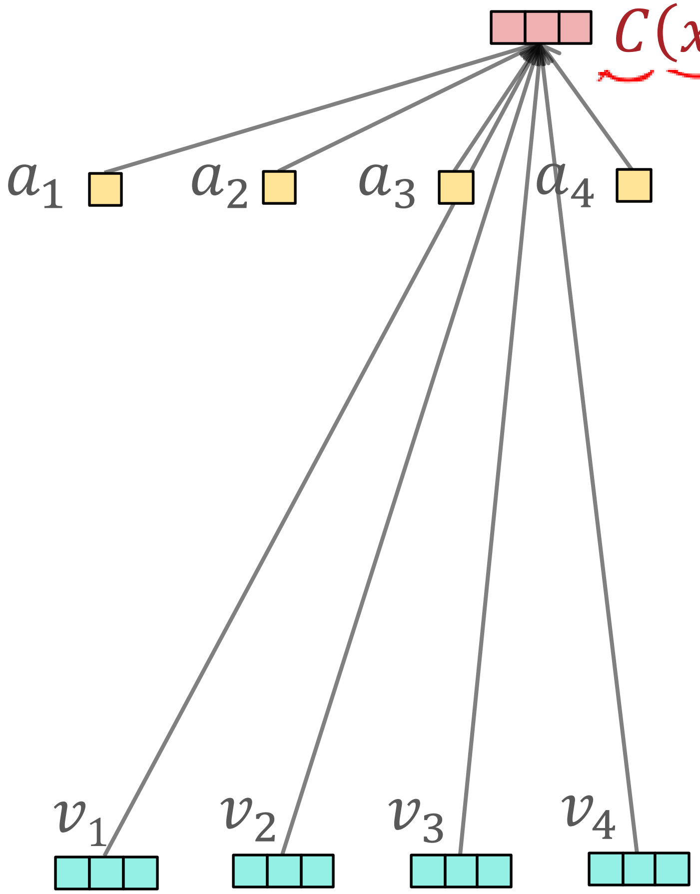
4. Those *scores* are turned into weights: $\text{softmax}(s(x', x_t))$
5. The model returns a weighted sum of the **values** as the contextualized representation:

$$\sum_t \text{softmax}(s(x', x_t)) v(x_t)$$

Attention

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

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

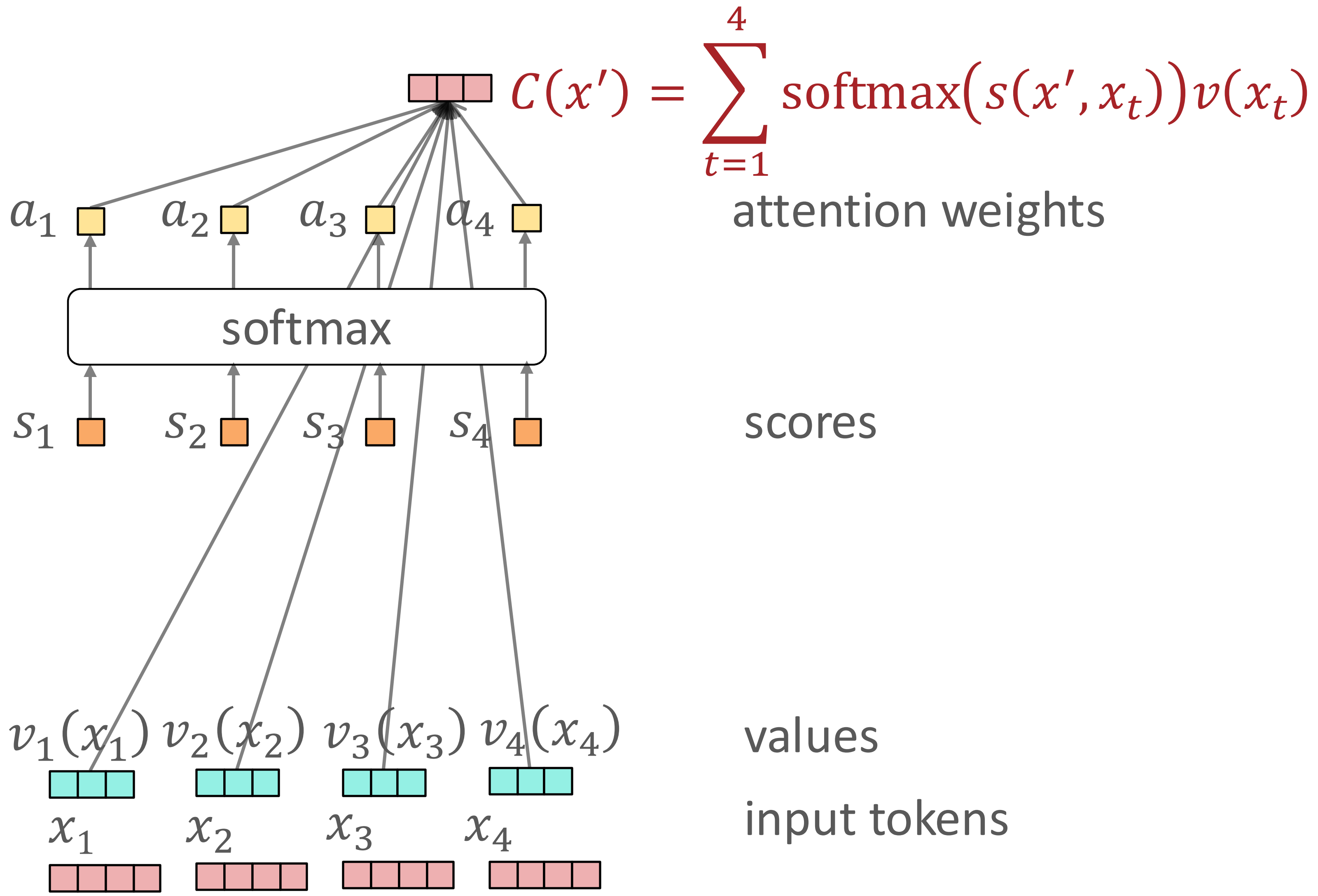


attention weights

values

Attention

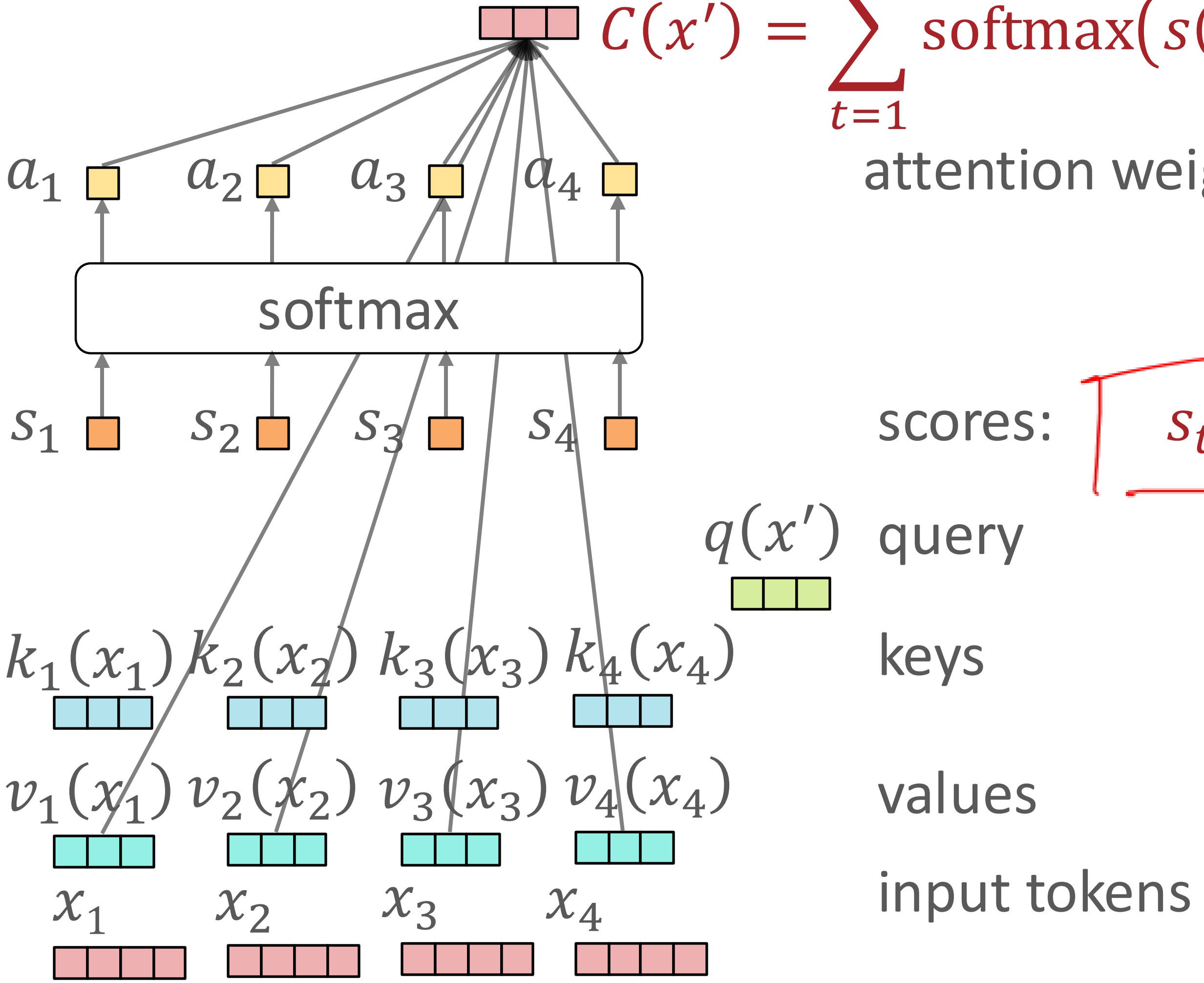
- Approach: compute a representation of the input sequence for each token x'



Scaled Dot-product Attention

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

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

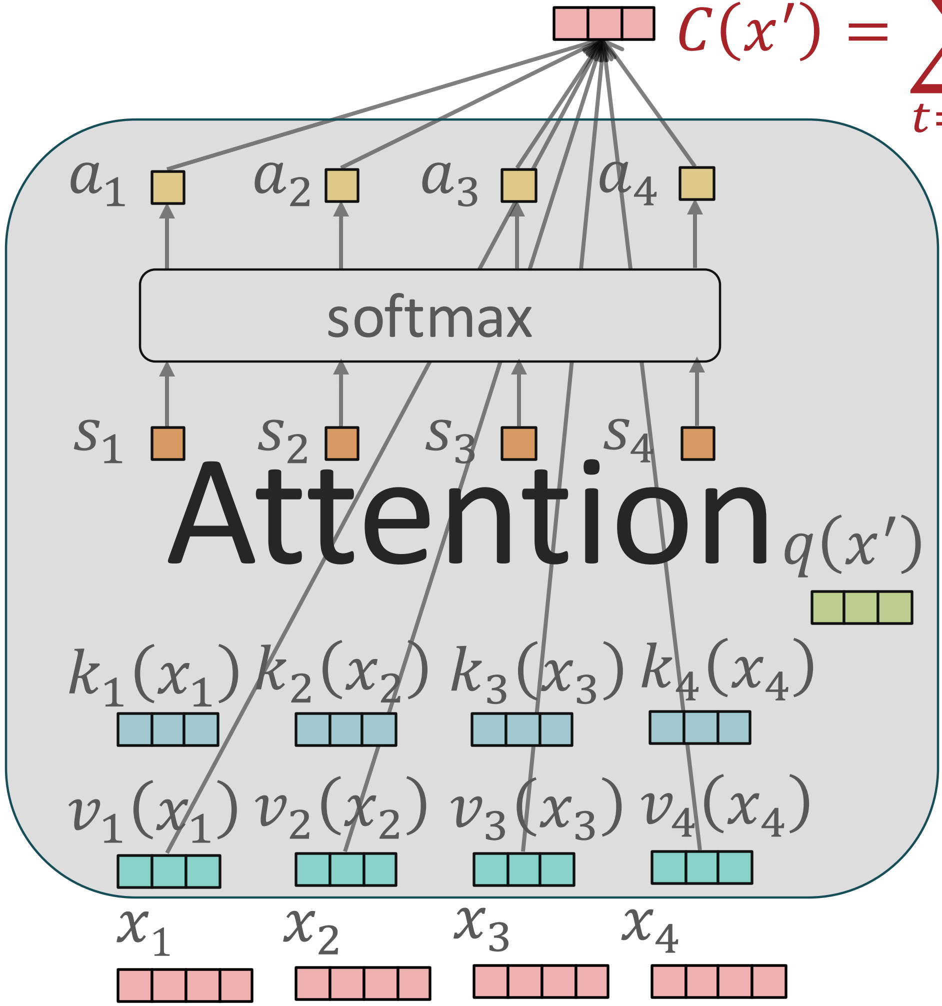


$$\boxed{s_t = \frac{k_t^T q}{\sqrt{\text{length}(q)}}}$$

Scaled Dot-product Attention

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

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



attention weights

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

query: $q = w_Q^T x'$

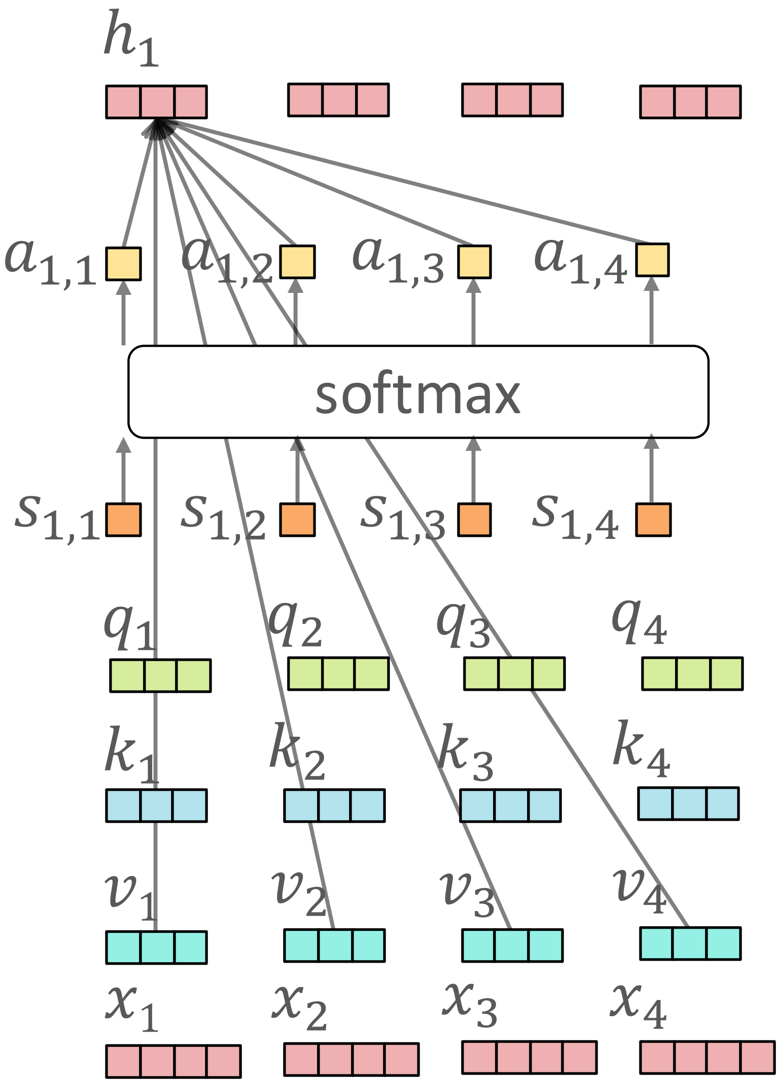
keys: $k_t = W_K x_t$

values: $v_t = W_V x_t$

input tokens

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: $q_t = W_Q x_t$

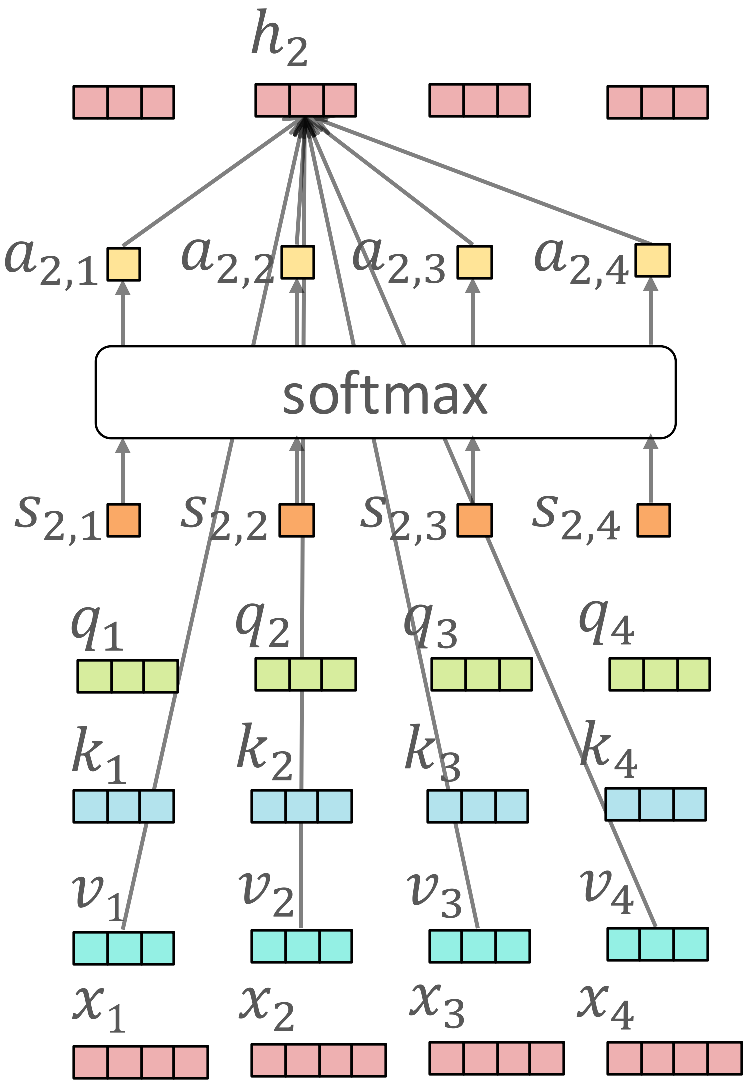
keys: $k_t = W_K x_t$

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



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

attention weights

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

queries: $q_t = W_Q x_t$

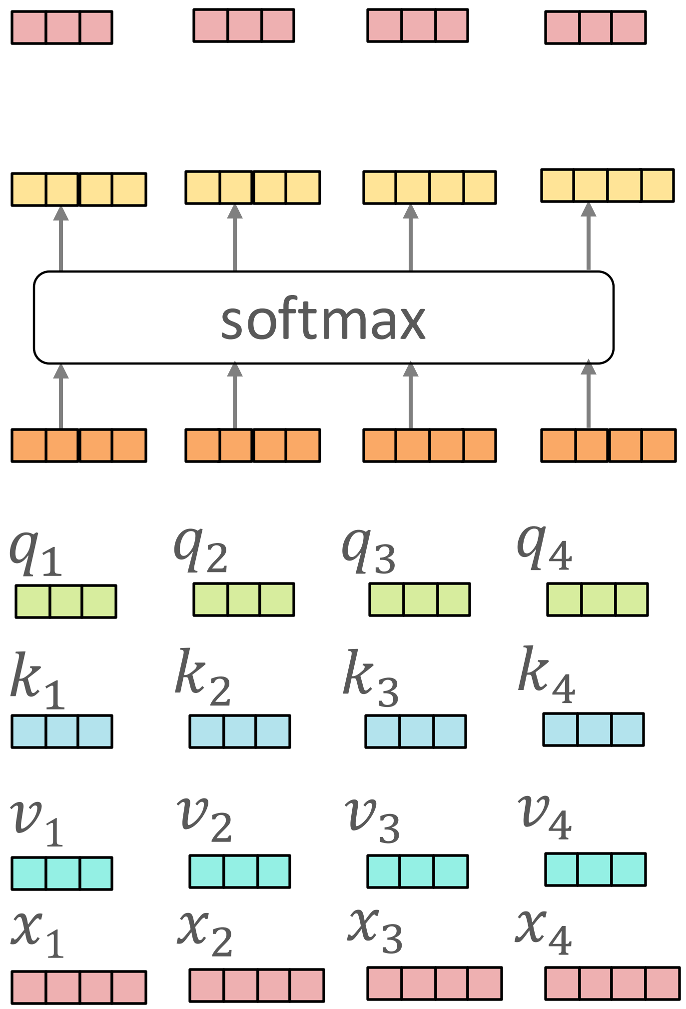
keys: $k_t = W_K x_t$

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

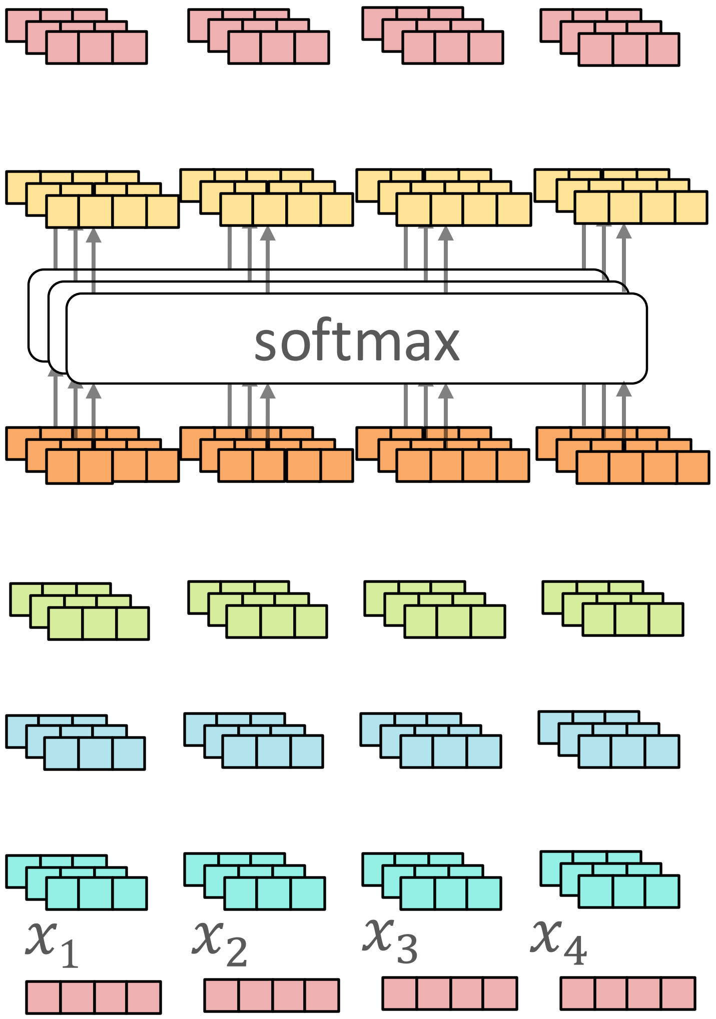
keys: $K = XW_K \in \mathbb{R}^{N \times d_k}$

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

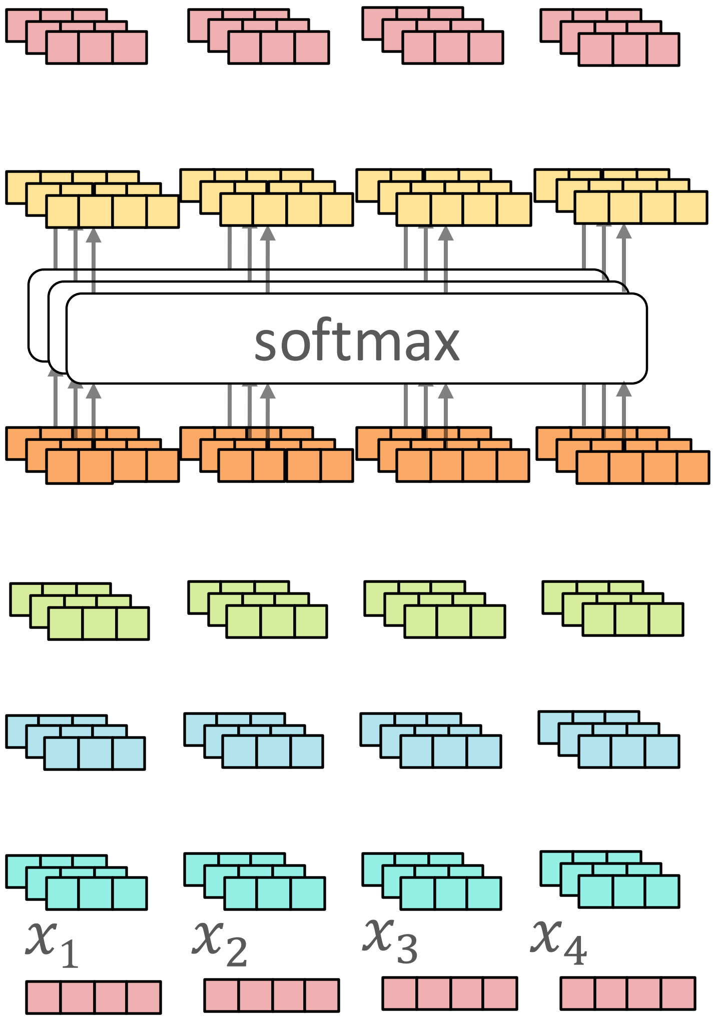
values: $V^{(h)} = XW_V^{(h)}$

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

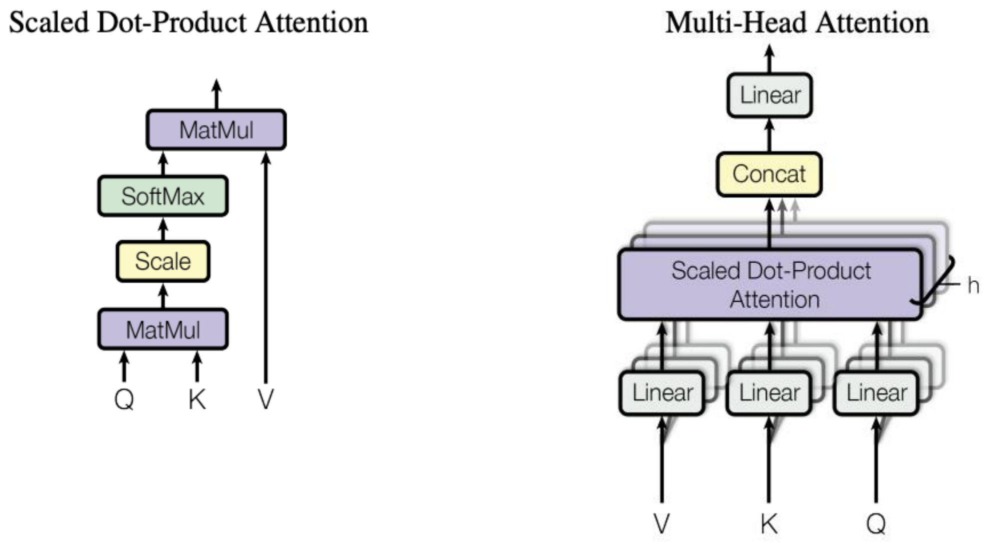
keys: $K^{(h)} = XW_K^{(h)}$

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!



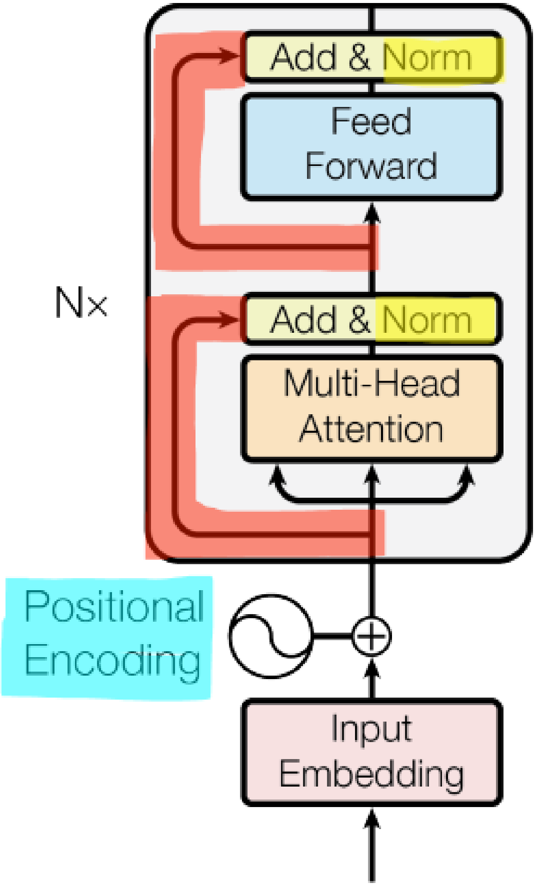
- The outputs from all the attention heads are concatenated together to get the final representation

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

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

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

Transformers

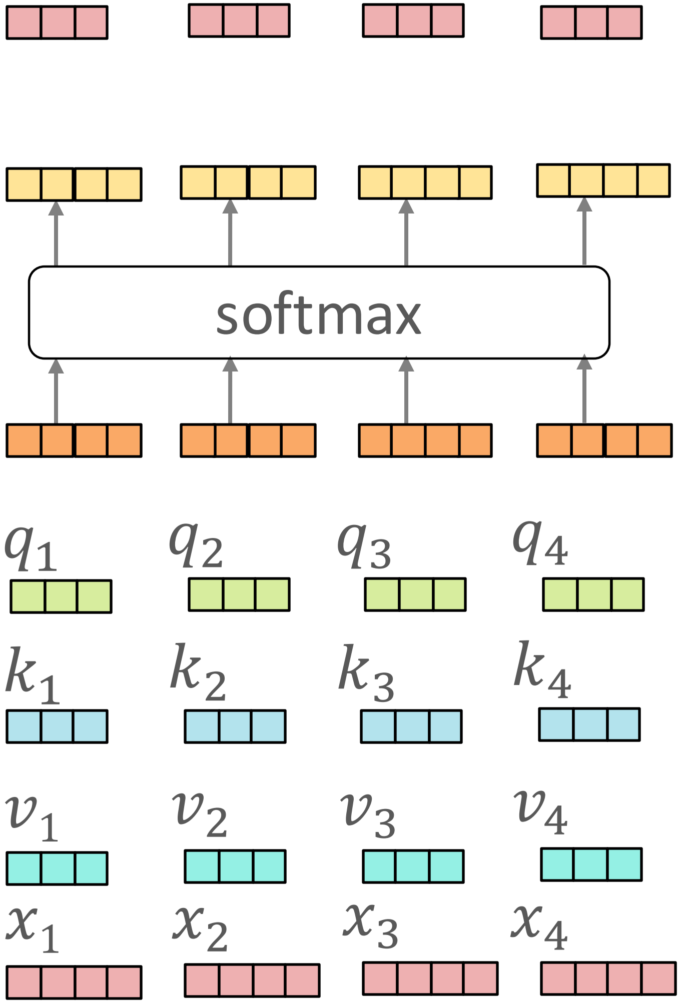


- In addition to multi-head attention, transformer architectures use
    1. Positional encodings
    2. Layer normalization
    3. Residual connections
    4. A fully-connected feed-forward network

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

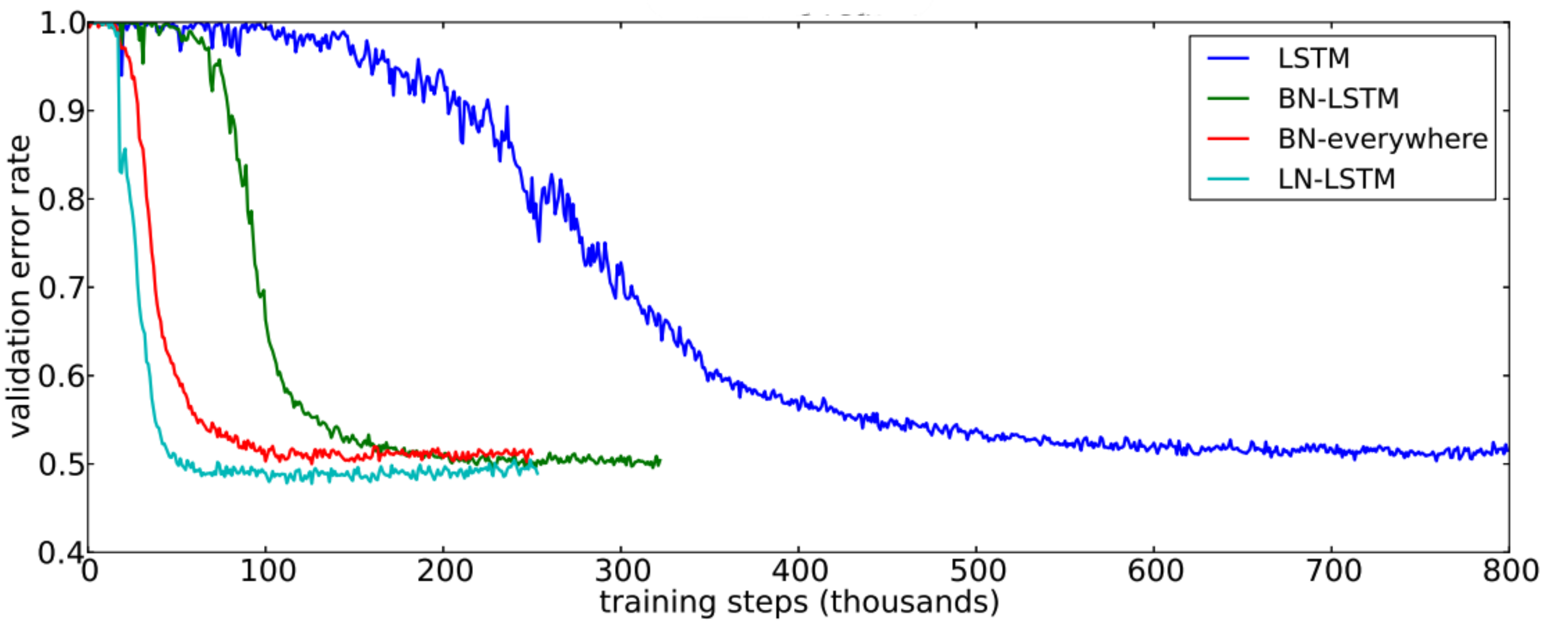
Layer Normalization

- 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

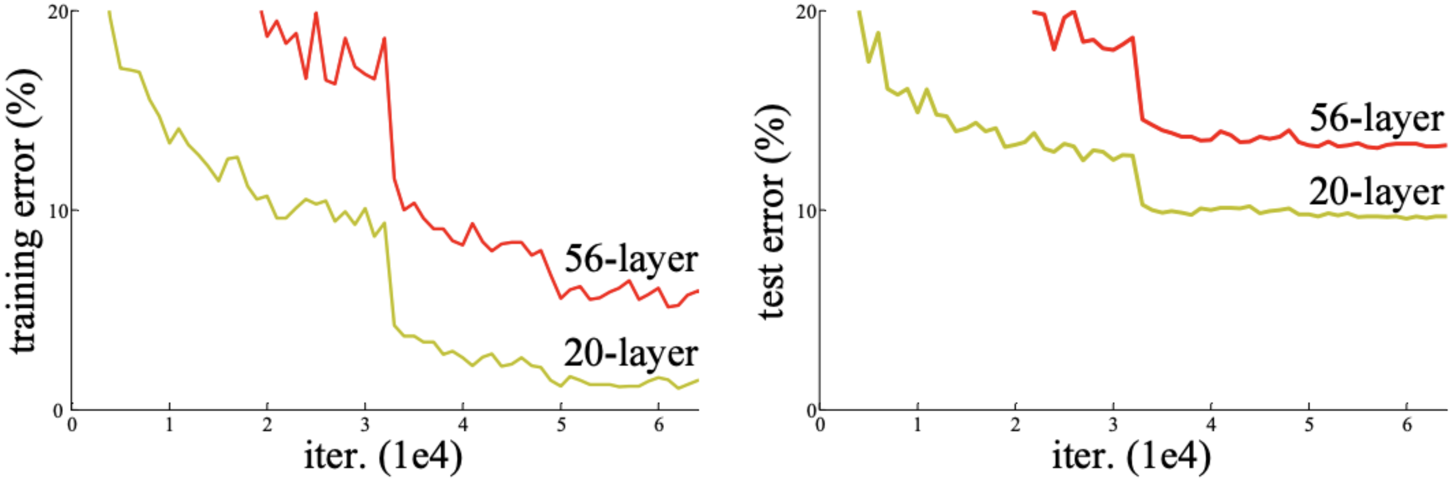
$$H' = \gamma\left(\frac{H - \mu}{\sigma}\right) + \beta$$

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

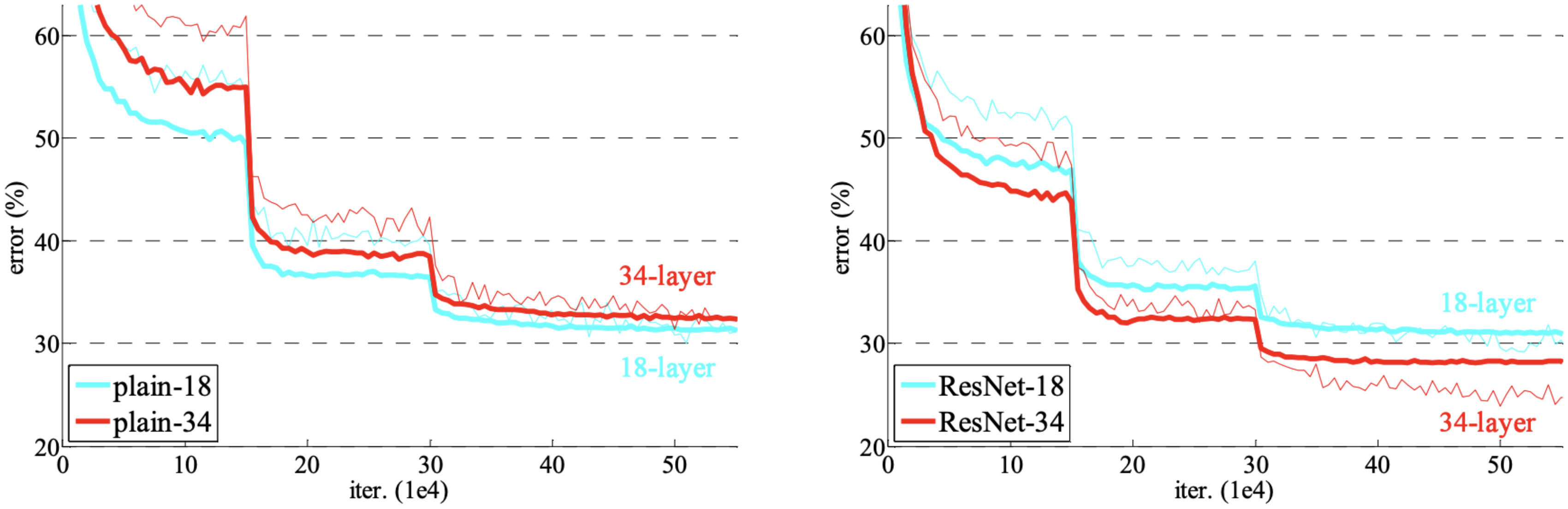
$$H' = H\left(x^{(i)}\right) + x^{(i)}$$

- 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