Deep Learning for **Natural Language Processing**

Antoine Bosselut

Natural Language Processing

Enabling Human-Machine Collaboration

Search Engines

Dialogue Agents

Text Generation

Accelerating Human-Human Communication

Machine Translation

Text Summarization

Information Extraction

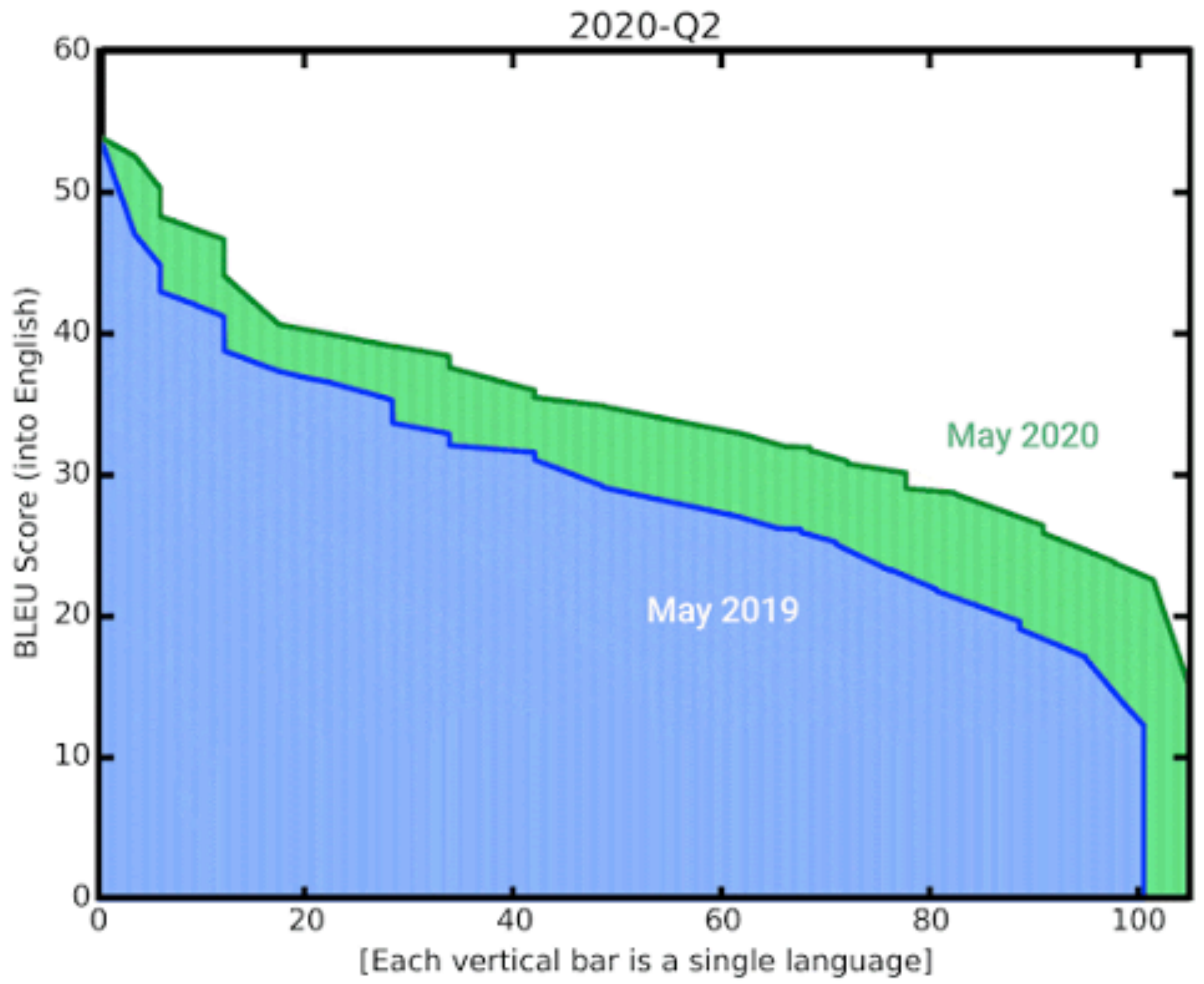
Mining Human Insights

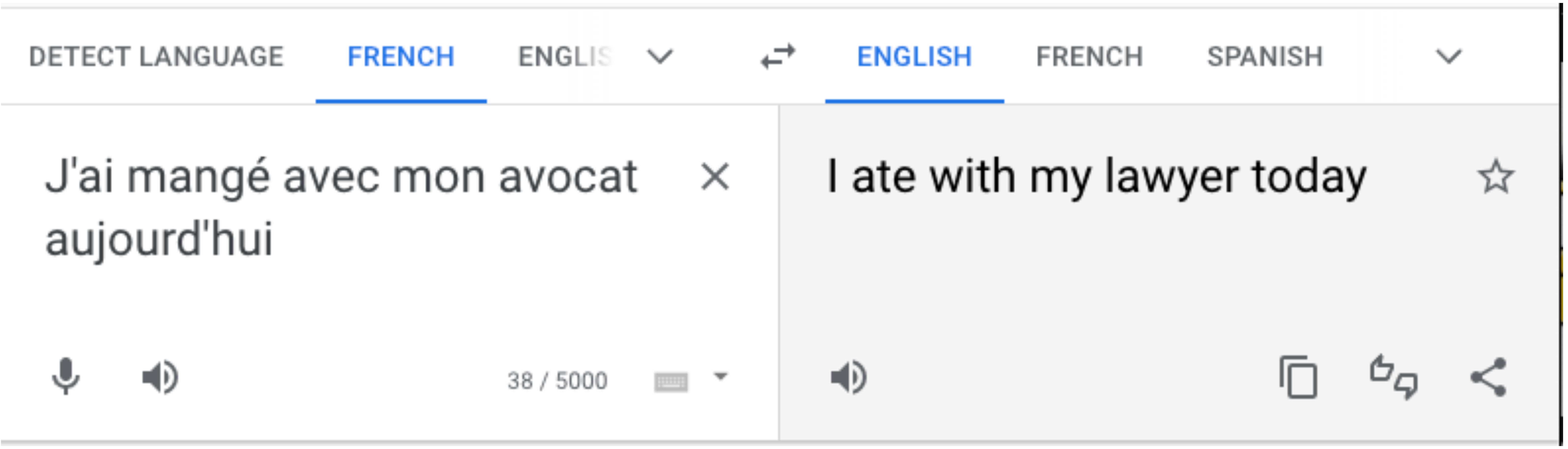
Sentiment Analysis

Motivation Analysis

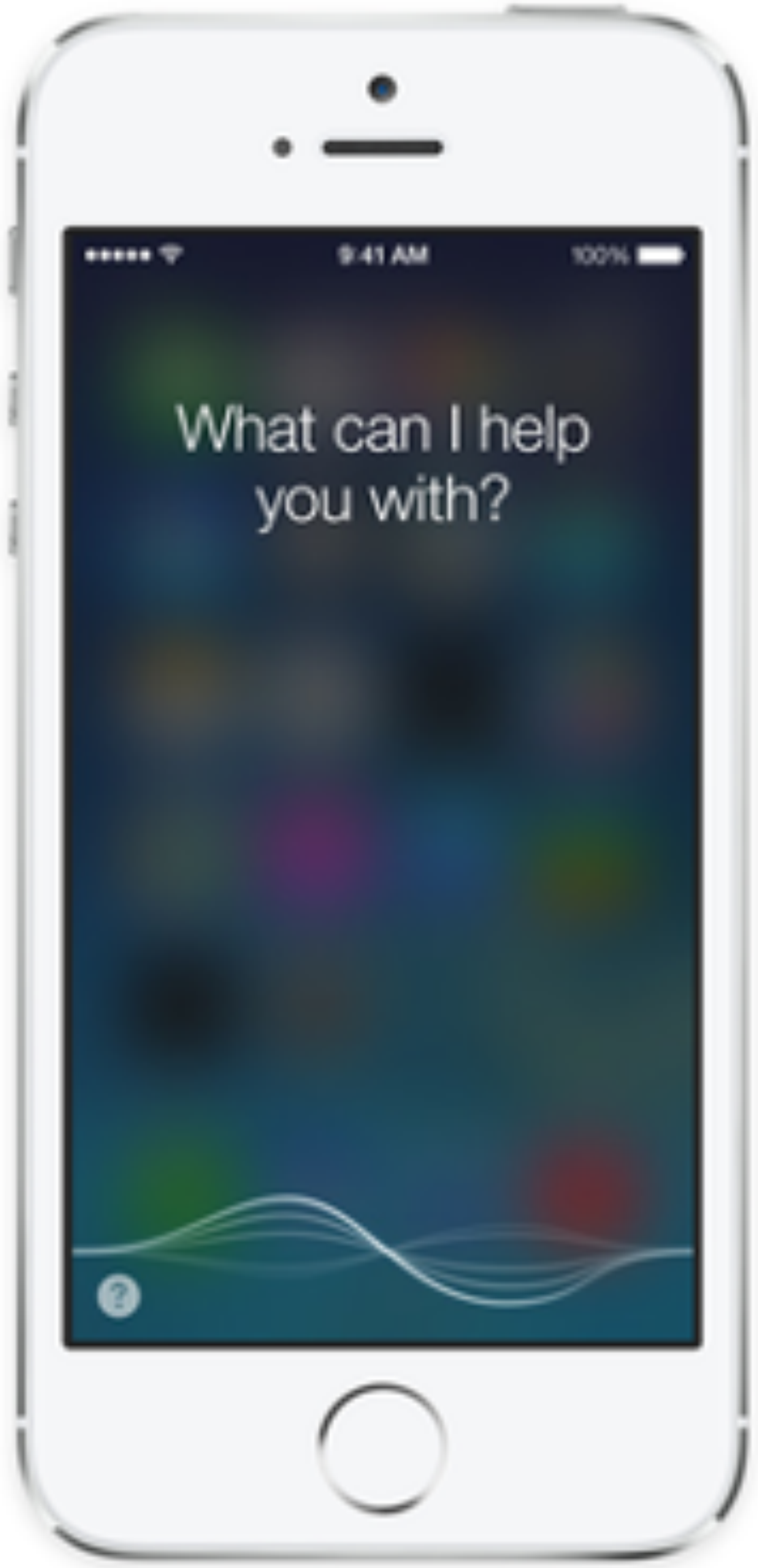
Emotion Detection

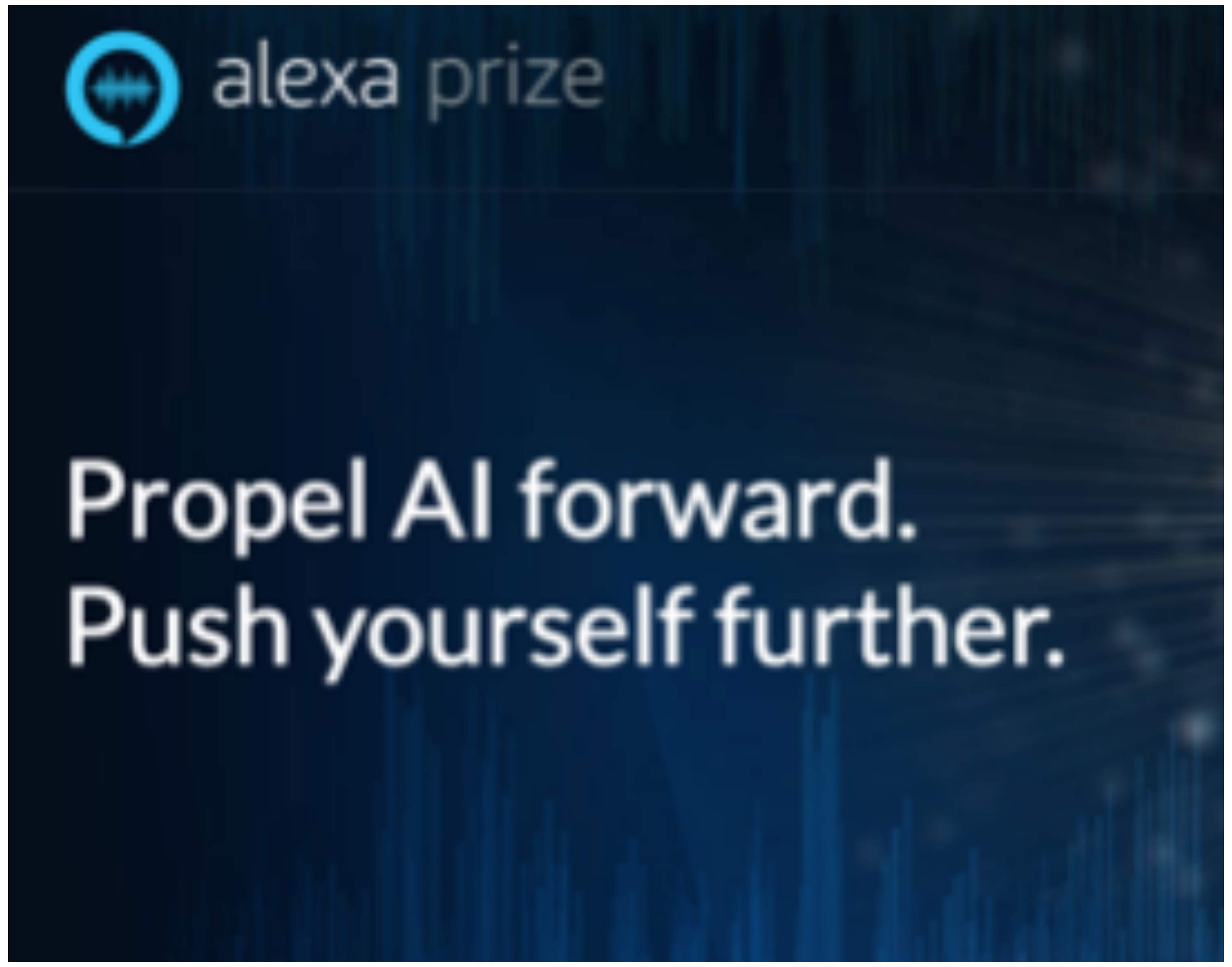
Machine Translation



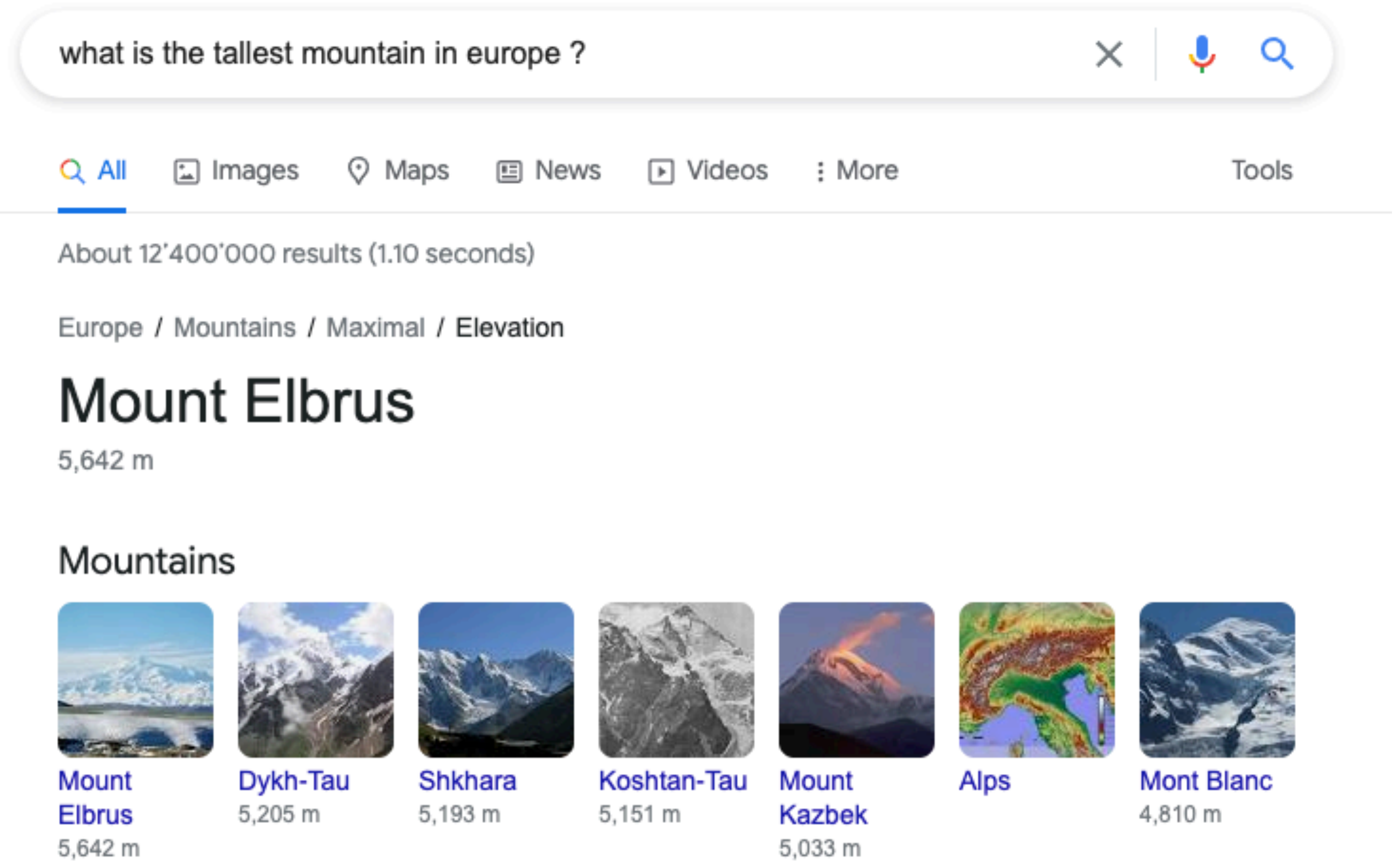


Conversational Systems





Question Answering



Text Generation

AN

Can you give me a two-day itinerary for a weekend trip to Paris ? Please don't include more than three activities per day.

Day 1:

1. Visit the Eiffel Tower - Start your day with the iconic symbol of Paris, the Eiffel Tower. Take a stroll around the Champ de Mars and enjoy the views from the tower.
2. Explore the Louvre Museum - After lunch, head to the Louvre Museum and admire the world-famous artwork such as the Mona Lisa and the Venus de Milo.
3. Stroll along the Seine River - In the evening, take a romantic stroll along the Seine River and enjoy the Parisian atmosphere.

Day 2:

1. Visit Notre-Dame Cathedral - Start your day by visiting one of the most famous Gothic cathedrals in the world, Notre-Dame Cathedral.
2. Explore the Marais neighborhood - After lunch, explore the Marais neighborhood, known for its historical architecture, trendy boutiques, and restaurants.
3. Enjoy a Cabaret Show - In the evening, enjoy a cabaret show at the famous Parisian venue, the Moulin Rouge, and end your trip with a memorable experience.

Next few weeks!

- **Today:** Deep Learning for Natural Language Processing
- **In two weeks:** Neural Text Generation
- **Final week:** Modern NLP & Ethical Implementation of NLP

Today's Outline

- **Introduction**
- **Section 1 -** Neural NLP & Word Embeddings
- **Section 2 -** Recurrent Neural Networks for Sequence Modeling
- **Section 3 -** Attentive Neural Modeling with Transformers
- **Exercise Session:** Attention in Transformer Language Models

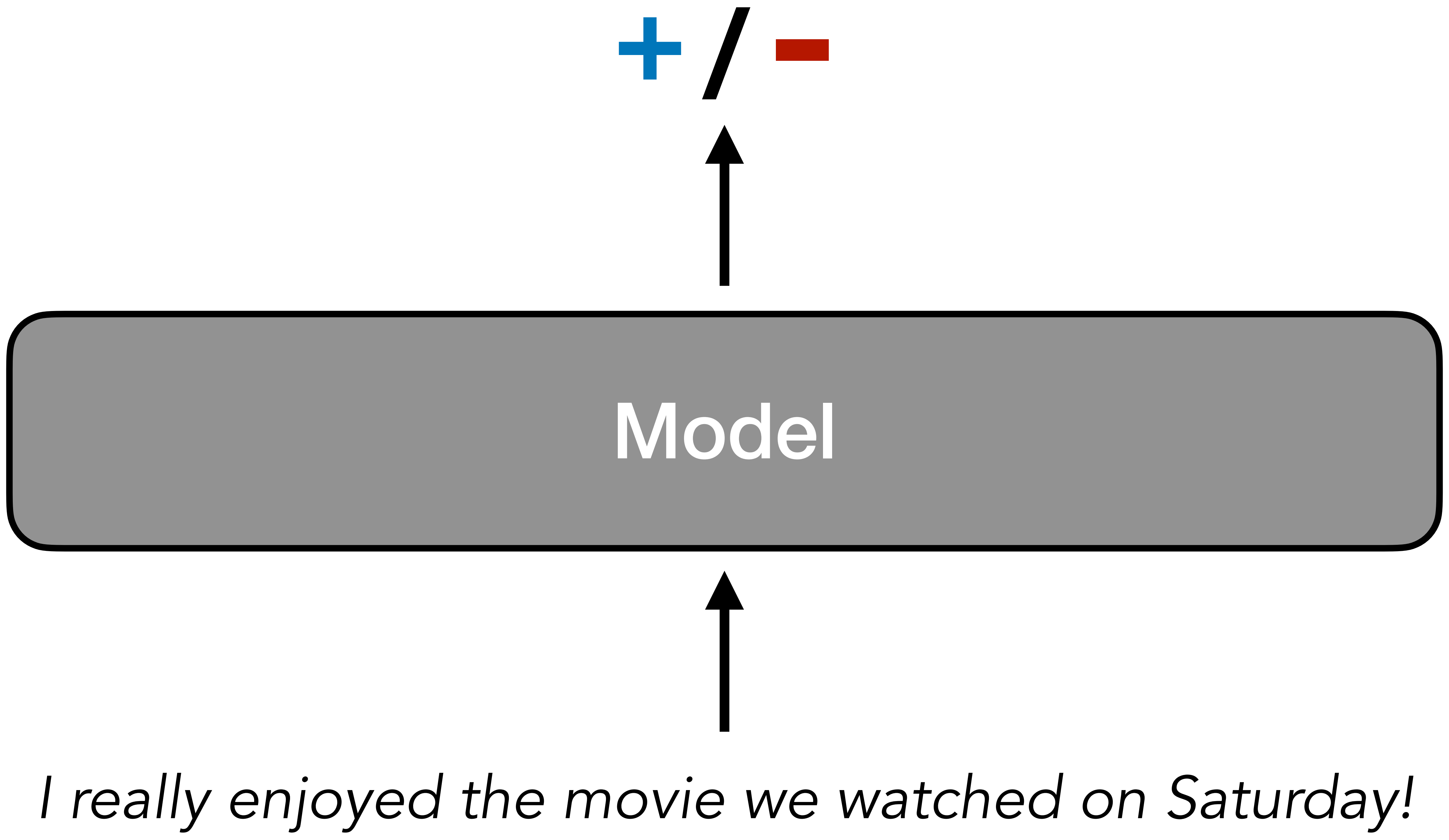
Part 1: Neural Embeddings

Section Outline

- **New:** Building our first neural classifier
- **Review**: sparse word vector representations
- **New:** Learning dense word vector representations - CBOW & Skipgram

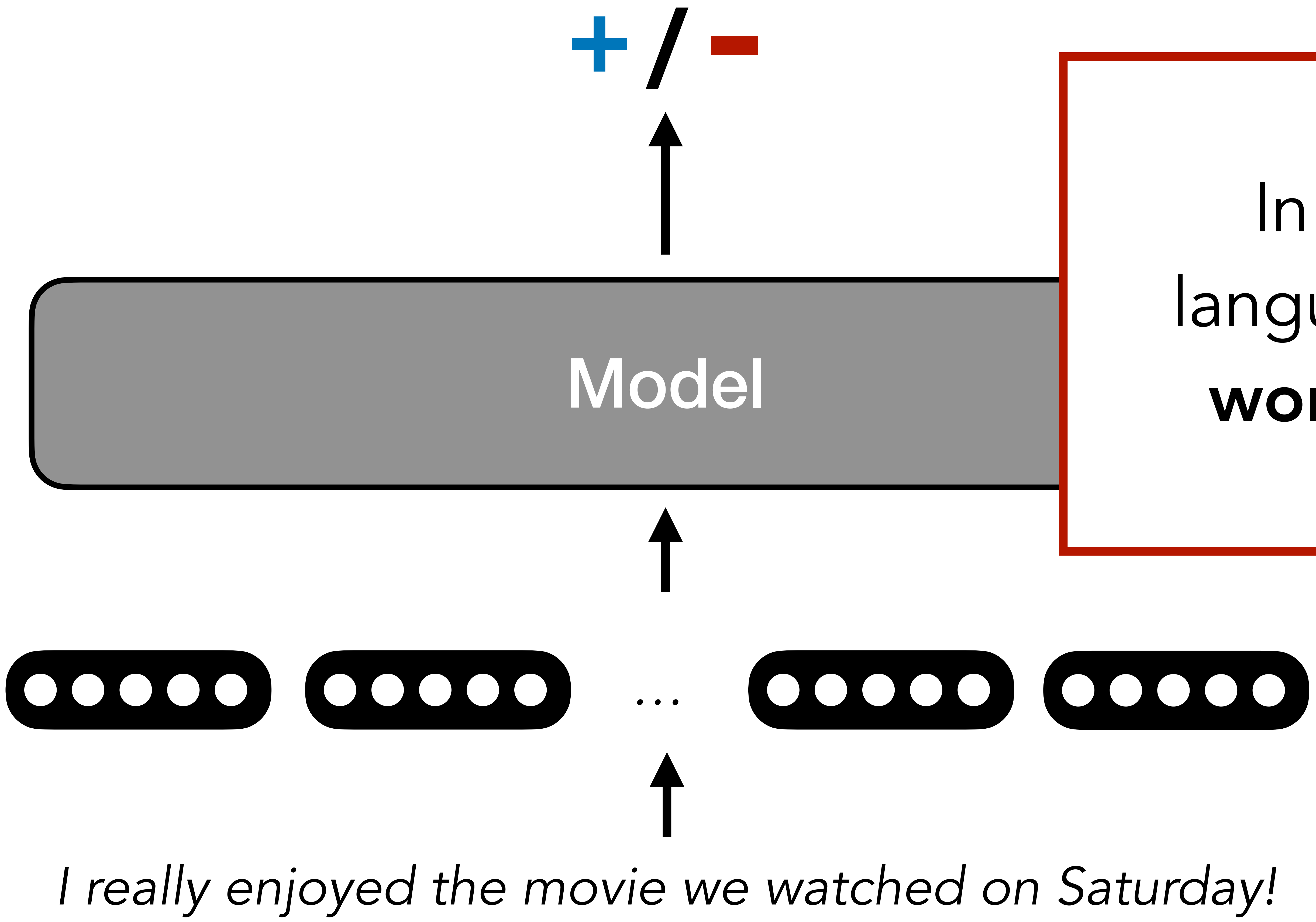
A simple NLP model

- How do we represent natural language sequences for NLP problems?



A simple NLP model

- How do we represent natural language sequences for NLP problems?



In neural natural language processing, **words** are **vectors**!

Question

What words should we model as vectors?

Choosing a vocabulary

- Language contains many words (e.g., ~600,000 in English)
  - **What about other tokens**: Capitalisation? Accents ? Typos!? Words in other languages!? In other scripts!? Emojis !? Unicode !?
  - **Millions of potential unique tokens!** Most rarely appear in our training data (Zipfian distribution)
  - Model has limited capacity

https://en.wikipedia.org/wiki/List_of_dictionaries_by_number_of_words

Choosing a vocabulary

- Language contains many words (e.g., ~600,000 in English)
  - **What about other tokens**: Capitalisation? Accents ? Typos!? Words in other languages!? In other scripts!? Emojis !? Unicode !?
  - **Millions of potential unique tokens!** Most rarely appear in our training data (Zipfian distribution)
  - Model has limited capacity

- How should we select which tokens we want our model to process?
  - CS-552: Modern NLP Week 13 - Tokenisation!
  - For now, initialize a vocabulary *V* of tokens that we can represent as a vector
  - Any token not in this vocabulary *V* is mapped to a special <UNK> token (e.g., unknown).

https://en.wikipedia.org/wiki/List_of_dictionaries_by_number_of_words

Question

How should we model a word as a vector?

Sparse Word Representations

$$w_i \in \{0,1\}^V$$

- Define a vocabulary *V*
- Each word in the vocabulary is represented by a sparse *vector*
- Dimensionality of sparse vector is size of vocabulary (e.g., thousands, possibly millions)

I ⟶ [0 … 0 0 0 1 … 0 0]

really ⟶ [0 … 1 … 0 0 0 0 0]

enjoyed ⟶ [0 … 0 0 0 1 0 … 0]

the ⟶ [0 … 0 1 0 0 0 … 0]

movie ⟶ [0 … 0 0 0 0 0 … 1]

! ⟶ [1 … 0 0 0 0 0 0 0 0 0]

Word Vector Composition

- To represent sequences, beyond words, define a composition function over sparse vectors

I really enjoyed the movie ! → [1 … 1 1 0 1 … 0 1] **Simple Counts**

I really enjoyed the movie ! → [0.01 … 0.1 0.1 0 0.001 … 0 0.5] **Weighted by Corpus Statistics (e.g., TF-IDF)**

Many others…

Problem

With sparse vectors, similarity is a function of common words!

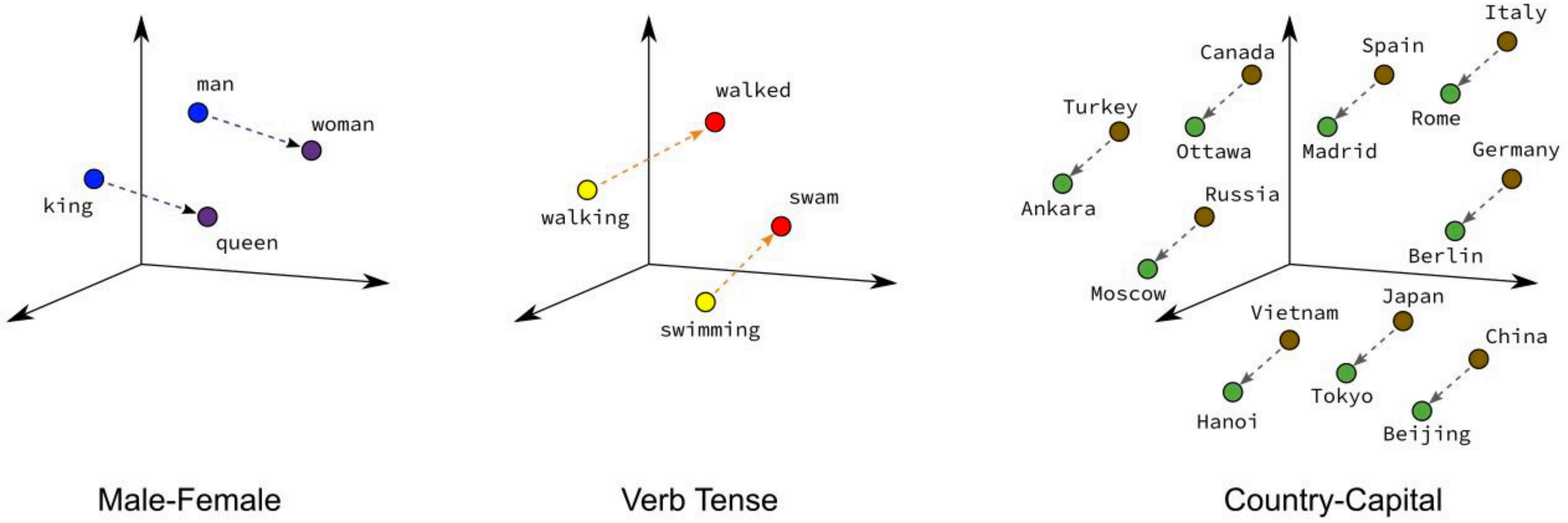
How do you learn learn similarity between words?

enjoyed → [0 … 0 0 0 1 … 0 0]

loved → [0 … 1 … 0 0 0 0 0]

sim(*enjoyed*, *loved*) = **0**

Embeddings Goal



How do we train semantics-encoding embeddings of words?

Image Credit: https://towardsdatascience.com/legal-applications-of-neural-word-embeddings-556b7515012f

Dense Word Vectors

- Represent each word as a high-dimensional*, **real-valued** vector
  - *Low-dimensional compared to V-dimension sparse representations, but still usually $O(10^2 - 10^3)$

I → [0.113 -0.782 1.893 0.984 6.349 ...]

really → [0.906 0.661 -0.214 -0.894 -0.880 ...]

enjoyed → [-0.842 0.647 -0.882 0.045 0.029 ...]

the → [0.100 0.765 -0.333 -0.538 -0.150 ...]

movie → [0.104 -0.054 -0.268 -0.877 0.005 ...]

! → [0.439 -0.577 -0.727 0.261 0.699 ...]

word vectors

word embeddings

neural embeddings

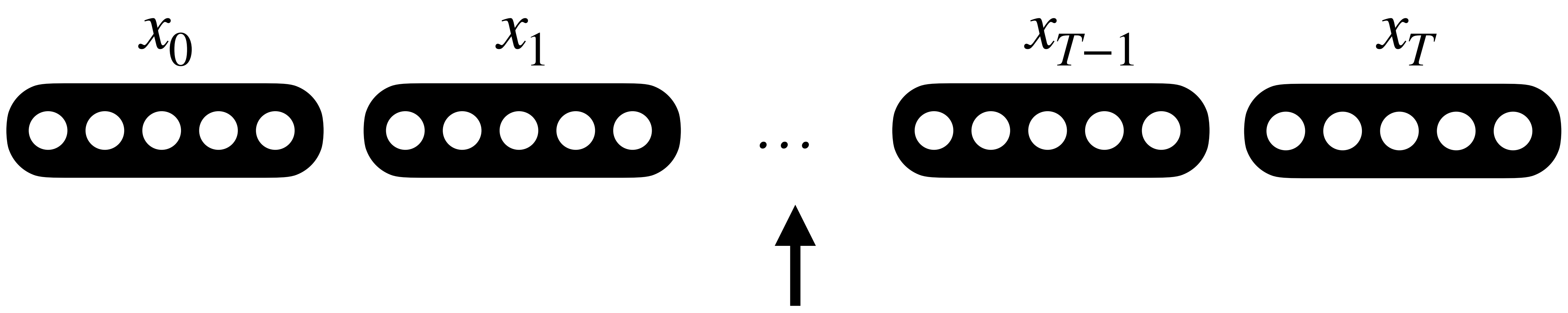
dense embeddings

others...

- Similarity of vectors represents similarity of meaning for particular words

A simple NLP model

- For each sequence S, we have a corresponding sequence of embeddings X



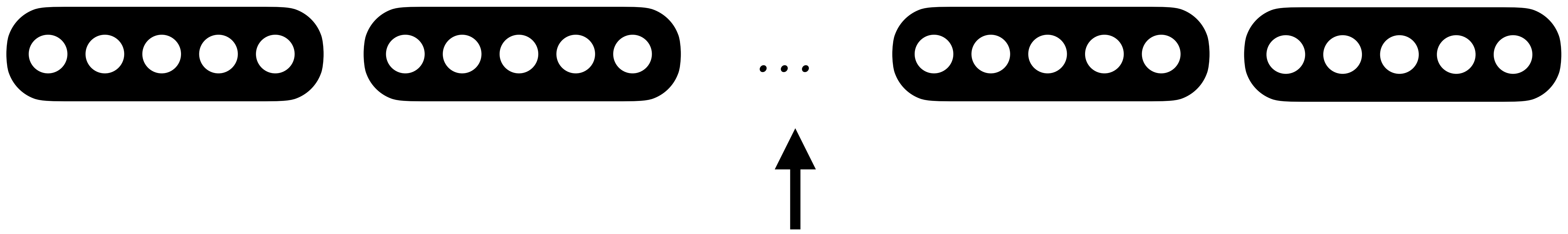
$X = \{x_0, x_1, \ldots, x_T\}$

$S =$ *I really enjoyed the movie we watched on Saturday!*

A simple NLP model

- For each sequence S, we have a corresponding sequence of embeddings X

x_0 x_1 $\ldots$ x_{T-1} x_T



$$X = \{x_0, x_1, \ldots, x_T\}$$

$S_1 =$ *I **really** enjoyed the movie **we** watched on Saturday **!***

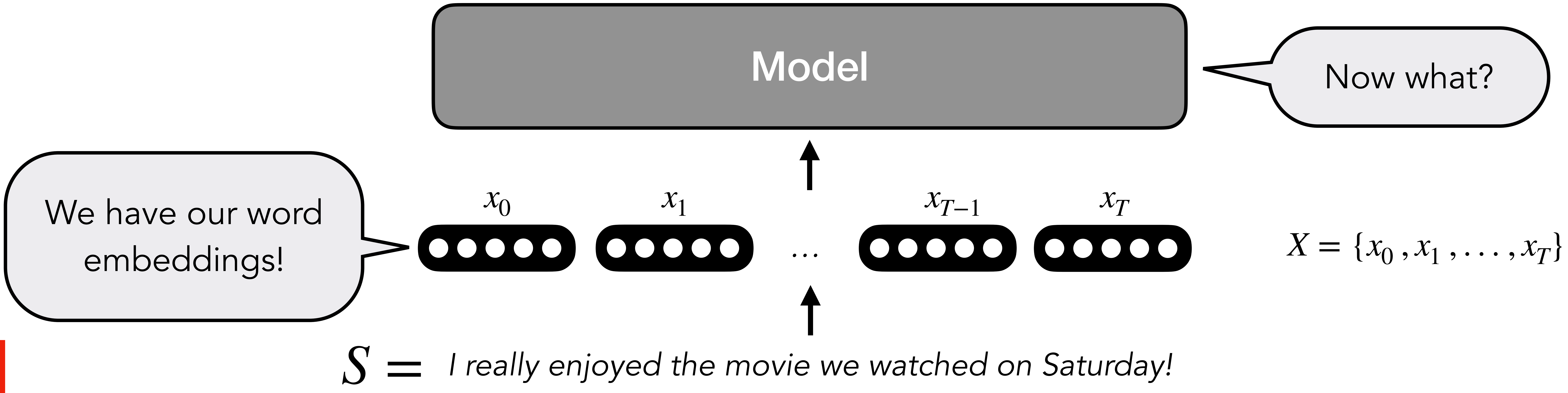
- Embeddings $x_t \in X$ are indexed from shared embedding dictionary $\mathbb{E}$ for all items in vocabulary V

$S_2 =$ ***We** really loved a film **we** saw last Sunday **!***

Bolded words would index the same embedding in $\mathbb{E}$

A simple NLP model

- For each sequence S, we have a corresponding sequence of embeddings X

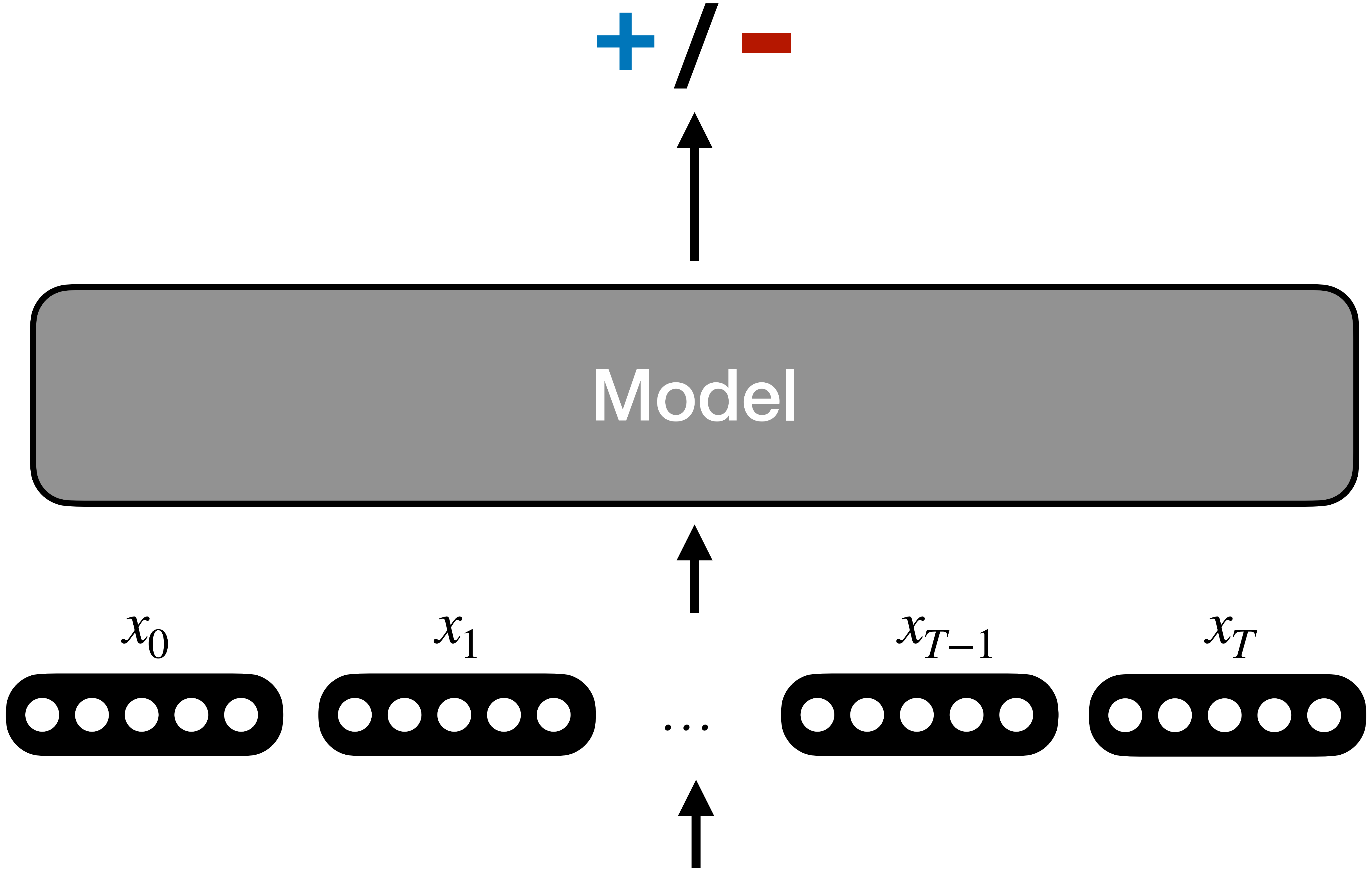


Question

What should we use as a model?

A simple NLP model

- Our model modifies and / or composes these word embeddings to formulate a representation that allows it to predict the correct label



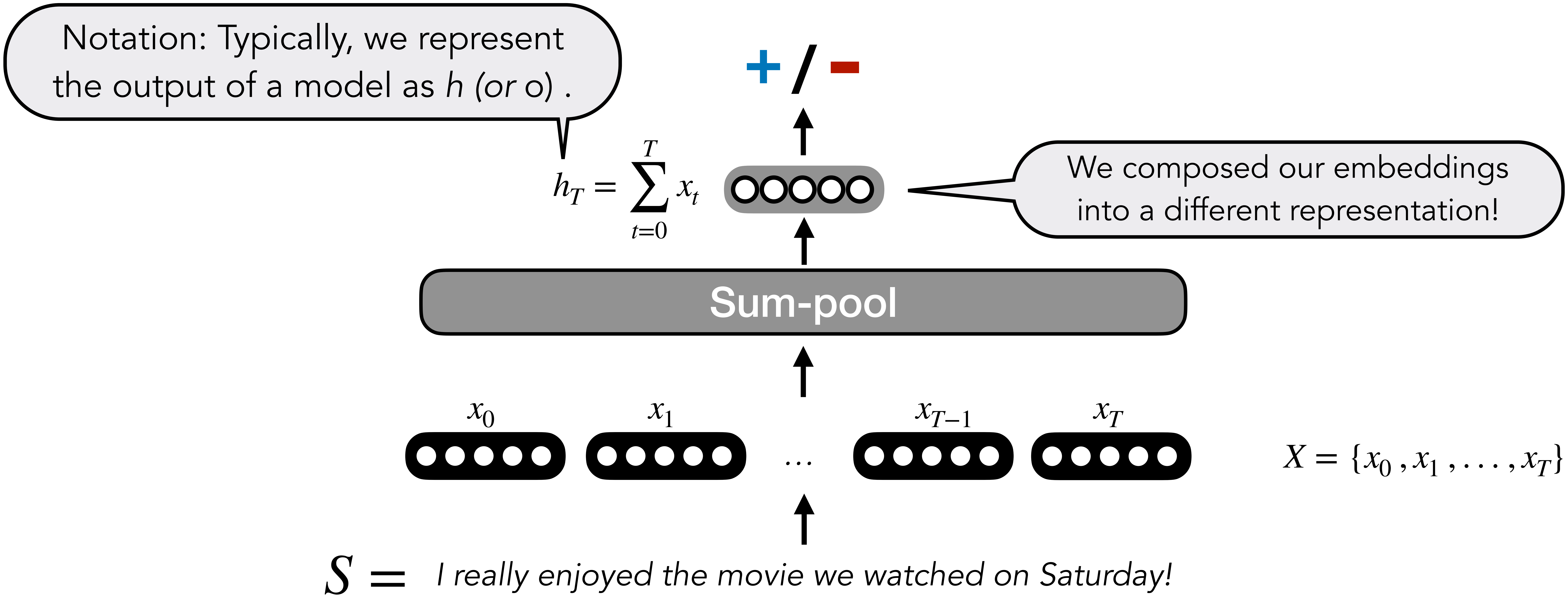
$X = \{x_0 \,, x_1 \,, \ldots, x_T\}$

$S =$ *I really enjoyed the movie we watched on Saturday!*

A simple NLP model

- Our model modifies and / or composes these word embeddings to formulate a representation that allows it to predict the correct label
  - Recurrent neural networks (RNNs) - Today!
  - RNN variants (LSTM, GRU, etc.) - Today!
  - Transformer - Today!

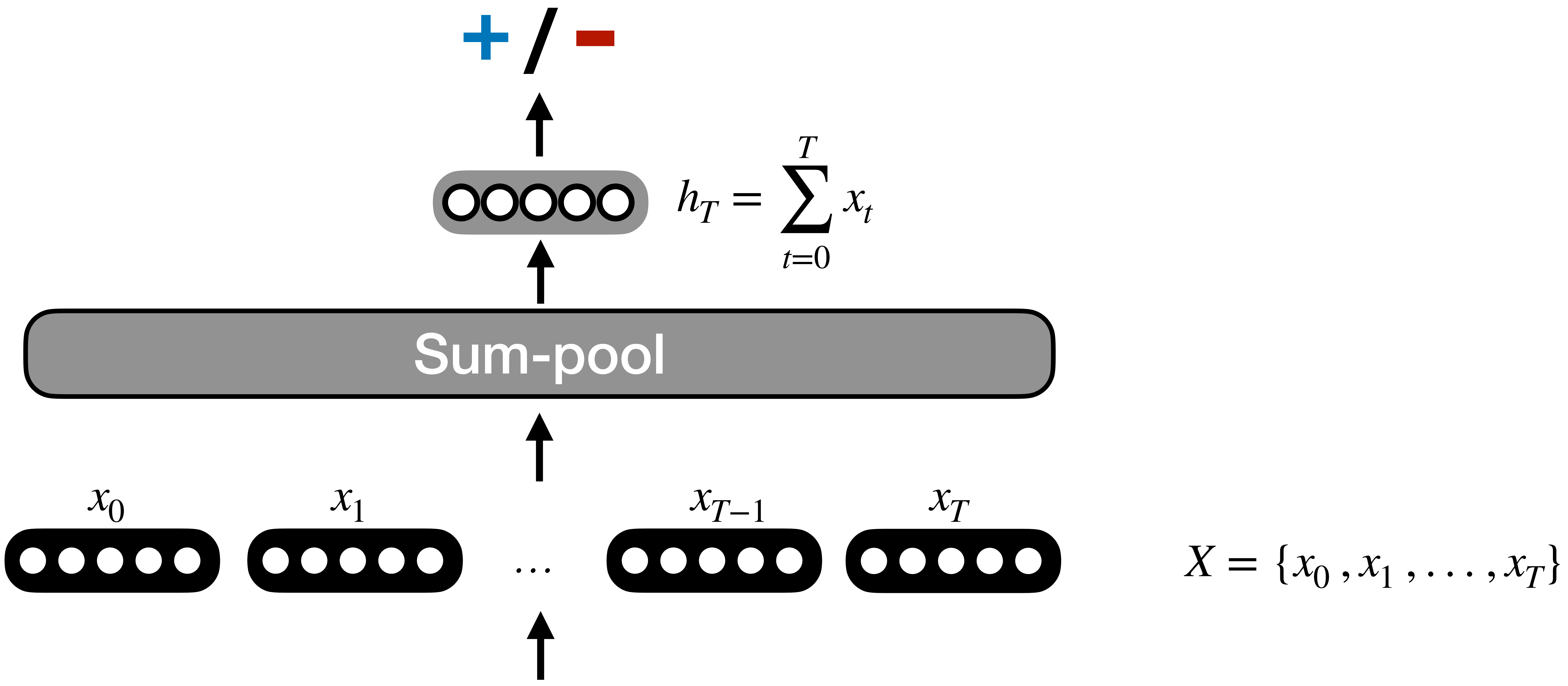
A simple NLP model



Question

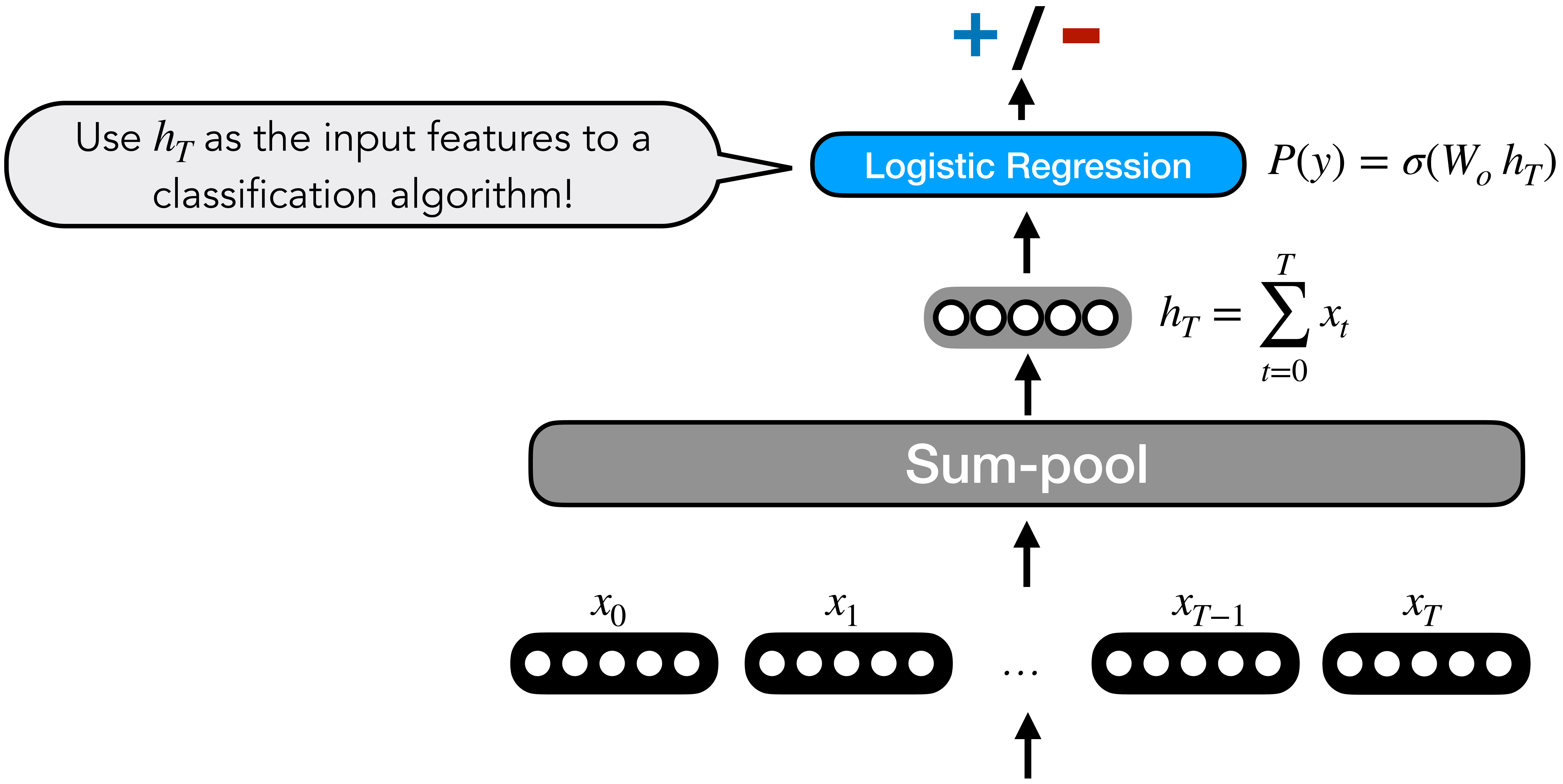
How do we convert the output of our model to a prediction?

Predicting the label



$S =$ *I really enjoyed the movie we watched on Saturday!*

Predicting the label



$S =$ *I really enjoyed the movie we watched on Saturday!*

Learn using **backpropagation:** compute gradients of loss with respect to initial embeddings X

Learn embeddings that allow you to do the task successfully!

$X = \{x_0\,, x_1\,, \ldots, x_T\}$

Question

What could be a better way to learn word embeddings?

"You shall know a word by the company it keeps"

–J.R. Firth, 1957

Context Representations

Solution:

Rely on the context in which words occur to learn their meaning

Context is the **set of words** that occur **nearby**

I really enjoyed the ____ we watched on Saturday!

The ___ growled at me, making me run away.

I need to go to the ____ to pick up some dinner.

Foundation of distributional semantics

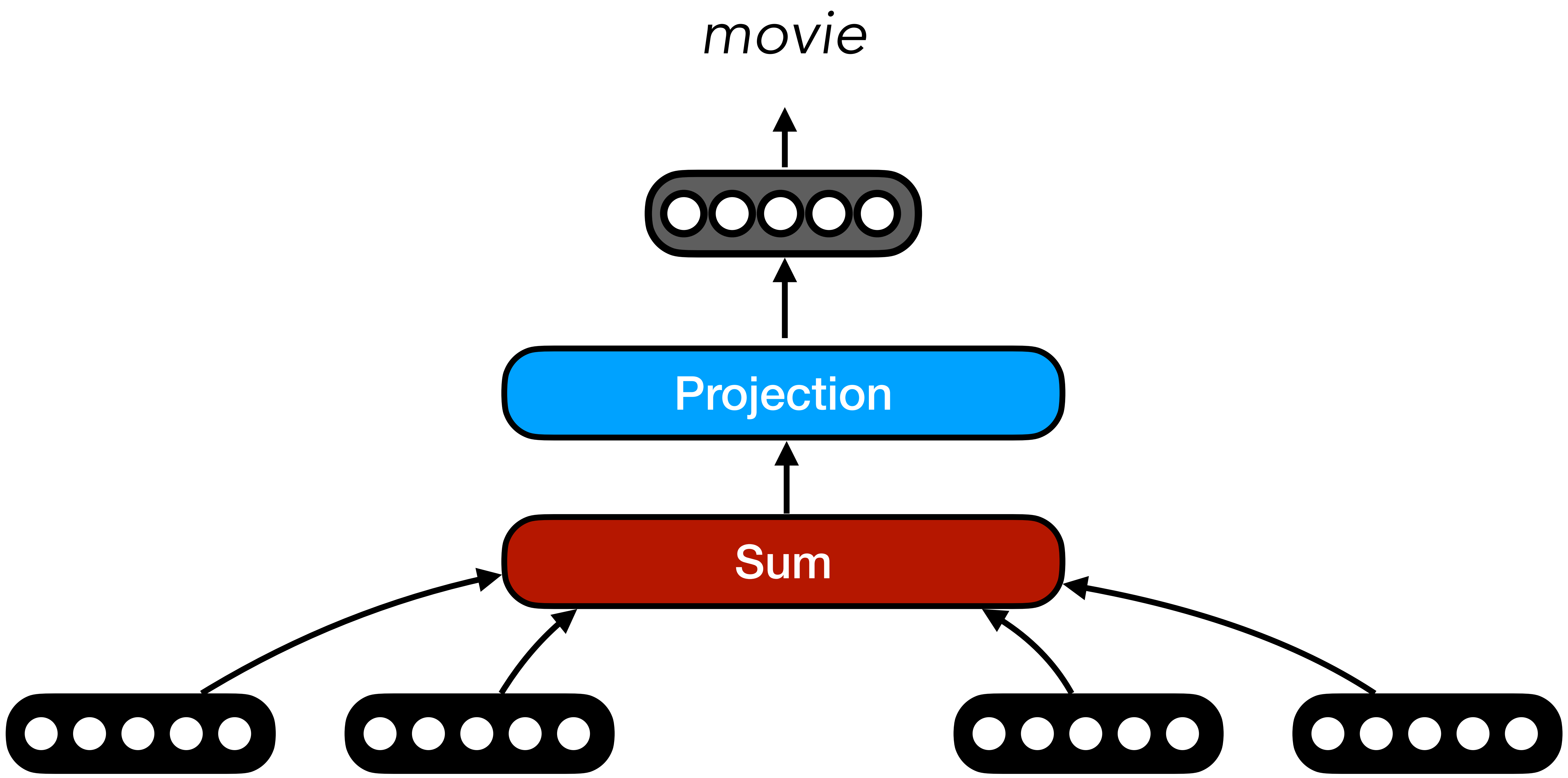
Learning Word Embeddings

- Many options, huge area of research, but three common approaches
- **Word2vec - Continuous Bag of Words (CBOW)**
  - Learn to predict missing word from surrounding window of words
- **Word2vec - Skip-gram**
  - Learn to predict surrounding window of words from given word
- **GloVe**
  - Not covered today

(Mikolov et al., 2013a; 2013b; Pennington et al., 2014)

Continuous Bag of Words (CBOW)

- Predict the missing word from a window of surrounding words

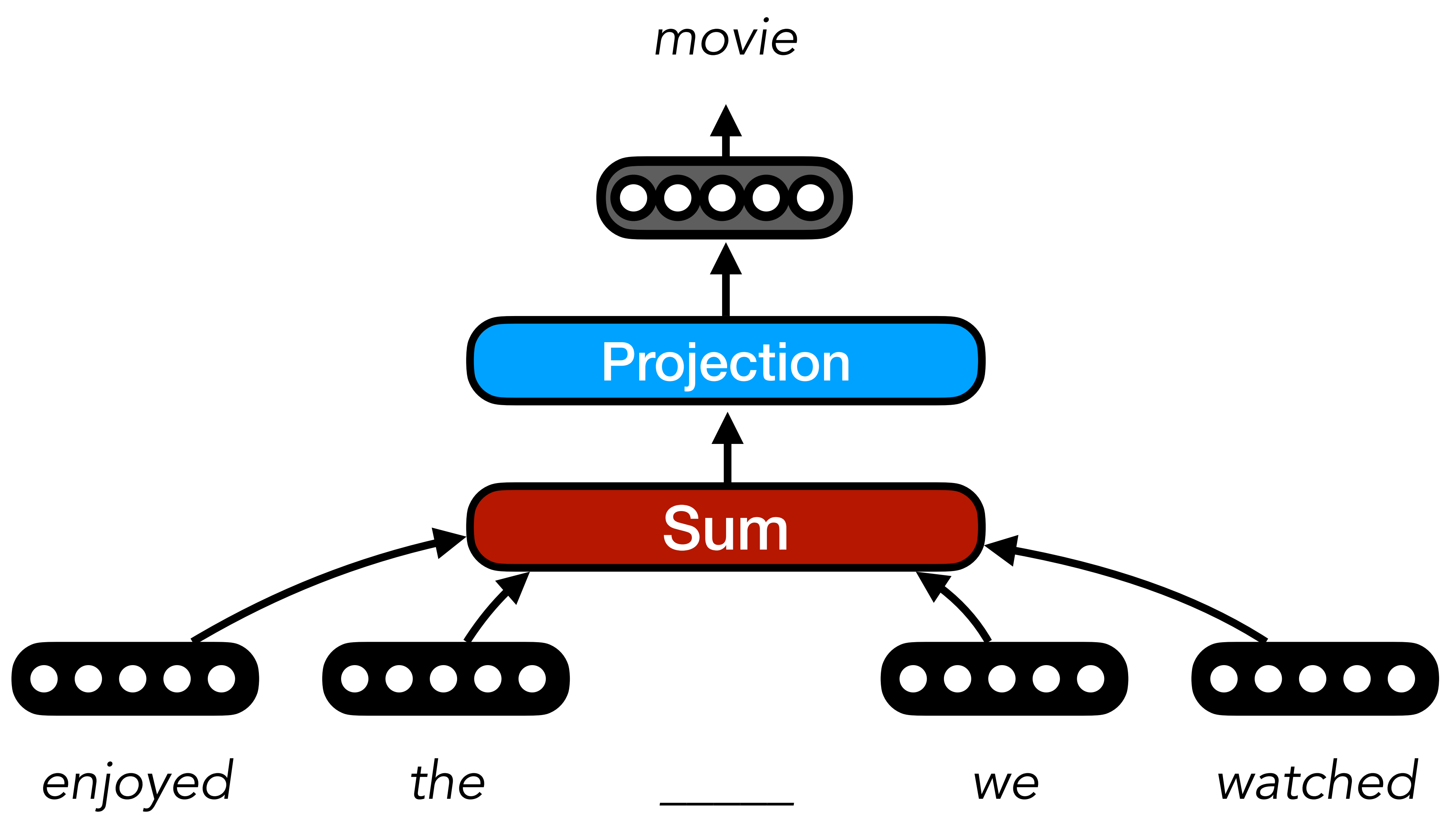


Context: *enjoyed* *the* ____ *we* *watched*

(Mikolov et al., 2013a)

Continuous Bag of Words (CBOW)

- Predict the missing word from a window of surrounding words



$$\max P(movie \mid enjoyed, the, we, watched)$$

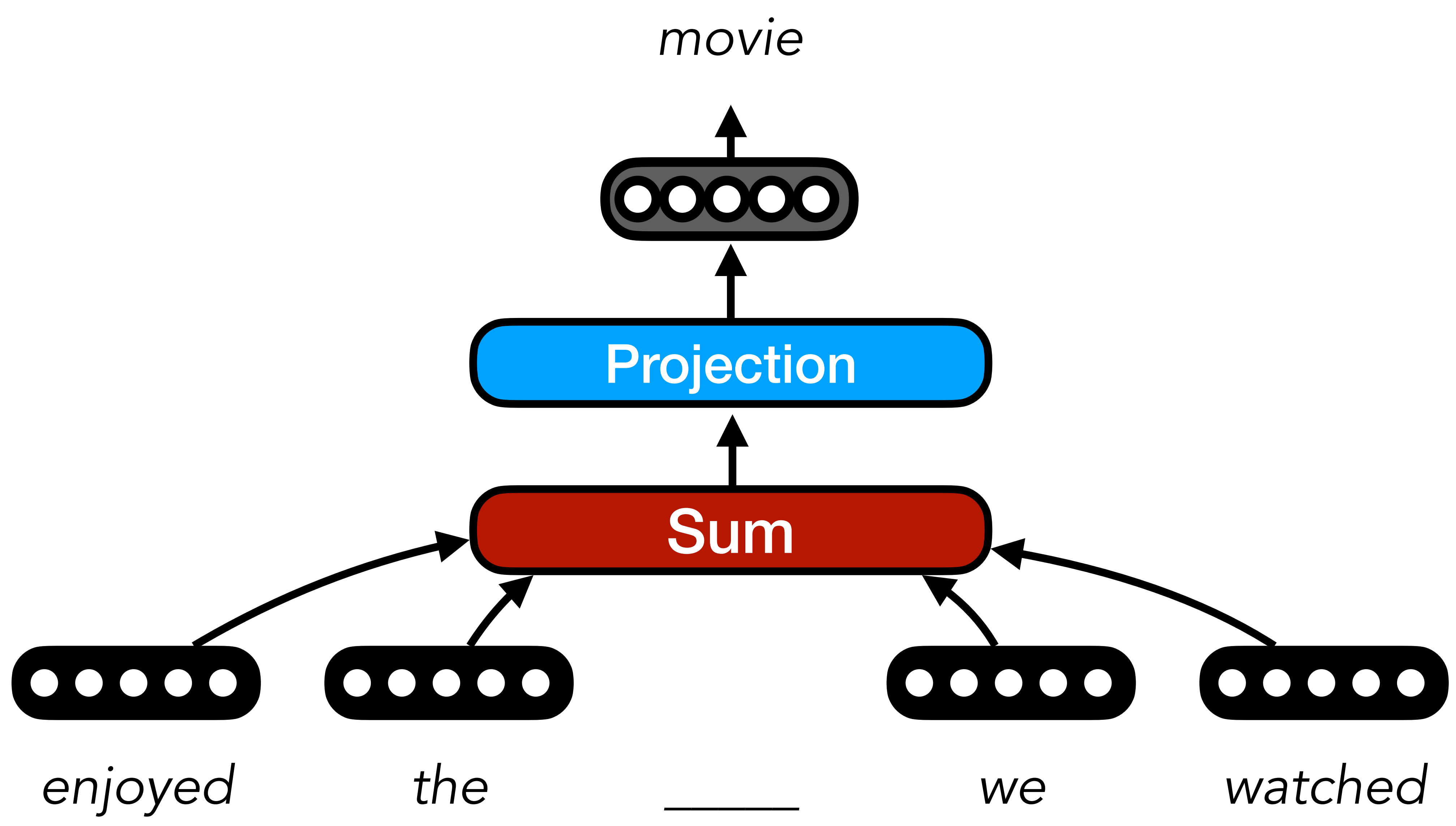
$$\max P(w_t \mid w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2})$$

$$\max P(w_t \mid \{w_x\}_{x=t-2}^{x=t+2})$$

(Mikolov et al., 2013a)

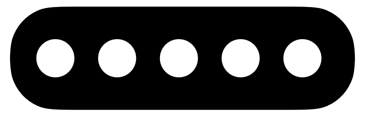
Continuous Bag of Words (CBOW)

- Predict the missing word from a window of surrounding words



$$P(w_t \mid \{w_x\}_{x=t-2}^{x=t+2}) = \textbf{softmax}\left(\mathbf{U} \sum_{\substack{x=t-2 \\ x \neq t}}^{t+2} \mathbf{w}_x\right)$$

$\mathbf{w}_x \in \mathbb{R}^{1 \times d}$



$\mathbf{U} \in \mathbb{R}^{d \times V}$

(Mikolov et al., 2013a)

Softmax Function

- The **softmax** function generates a probability distribution from the elements of the vector it is given

$$\textbf{softmax}(\mathbf{a})_i = \frac{e^{a_i}}{\sum_{j=1}^{|\mathbf{a}|} e^{a_j}}$$

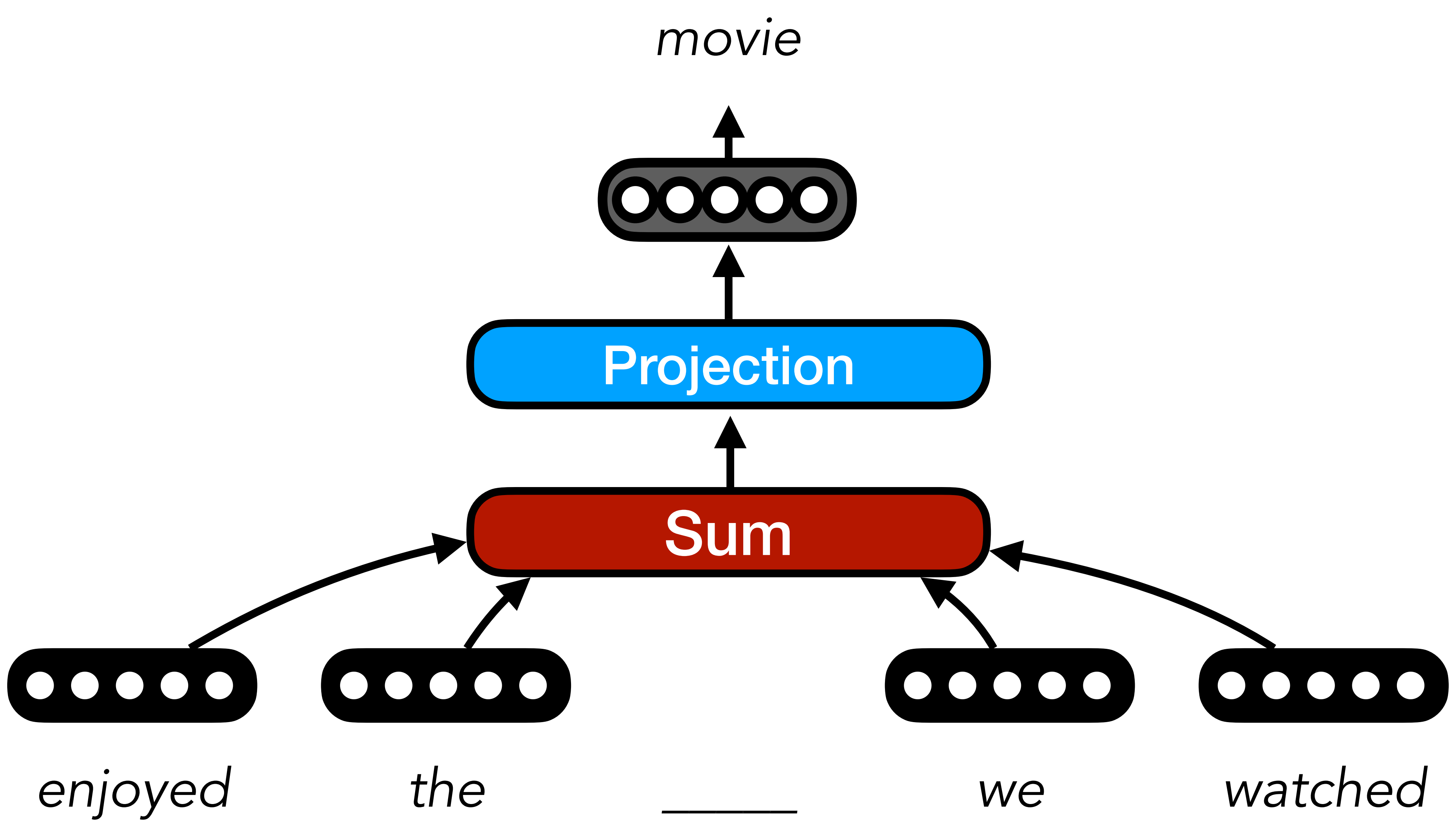
V = [0.790 -0.851 0.506 0.767 -0.788 0.793 0.887 0.219 -0.052 0.461]

$\Big\downarrow$ **Softmax(V)**

P(V) = [0.144 0.028 0.108 0.141 0.030 0.144 0.159 0.081 0.062 0.104]

Continuous Bag of Words (CBOW)

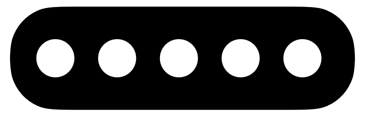
- Predict the missing word from a window of surrounding words



$$P(w_t \mid \{w_x\}_{x=t-2}^{x=t+2}) = \textbf{softmax}\left(\mathbf{U} \sum_{\substack{x=t-2 \\ x \neq t}}^{t+2} \mathbf{w}_x\right)$$

$\mathbf{w}_x \in \mathbb{R}^{1 \times d}$

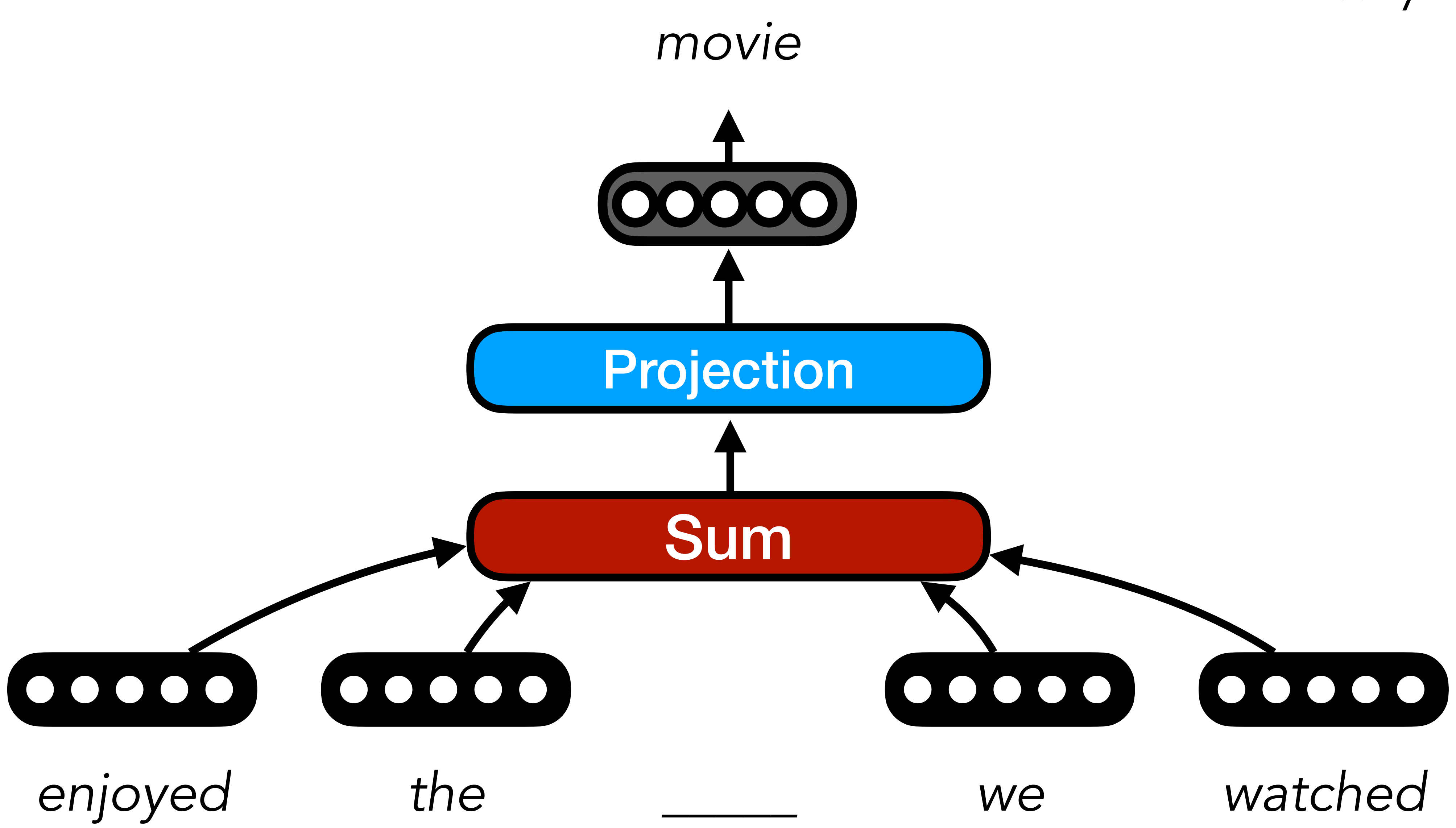
$\mathbf{U} \in \mathbb{R}^{d \times V}$



(Mikolov et al., 2013a)

Continuous Bag of Words (CBOW)

$$P(w_t \mid \{w_x\}_{x=t-2}^{x=t+2}) = \textbf{softmax}\left(\mathbf{U} \sum_{\substack{x = t-2 \\ x \neq t}}^{t+2} \mathbf{w}_x\right)$$

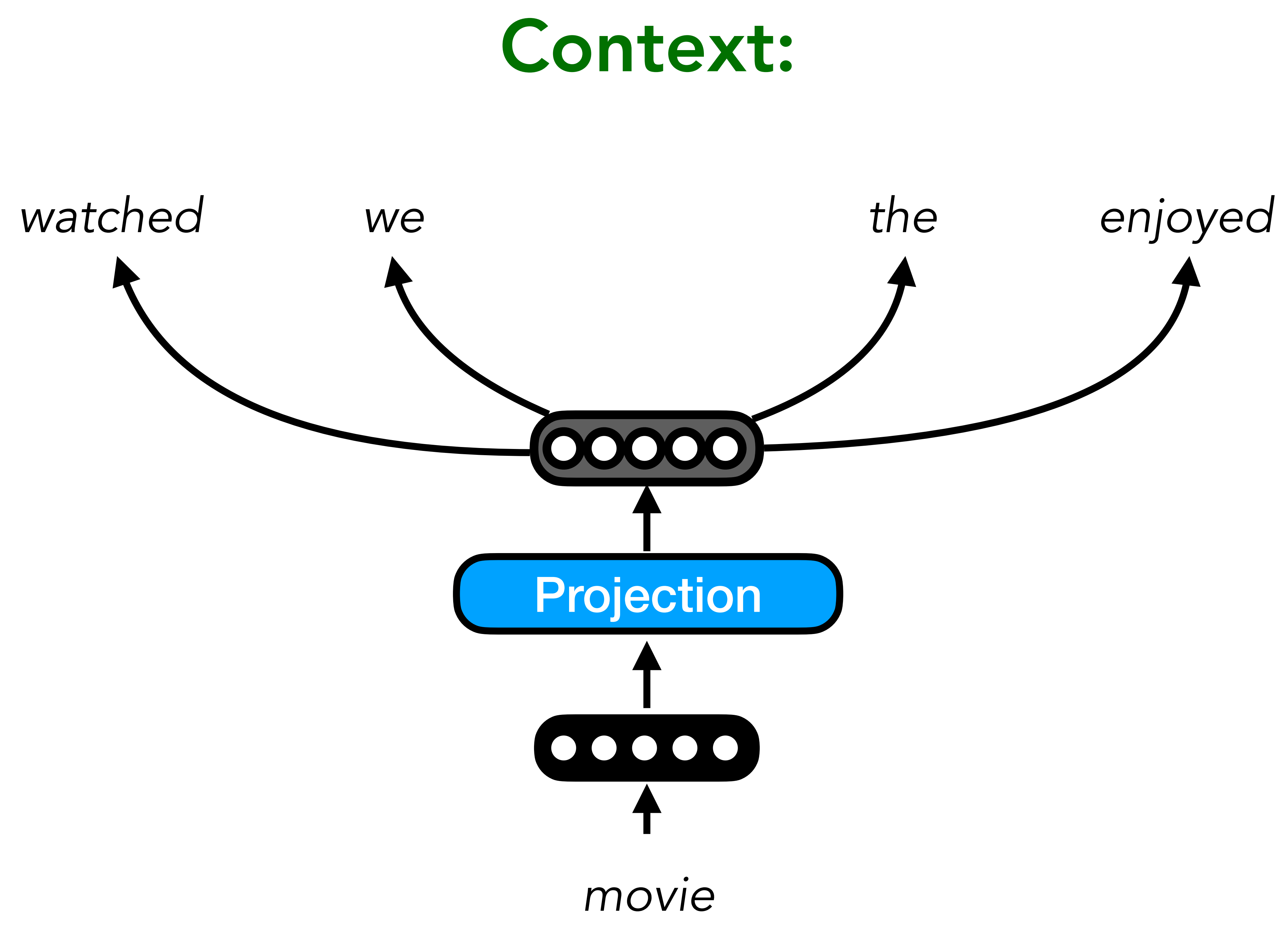


- Model is trained to **maximise** the **probability** of the missing word
  - For computational reasons, the model is typically trained to **minimise** the **negative log probability** of the missing word
- Here, we use a window of **N=2**, but the window size is a **hyperparameter**
- For computational reasons, a **hierarchical softmax** used to compute distribution

(Mikolov et al., 2013a)

Skip-gram

- We can also learn embeddings by predicting the surrounding context from a single word



$$\max P(enjoyed, the, we, watched \mid movie)$$

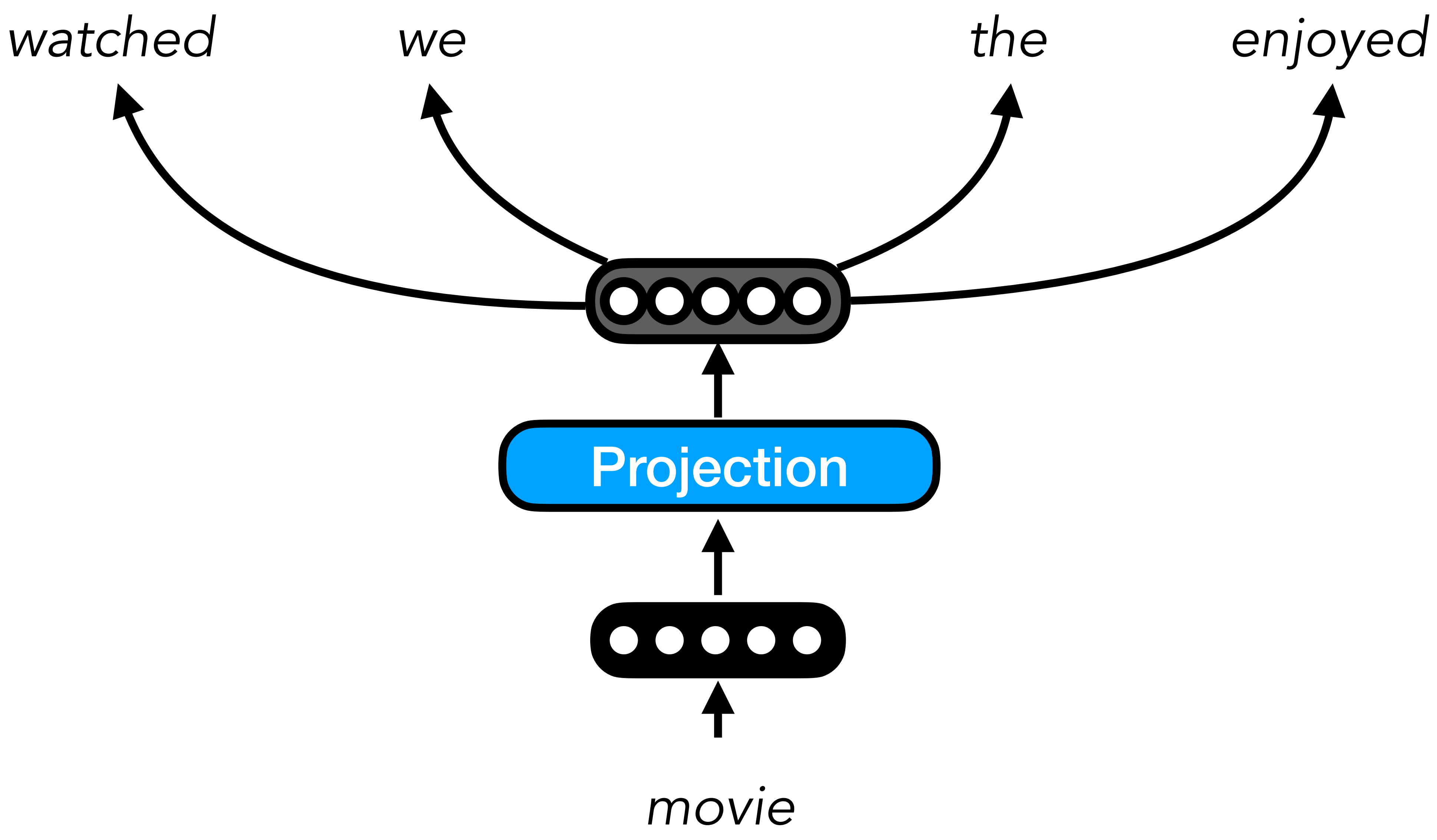
$$\max P(w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2} \mid w_t)$$

(Mikolov et al., 2013b)

Skip-gram

- We can also learn embeddings by predicting the surrounding context from a single word

Context:



$$\max P(enjoyed, the, we, watched \mid movie)$$

$$\max P(w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2} \mid w_t)$$

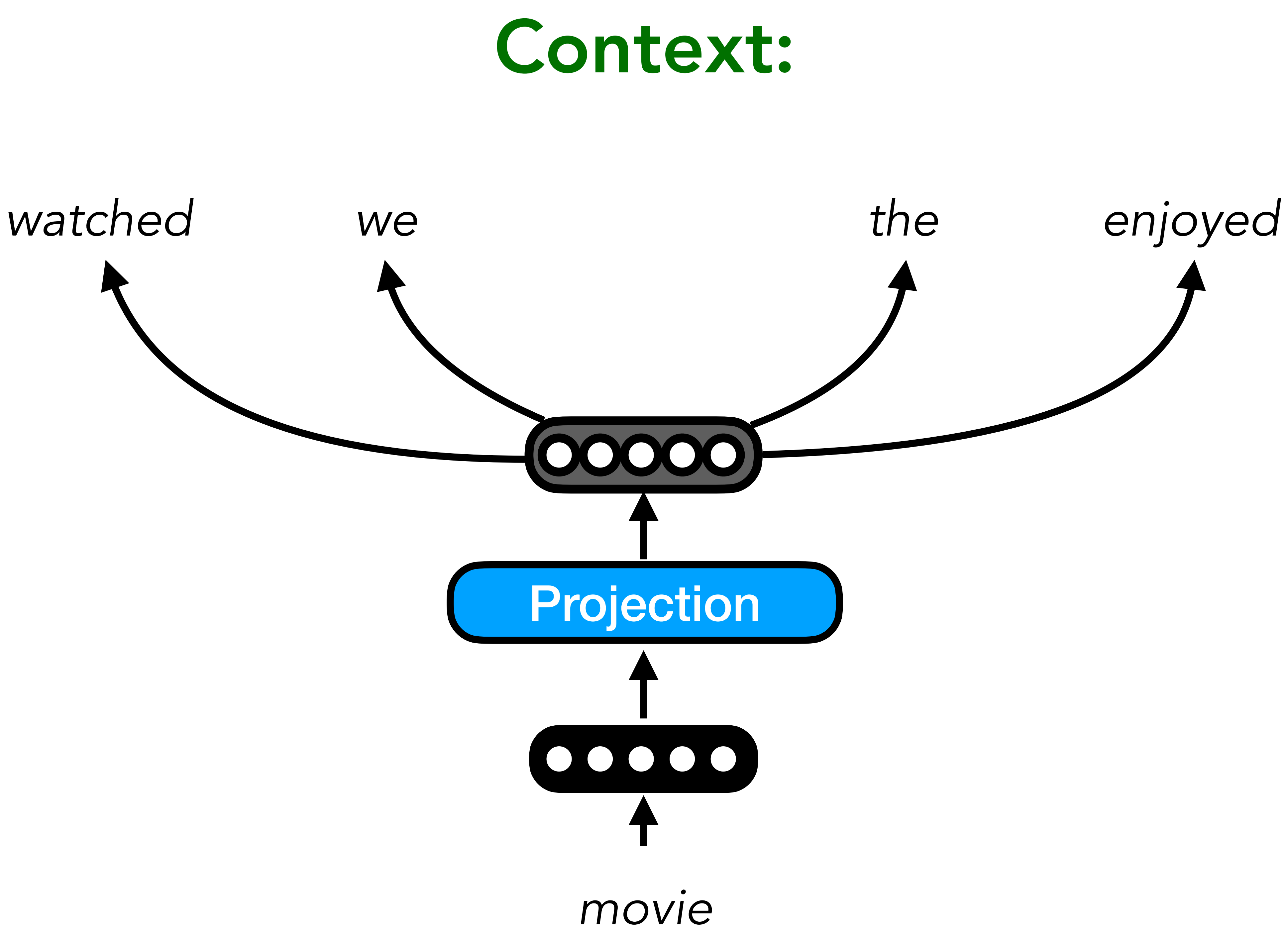
$$\max \log P(w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2} \mid w_t)$$

$$\max \Big(\log P(w_{t-2} \mid w_t) + \log P(w_{t-1} \mid w_t) + \log P(w_{t+1} \mid w_t) + \log P(w_{t+2} \mid w_t) \Big)$$

(Mikolov et al., 2013b)

Skip-gram

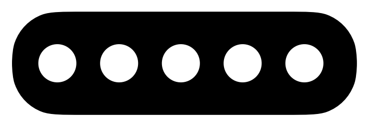
- We can also learn embeddings by predicting the surrounding context from a single word



$$P(w_x \mid w_t) = \textbf{softmax}\left(\mathbf{U}\mathbf{w}_t\right)$$

$$\mathbf{w}_t \in \mathbb{R}^{1\times d}$$

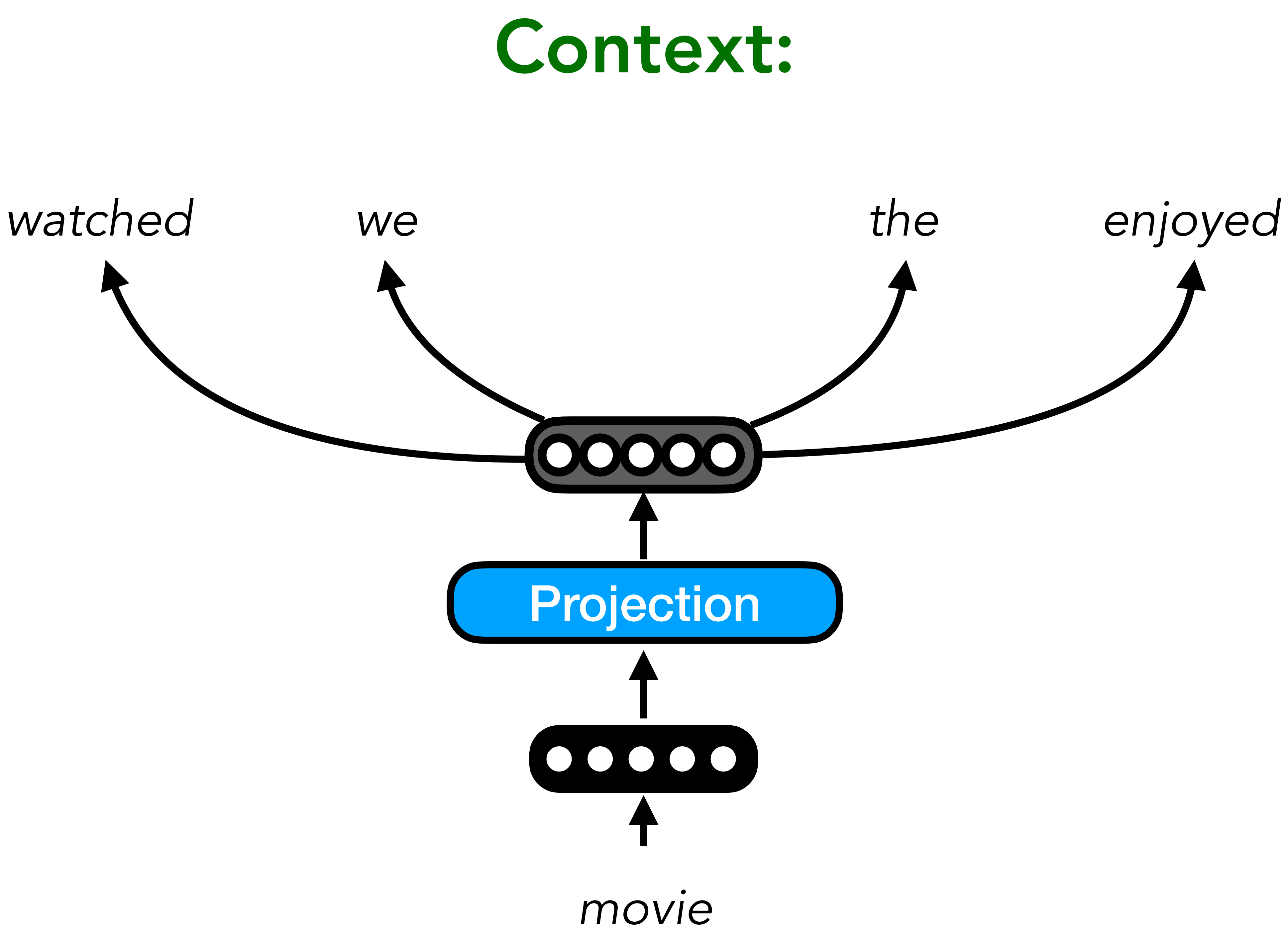
$$\mathbf{U} \in \mathbb{R}^{d\times V}$$



(Mikolov et al., 2013b)

Skip-gram

- We can also learn embeddings by predicting the surrounding context from a single word



- Model is trained to **minimise** the **negative log probability** of the surrounding words
- Here, we use a window of **N=2**, but the window size is a **hyperparameter** to set
- Typically, set large window **(N=10),** but randomly select $i \in [1,N]$ as dynamic window size so that closer words contribute more to learning

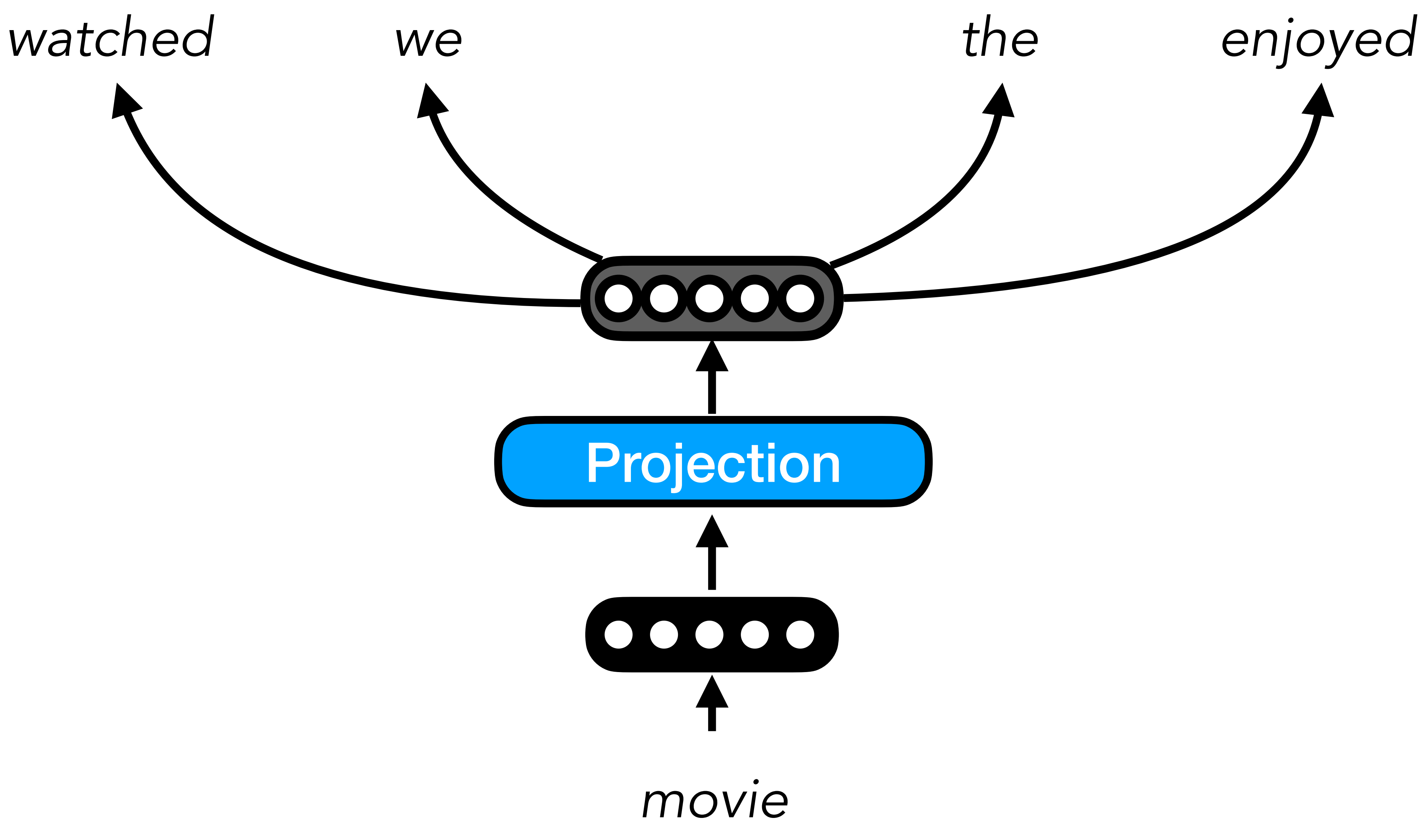
(Mikolov et al., 2013b)

Question

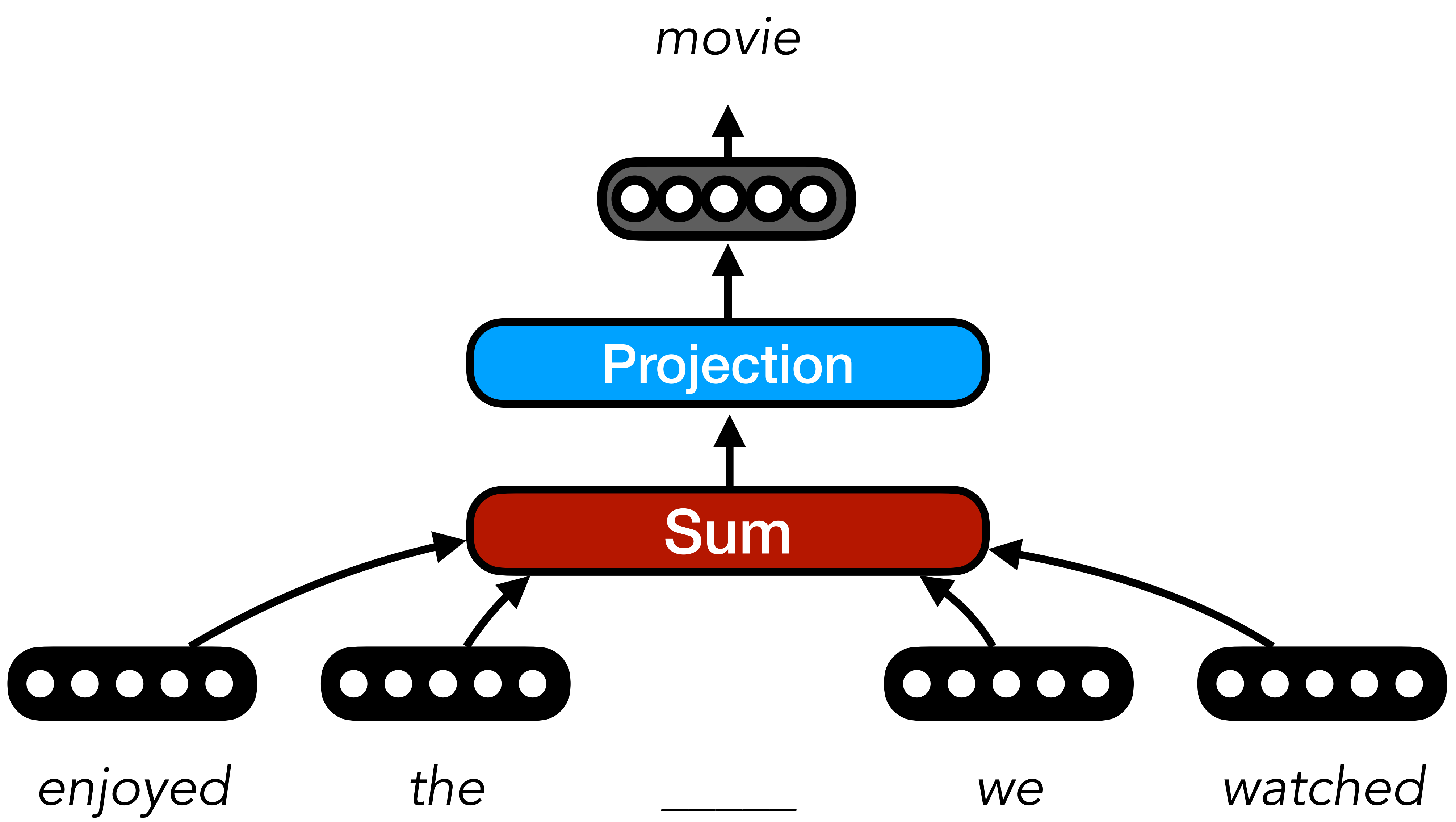
What is the major conceptual difference between the CBOW and Skipgram methods for training word embeddings?

Skip-gram vs. CBOW

- **Question:** Do you expect a difference between what is learned by CBOW and Skipgram methods?



(Mikolov et al., 2013b)



(Mikolov et al., 2013a)

Example

CBOW

```
[ ]  top_cbow = cbow.wv.most_similar('cut', topn=10)

     print(tabulate(top_cbow, headers=["Word", "Simi
```

```
Word          Similarity
----------  ------------
slice           0.662173
crosswise       0.650036
score           0.630569
tear            0.618827
dice            0.563946
lengthwise      0.557231
cutting         0.557228
break           0.551517
chop            0.541566
carve           0.537967
```

Skip-gram

```
[ ]  top_sg = skipgram.wv.most_similar('cut', topn=10)

     print(tabulate(top_sg, headers=["Word", "Similarit
```

```
Word              Similarity
--------------  ------------
crosswise           0.72921
score               0.702693
slice               0.696898
crossways           0.680091
1/2-inch-thick      0.678496
diamonds            0.671814
diagonally          0.670319
lengthwise          0.665378
cutting             0.66425
wise                0.656825
```

Recap

- Neural NLP: **Words are vectors!**
- Word embeddings can be learned in a self-supervised manner from large quantities of raw text
- **Two algorithms:** Continuous Bag of Words (CBOW) and Skip-gram

Resources

- **word2vec**: https://code.google.com/archive/p/word2vec/
- **GloVe**: https://nlp.stanford.edu/projects/glove/
- **FastText**: https://fasttext.cc/
- **Gensim:** https://radimrehurek.com/gensim/

Download pre-trained word vectors

- Pre-trained word vectors. This data is made available under the Public Domain Dedication and License v1.0 whose full text can be found at: http://www.opendatacommons.org/licenses/pddl/1.0/.
  - Wikipedia 2014 + Gigaword 5 (6B tokens, 400K vocab, uncased, 50d, 100d, 200d, & 300d vectors, 822 MB download): glove.6B.zip
  - Common Crawl (42B tokens, 1.9M vocab, uncased, 300d vectors, 1.75 GB download): glove.42B.300d.zip
  - Common Crawl (840B tokens, 2.2M vocab, cased, 300d vectors, 2.03 GB download): glove.840B.300d.zip
  - Twitter (2B tweets, 27B tokens, 1.2M vocab, uncased, 25d, 50d, 100d, & 200d vectors, 1.42 GB download): glove.twitter.27B.zip
- Ruby script for preprocessing Twitter data

References

- Firth, J.R. (1957). A Synopsis of Linguistic Theory, 1930-1955.
- Mikolov, T., Chen, K., Corrado, G.S., & Dean, J. (2013a). Efficient Estimation of Word Representations in Vector Space. *International Conference on Learning Representations.*
- Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., & Dean, J. (2013b). Distributed Representations of Words and Phrases and their Compositionality. *ArXiv, abs/1310.4546*.
- Pennington, J., Socher, R., & Manning, C.D. (2014). GloVe: Global Vectors for Word Representation. *Conference on Empirical Methods in Natural Language Processing*.
- Bojanowski, P., Grave, E., Joulin, A., & Mikolov, T. (2017). Enriching word vectors with subword information. *Transactions of the association for computational linguistics.*
- Mikolov, T., Grave, E., Bojanowski, P., Puhrsch, C., & Joulin, A. (2018). Advances in pre-training distributed word representations. *International Conference on Language Resources and Evaluation.*

Deep Learning for
Natural Language Processing

Antoine Bosselut

EPFL

nlp lab

Part 3: Attentive Neural Modeling with Transformers

Section Outline

- **Background**: Long-range Dependency Modeling
- **Content:** Attention, Self-Attention, Multi-headed Attention, Transformer Blocks, Transformers
- **Exercise Session:** Visualizing Transformer Attention

Issue with Recurrent Models

- Multiple steps of state overwriting makes it challenging to learn long-range dependencies.

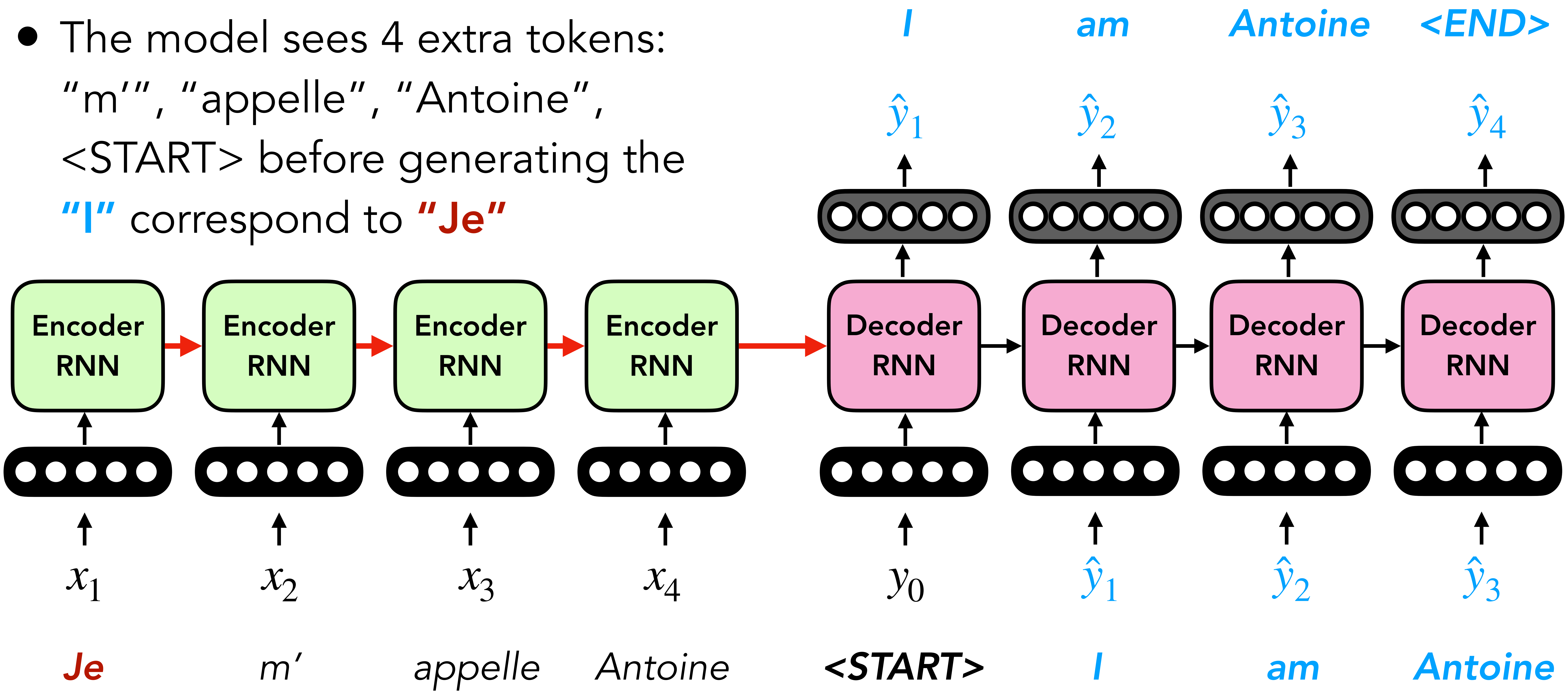
They tuned, discussed for a moment, then struck up a lively ***jig****. Everyone joined in, turning the courtyard into an even more chaotic scene, people now* ***dancing*** *in circles,* ***swinging*** *and* ***spinning*** *in circles, everyone making up their own* ***dance steps****. I felt my feet tapping, my body wanting to move. Aside from writing, I 've always loved* **dancing** *.*

- Nearby words should affect each other more than farther ones, but RNNs make it challenging to learn **any** long-range interactions

LAMBADA dataset, 2016

Toy Example

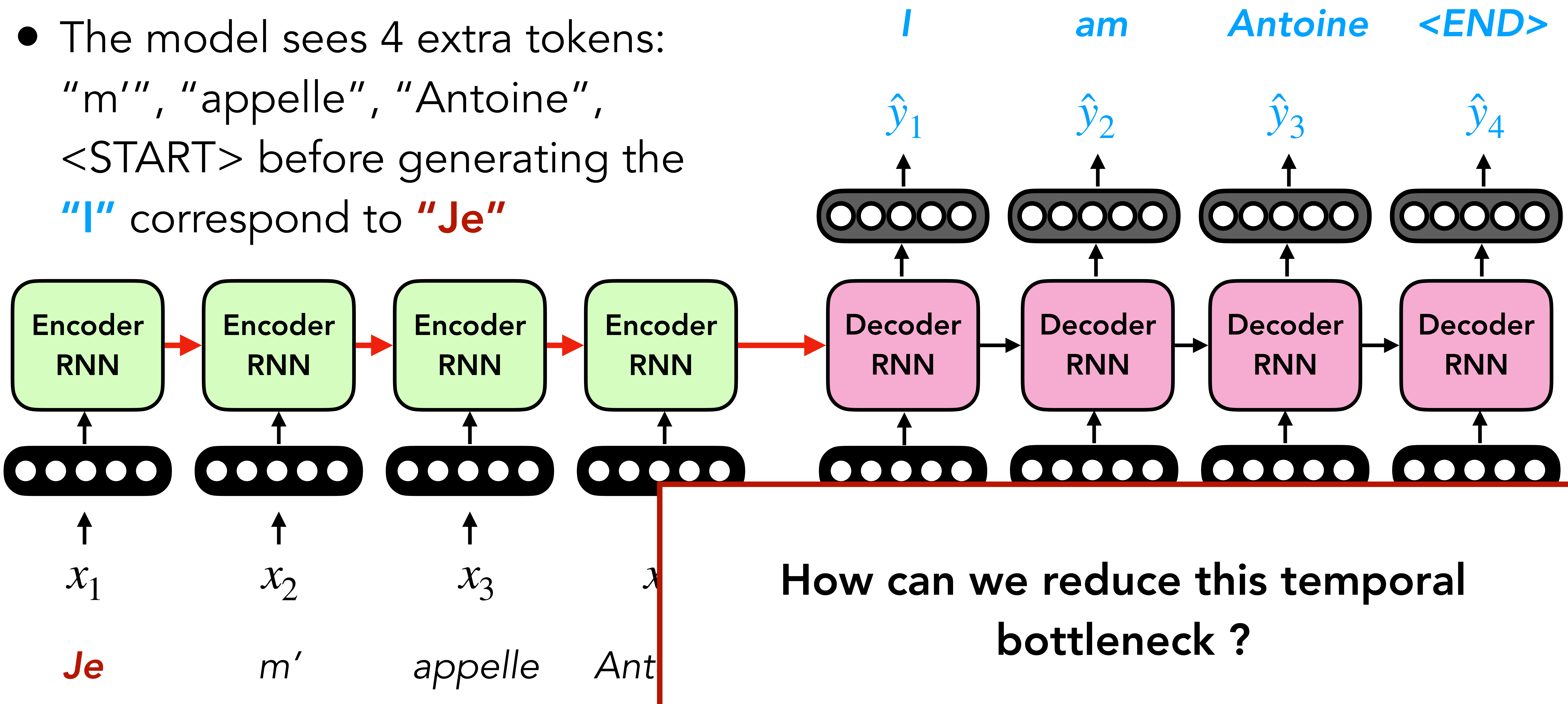
- The model sees 4 extra tokens: "m'", "appelle", "Antoine", <START> before generating the **"I"** correspond to **"Je"**



(Sutskever et al., 2014)

Toy Example

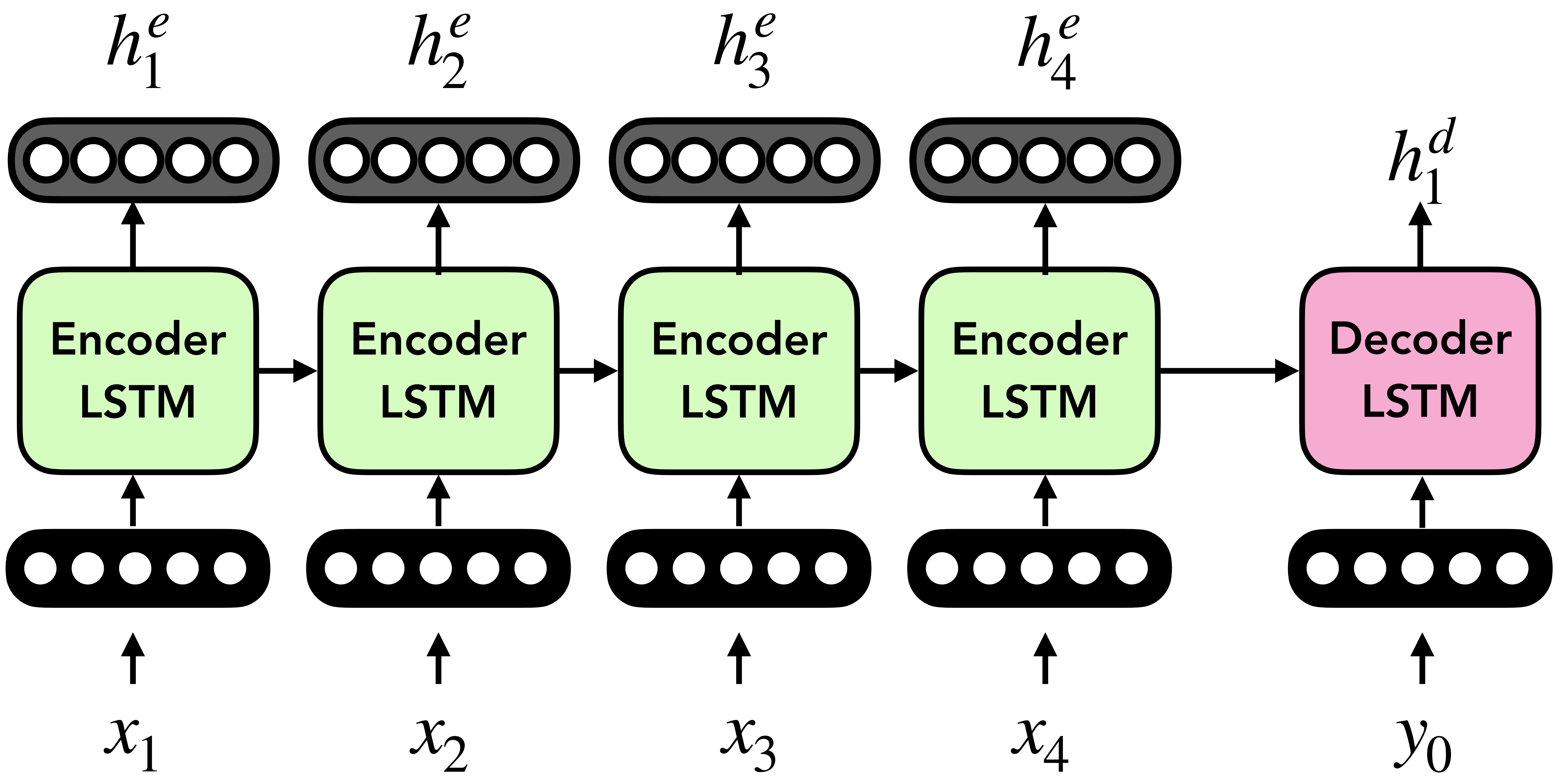
- The model sees 4 extra tokens: "m'", "appelle", "Antoine", <START> before generating the **"I"** correspond to **"Je"**



How can we reduce this temporal bottleneck ?

(Sutskever et al., 2014)

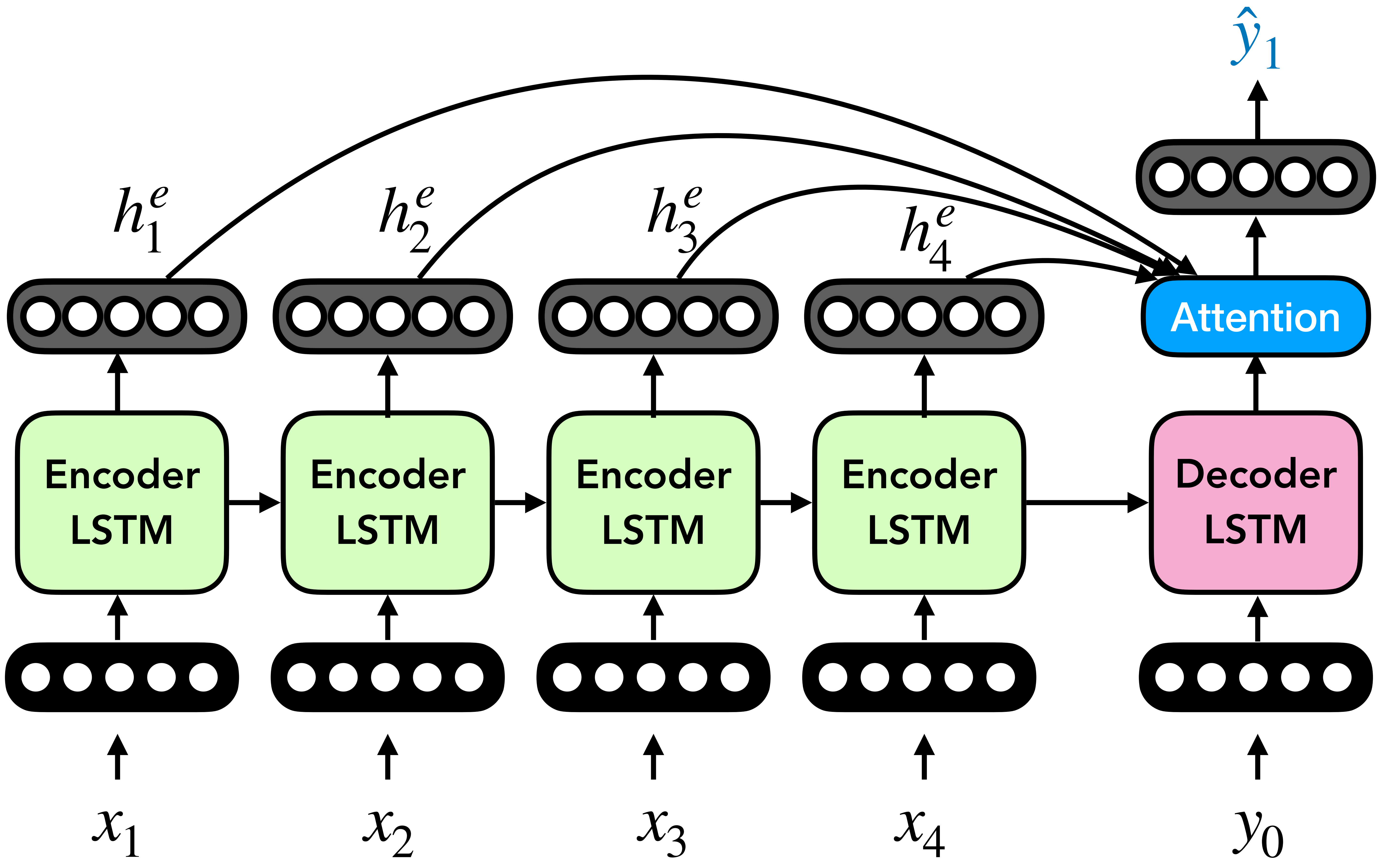
Attentive Encoder-Decoder Models



- **Recall:** At each encoder time step, there is an output of the RNN!

(Bahdanau et al., 2015)

Attentive Encoder-Decoder Models



- **Recall:** At each encoder time step, there is an output of the RNN!
- **Idea:** Use the output of the Decoder LSTM to compute an **attention** (i.e., a mixture) over all the h_t^e outputs of the encoder LSTM
- **Intuition:** focus on different parts of the input at each time step

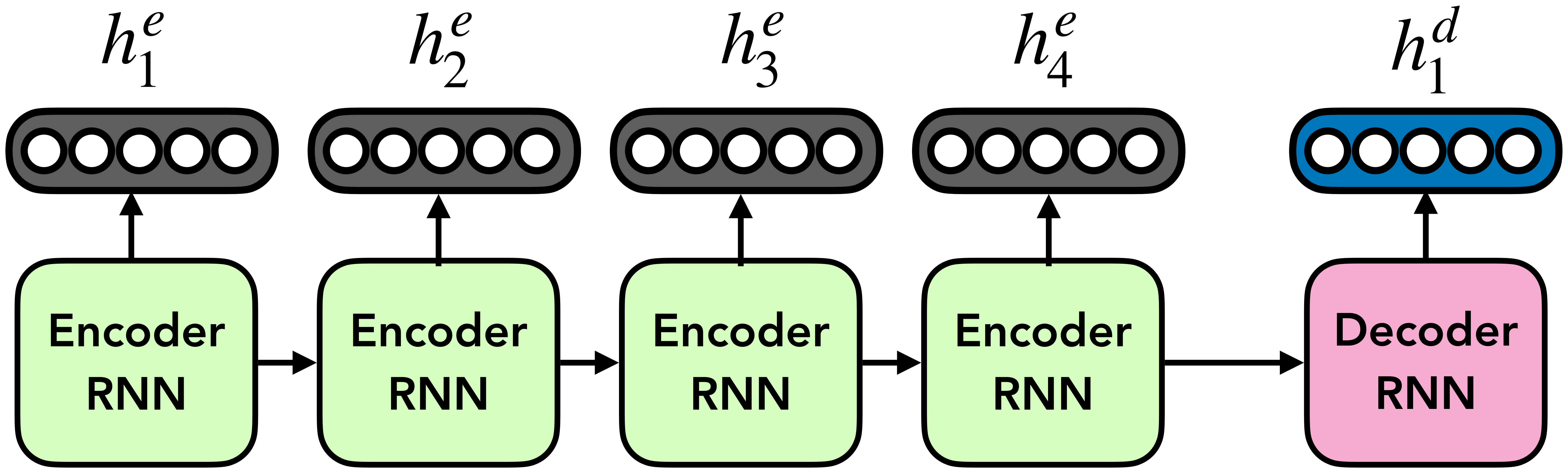
(Bahdanau et al., 2015)

What is attention?

- Attention is a **weighted average over a set of inputs**

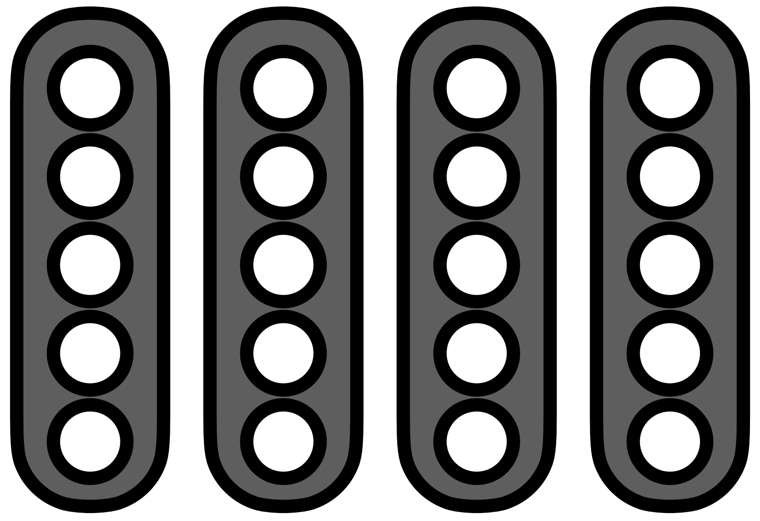
h_t^e = encoder output hidden states

- How should we compute this weighted average?



Attention Function

- **Compute** pairwise similarity between each encoder hidden state and decoder hidden state ("idea of what to decode")



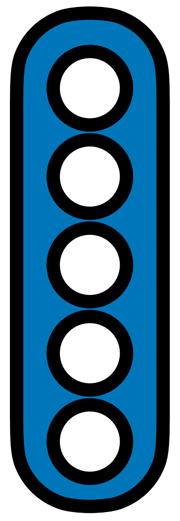
h_1^e h_2^e h_3^e h_4^e

h_t^e = encoder output hidden states

Also known as a "keys"

h_t^d = decoder output hidden state

Also known as a "query"



Attention Function

- **Compute** pairwise similarity between each encoder hidden state and decoder hidden state (“idea of what to decode”)

h_t^e = encoder output hidden states

Also known as a “keys”

h_t^d = decoder output hidden state

Also known as a “query”

- **We have a single query vector for multiple key vectors**

Attention Function

Attention Function	Formula
Multiplicative	$a = h^e \mathbf{W} h^d$
Linear	$a = v^T \phi(\mathbf{W}[h^e; h^d])$
Scaled Dot Product	$a = \dfrac{(\mathbf{W}h^e)^T(\mathbf{U}h^d)}{\sqrt{d}}$

Attention Function

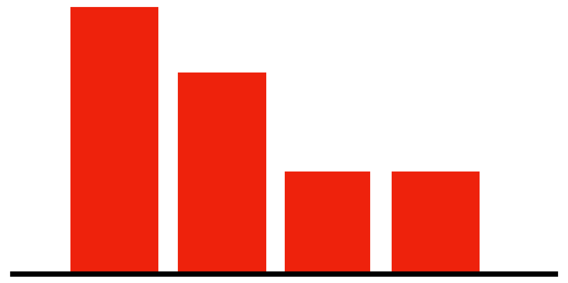
- **Compute** pairwise similarity between each encoder hidden state and decoder hidden state ("idea of what to decode")

$$a_1 = f(h_1^e, h_1^d) \qquad a_2 = f(h_2^e, h_1^d) \qquad a_3 = f(h_3^e, h_1^d)$$

- **Convert** pairwise similarity scores to probability **distribution** (using softmax!) over encoder hidden states and compute weighted average:

Softmax!

$$\boxed{\alpha_t = \frac{e^{a_t}}{\sum_j e^{a_j}}} \rightarrow$$

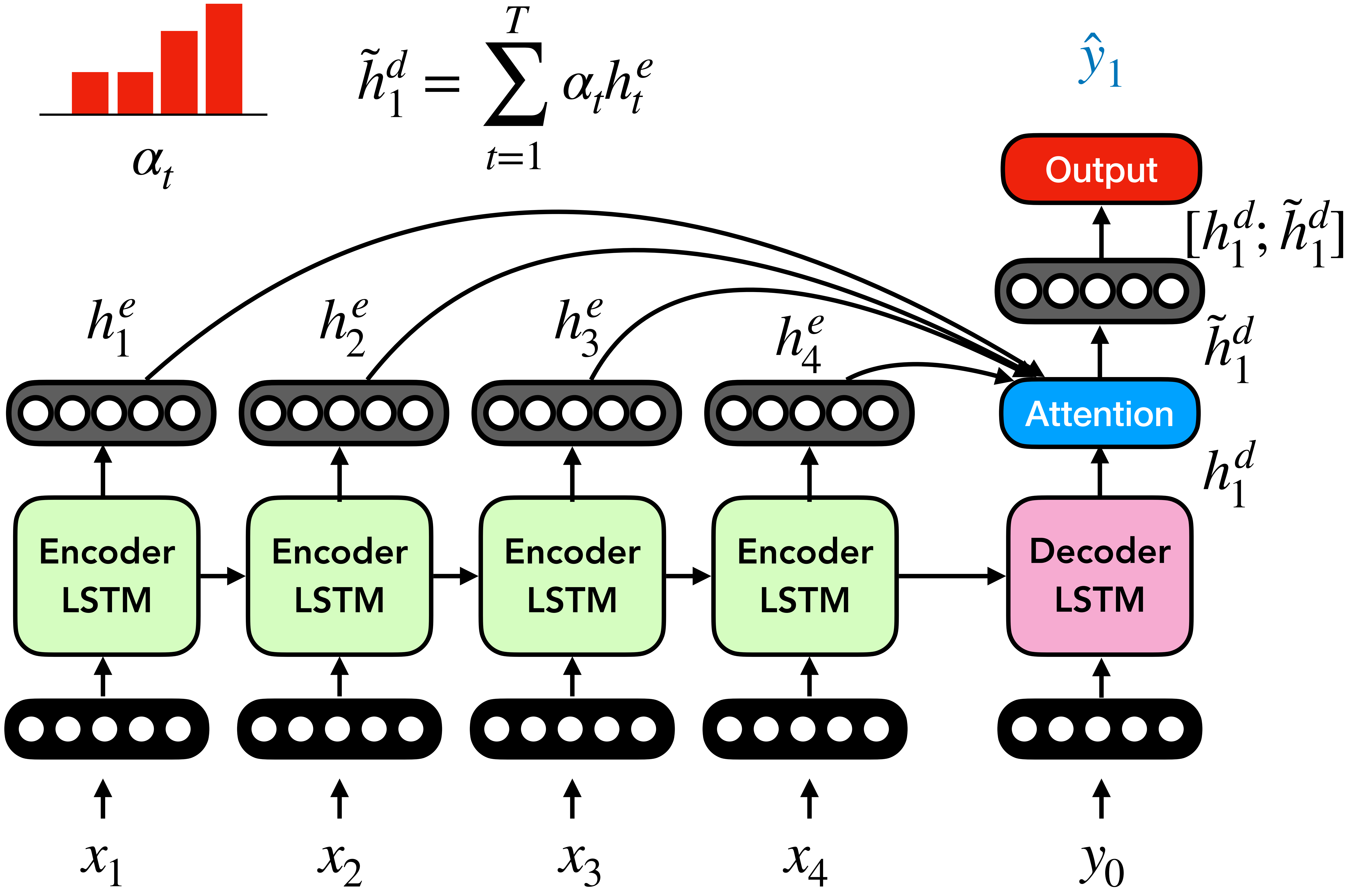


α_t

$$\rightarrow \tilde{h}_1^d = \sum_{t=1}^{T} \alpha_t h_t^e$$

Here h_t^e is known as the "value"

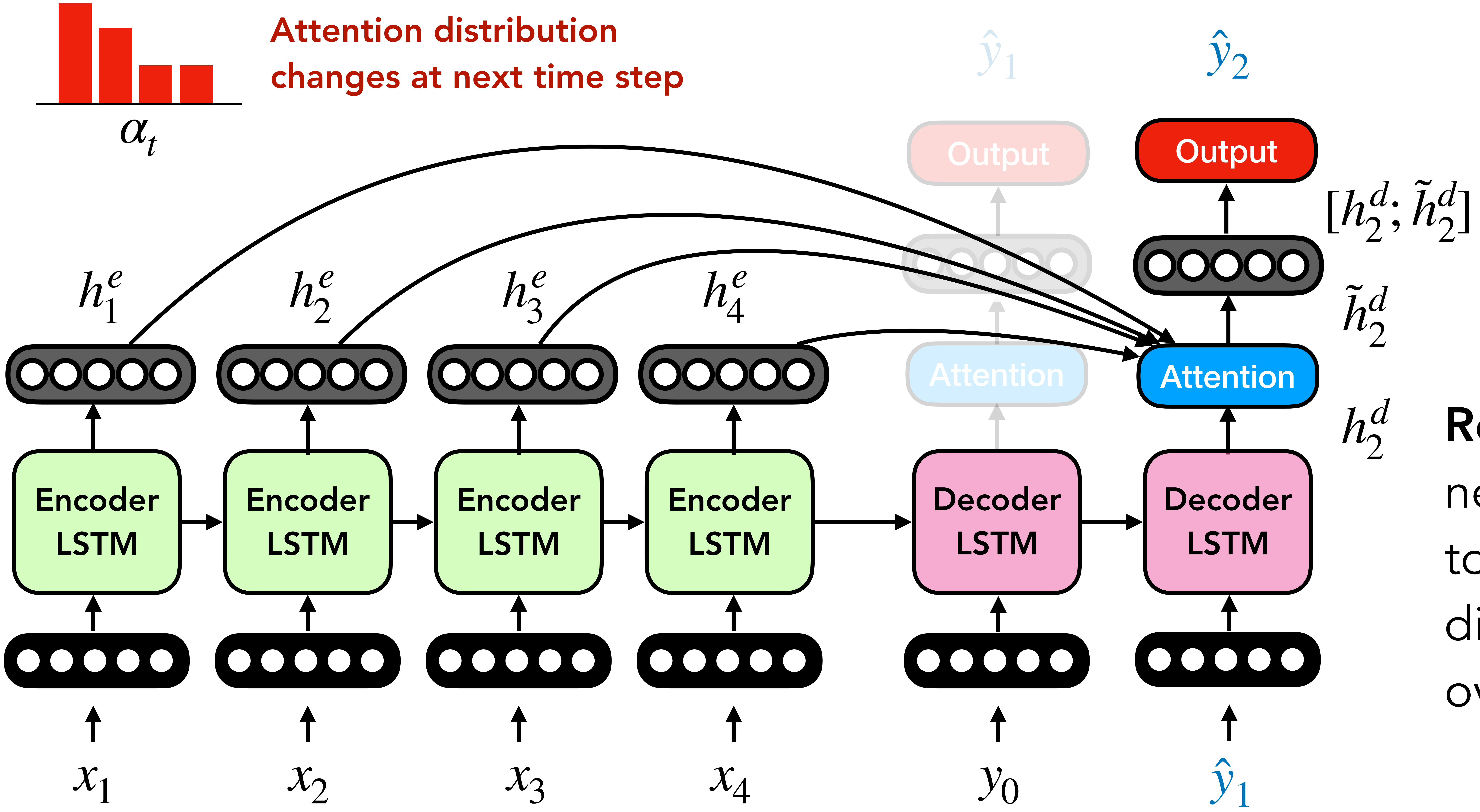
Attentive Encoder-Decoder Models



- **Intuition:** $\tilde{h}^d_1$ contains information about hidden states that got **high** attention
- Typically, $\tilde{h}^d_1$ is concatenated (or composed in some other manner) with h^d_1 (the original decoder state) before being passed to the **output** layer
- **Output** layer predicts the most likely output token $\hat{y}_1$

(Bahdanau et al., 2015)

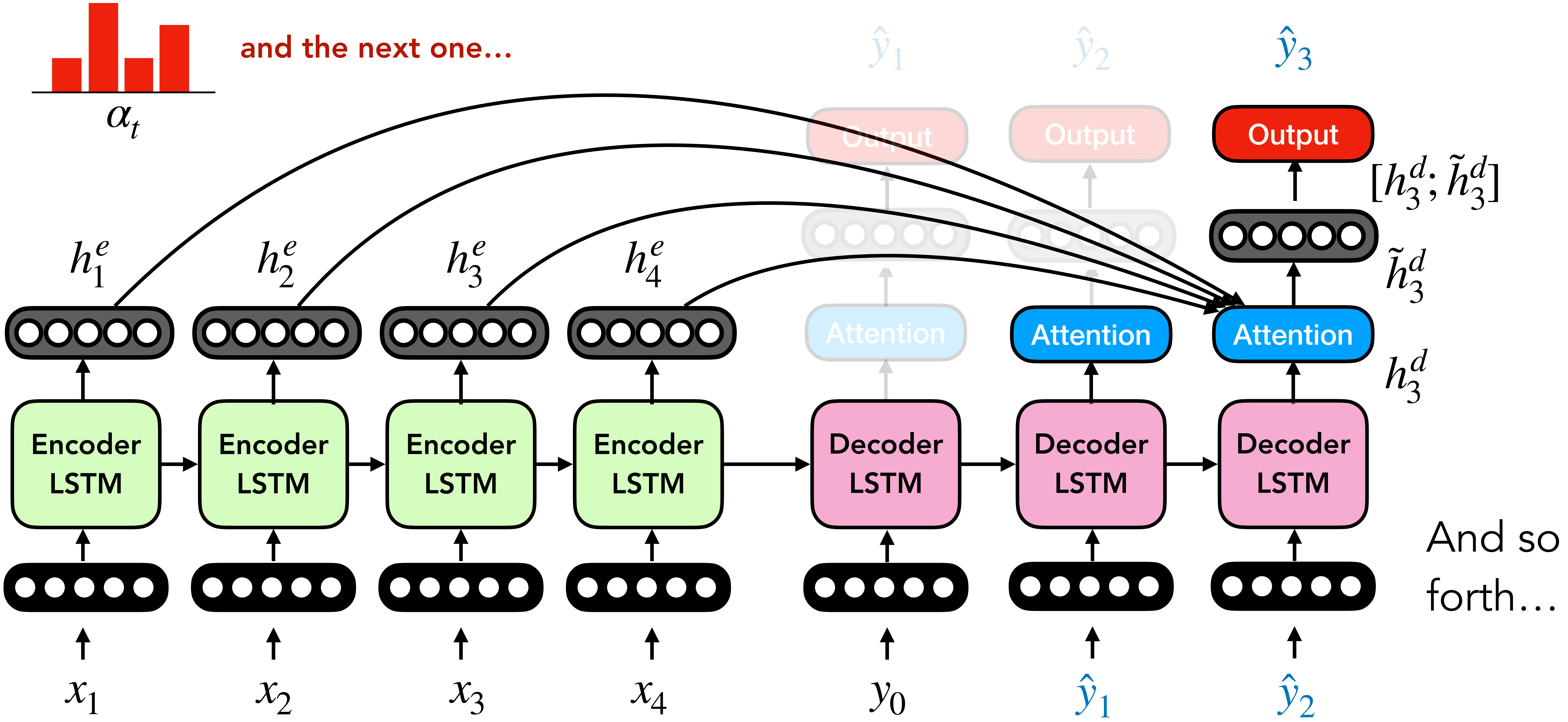
Attentive Encoder-Decoder Models



Repeat in next time step to get new distribution over states

(Bahdanau et al., 2015)

Attentive Encoder-Decoder Models



(Bahdanau et al., 2015)

Attention Recap

- **Main Idea:** Decoder computes a weighted sum of encoder outputs
- Compute pairwise score between each encoder hidden state and initial decoder hidden state

h_t^e = encoder output hidden states h_t^d = decoder initial hidden state

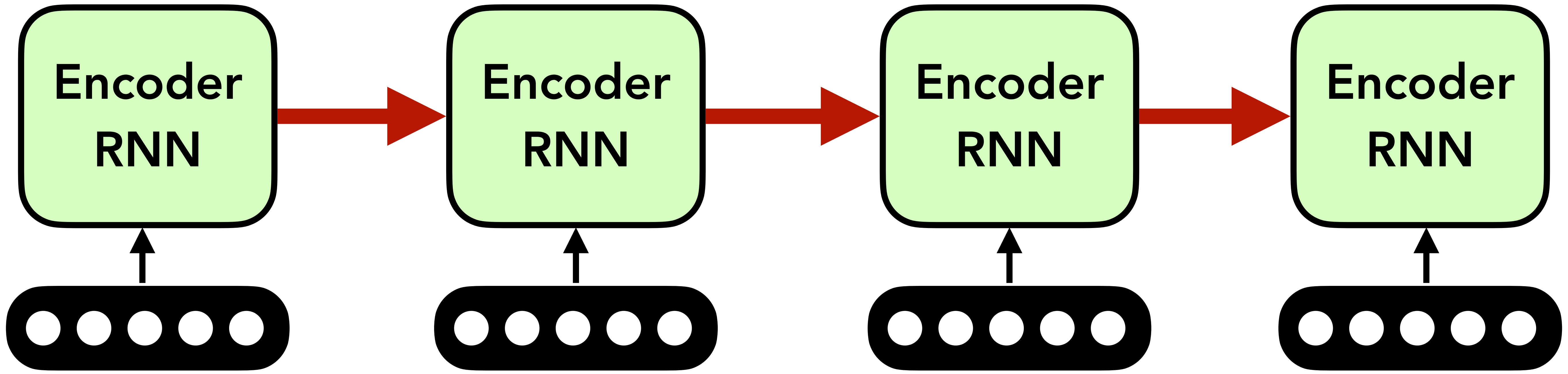
- Many possible functions for computing scores (dot product, bilinear, etc.)
- **Temporal Bottleneck Fixed! Direct connection** between decoder and **ALL** encoder states

Question

Do any other inefficiencies remain in our sequence to sequence pipelines?

Encoder is still Recurrent

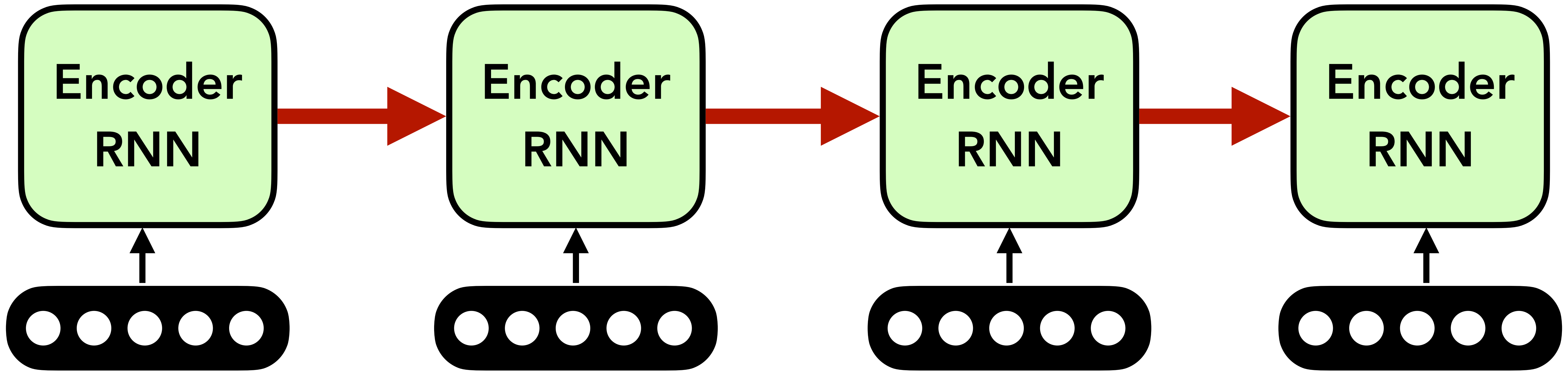
- **Encoder:** Recurrent functions can't be parallelized because previous state needs to be computed to encode next one



- **Problem: Encoder hidden states must still be computed in series**

Encoder is still Recurrent

- **Encoder:** Recurrent functions can't be parallelized because previous state needs to be computed to encode next one



- **Problem: Encoder hidden states must still be computed in series**

Who can think of a task where this might be a problem?

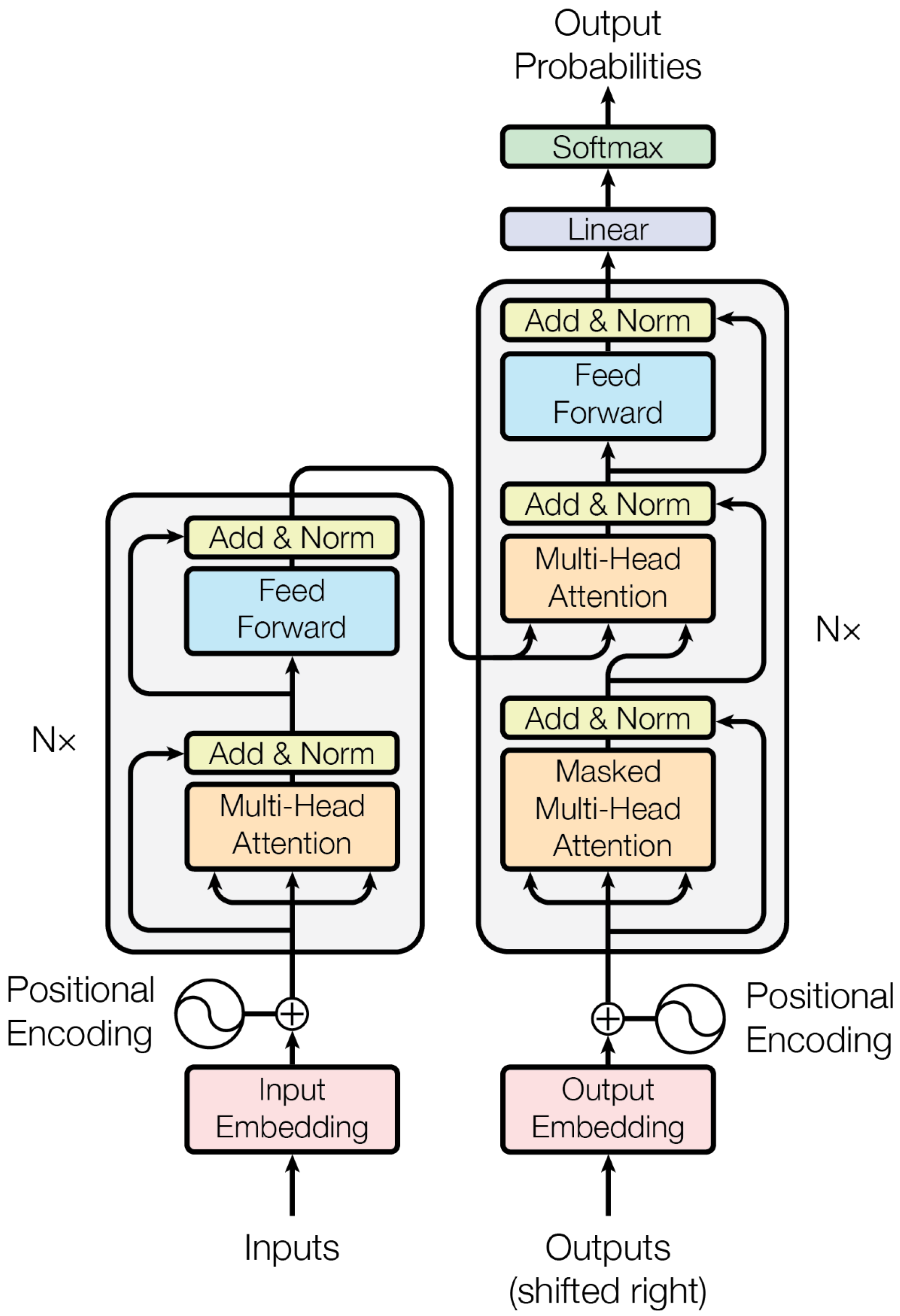
Solution:

Transformers!

Full Transformer

- Made up of encoder and decoder
- Both encoder and decoder made up of multiple cascaded transformer blocks
    - slightly different architecture in encoder and decoder transformer blocks
- Blocks generally made up **multi-headed attention** layers (self-attention) and **feedforward** layers
- No recurrent computations!

Encode sequences with self-attention

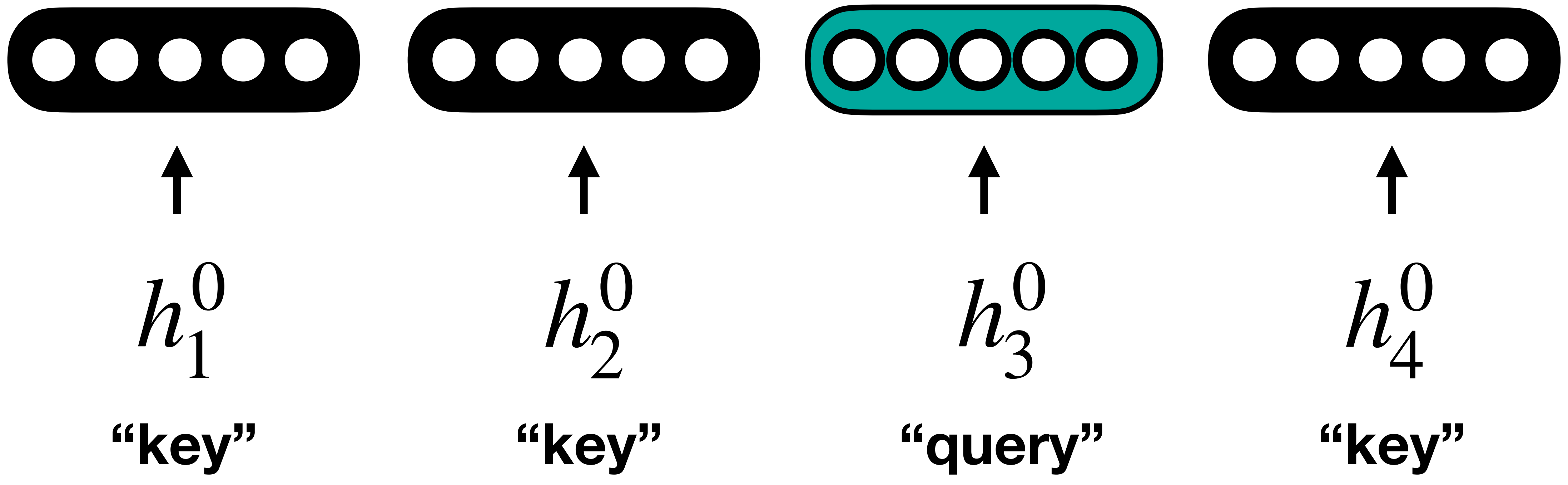


(Vaswani et al., 2017)

Self-Attention Toy Example

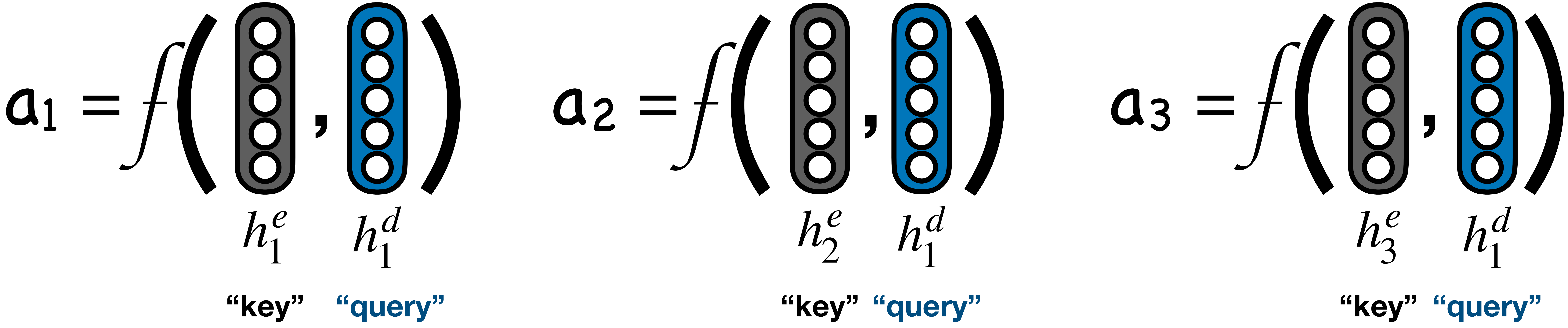
- **Original Idea:** Use decoder hidden state to compute attention distribution over encoder hidden states
- **New Idea: Could we use encoder hidden states to compute attention distribution over themselves?**
- **Ditch recurrence** and compute encoder state representations in parallel!

h_t^{ℓ} = encoder hidden state at time step t at layer ℓ



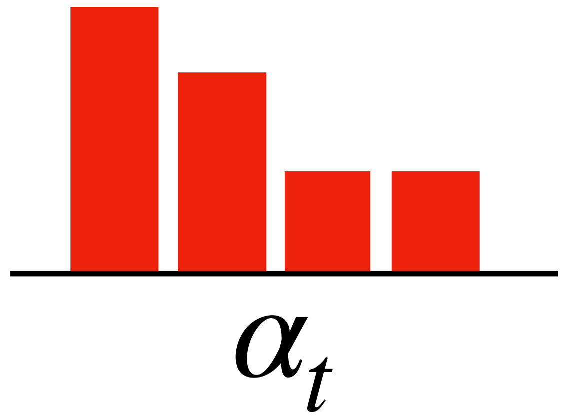
Recap: Attention with RNNs

- **Compute** pairwise similarity between each encoder hidden state and decoder hidden state ("idea of what to decode")



- **Convert** pairwise similarity scores to probability **distribution** (using softmax!) over encoder hidden states and compute weighted average:

Softmax! $\boxed{\alpha_t = \frac{e^{a_t}}{\sum_j e^{a_j}}}$ $\rightarrow$

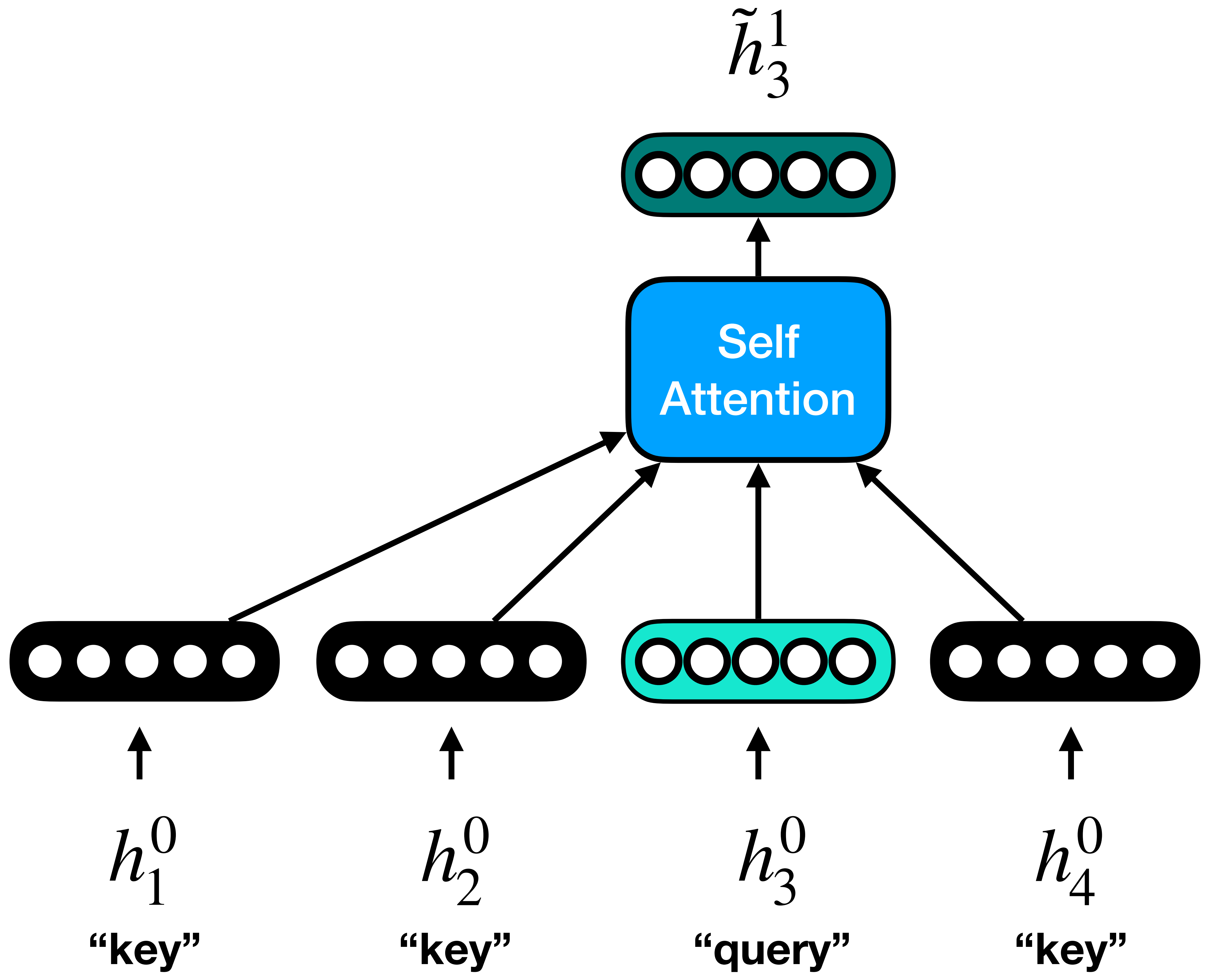


$\rightarrow$ $\tilde{h}_1^d = \sum_{t=1}^{T} \alpha_t h_t^e$

Here h_t^e is known as the "value"

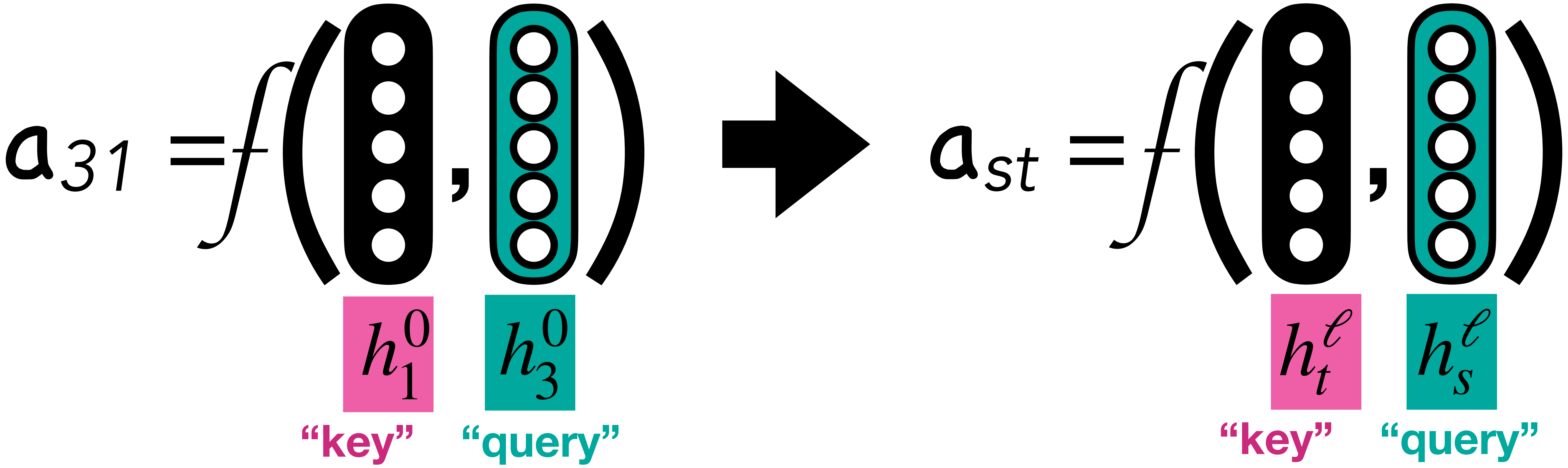
Self-Attention Toy Example

- For a particular encoder time step, compute pairwise score between this hidden state (the query) and the other encoder hidden states



Self-Attention Toy Example

h_t^ℓ = encoder hidden state at time step t at layer ℓ



$$a_{st} = \frac{(\mathbf{W}^Q h_s^\ell)^T (\mathbf{W}^K h_t^\ell)}{\sqrt{d}}$$

Compute pairwise scores

$$\alpha_{st} = \frac{e^{a_{st}}}{\sum_j e^{a_{sj}}}$$

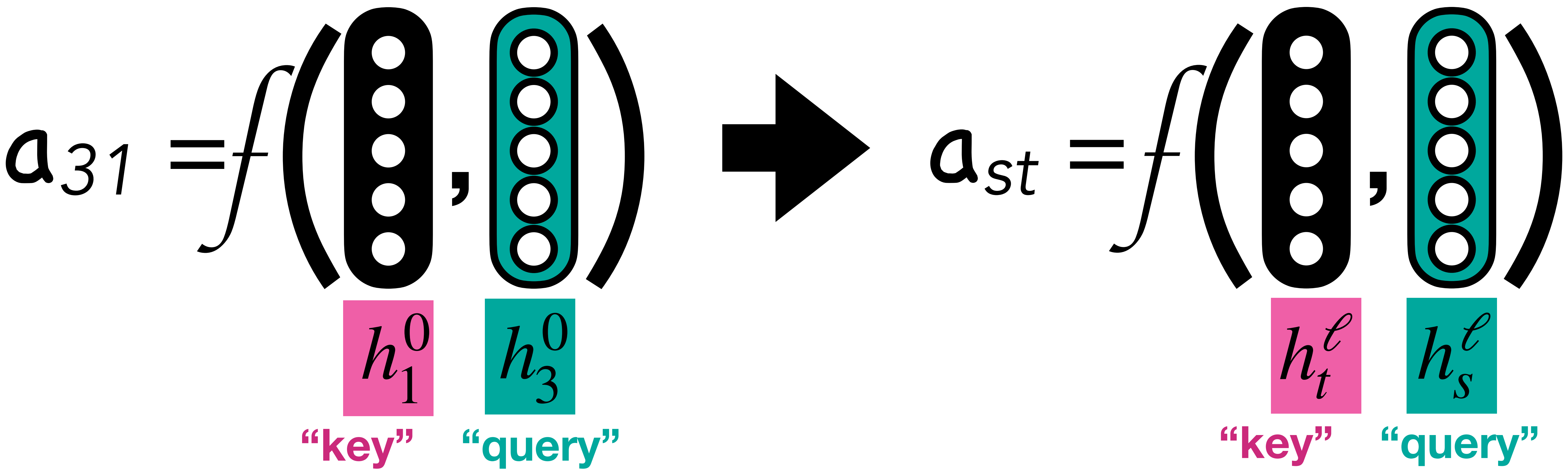
Get attention distribution

$$\tilde{h}_s^\ell = \sum_{t=1}^{T} \alpha_{st} \left(\mathbf{W}^V h_t^\ell\right)$$

Attend to values to get weighted sum

Self-Attention Toy Example

h_t^ℓ = encoder hidden state at time step t at layer ℓ



{1, …, t, …, T} includes s!

$$a_{st} = \frac{(\mathbf{W}^Q h_s^\ell)^T(\mathbf{W}^K h_t^\ell)}{\sqrt{d}}$$

Compute pairwise scores

$$\alpha_{st} = \frac{e^{a_{st}}}{\sum_j e^{a_{sj}}}$$

Get attention distribution

$$\boxed{\tilde{h}_s^\ell = \sum_{t=1}^{T} \alpha_{st}(\mathbf{W}^V h_t^\ell)}$$

Attend to values to get weighted sum

Self-attention!

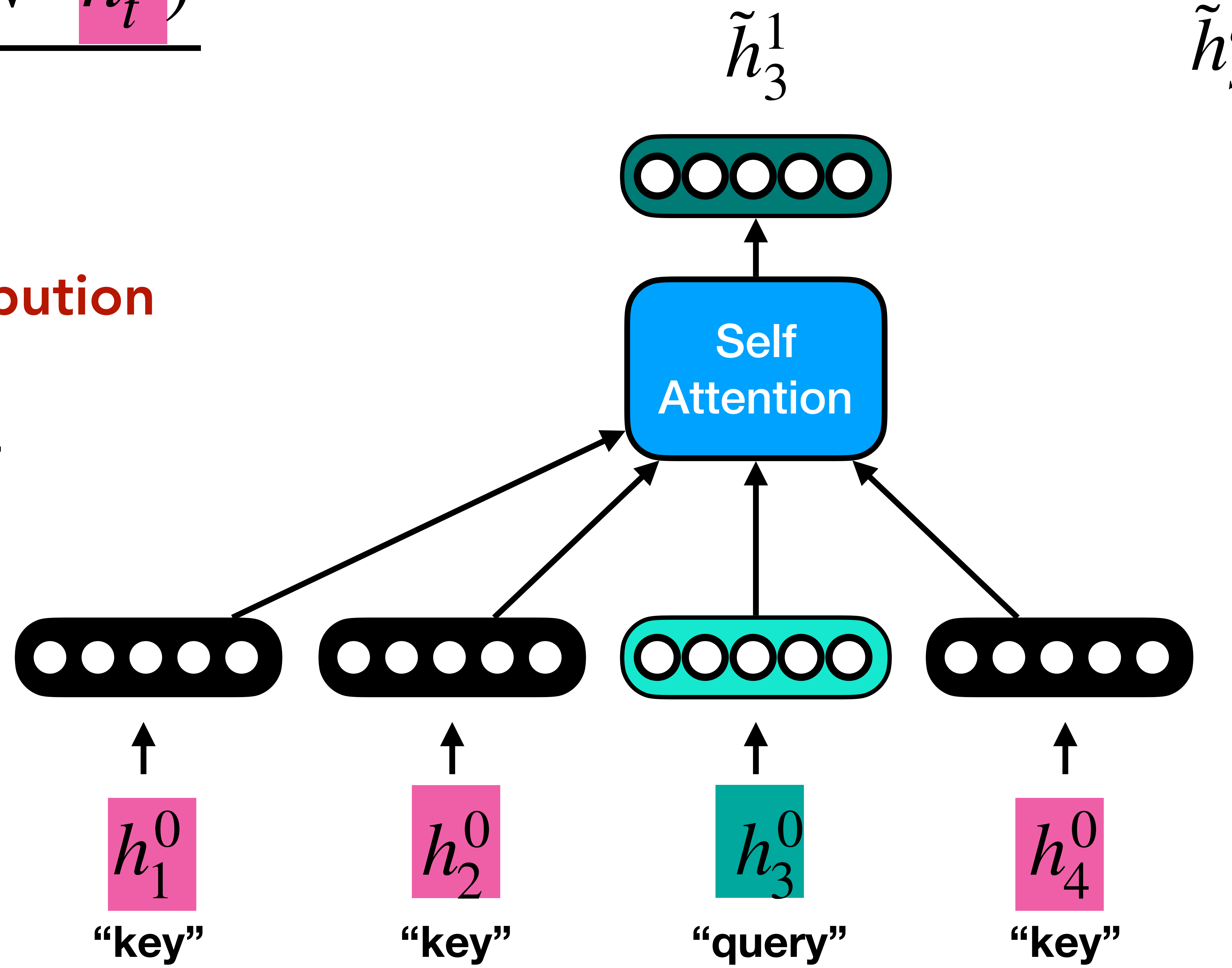
Self-Attention Toy Example

Compute pairwise scores

$$a_{st} = \frac{(\mathbf{W}^Q h_s^\ell)^T(\mathbf{W}^K h_t^\ell)}{\sqrt{d}}$$

Get attention distribution

$$\alpha_{st} = \frac{e^{a_{st}}}{\sum_j e^{a_{sj}}}$$



Attend to values to get weighted sum

$$\tilde{h}_s^\ell = \sum_{t=1}^{T} \alpha_{st}(\mathbf{W}^V h_t^\ell)$$

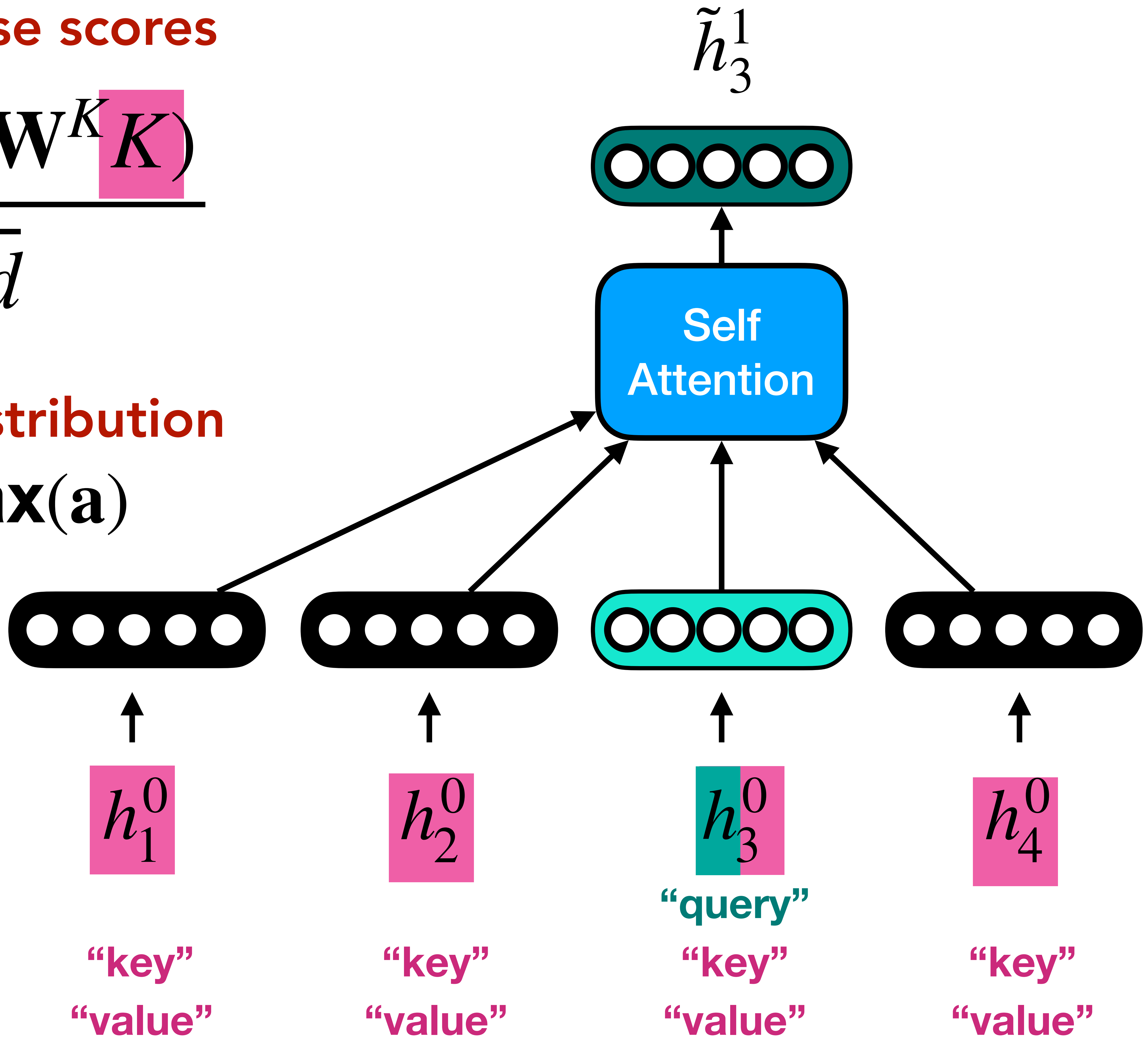
Self-Attention Toy Example

Compute pairwise scores

$$\mathbf{a} = \frac{(\mathbf{W}^Q q)(\mathbf{W}^K K)}{\sqrt{d}}$$

Get attention distribution

$$\alpha = \textbf{softmax}(\mathbf{a})$$



Attend to values to get weighted sum

$$\tilde{h}^\ell = W^O \alpha(V\mathbf{W}^V)$$

"query" $q = h_s^\ell$

"values"

$K = V = \{h_t^\ell\}_{t=0}^T$

"keys"

For each attention computation, every element is a key and value, and one element is a query

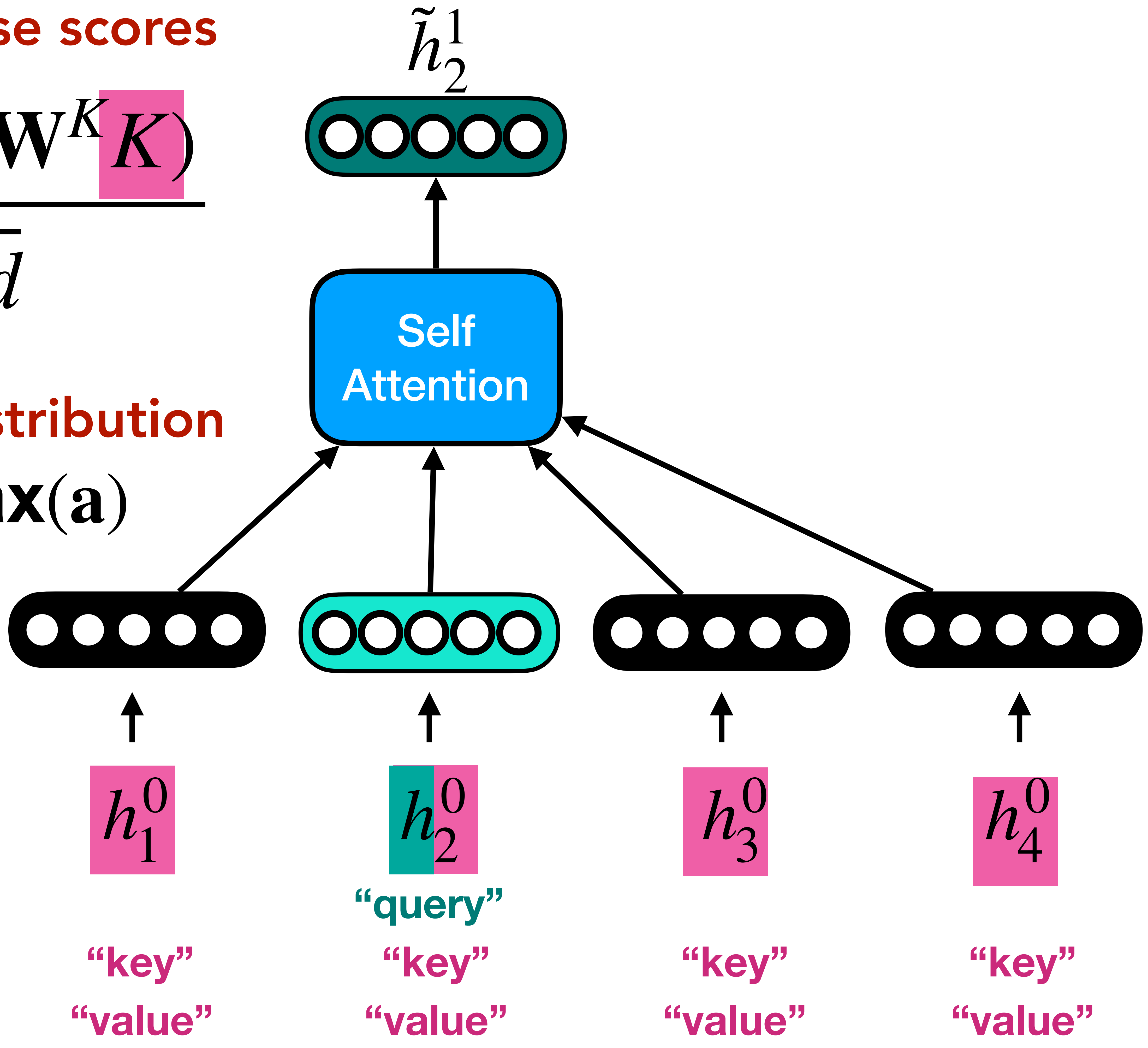
Self-Attention Toy Example

Compute pairwise scores

$$\mathbf{a} = \frac{(\mathbf{W}^Q q)(\mathbf{W}^K K)}{\sqrt{d}}$$

Get attention distribution

$$\alpha = \textbf{softmax}(\mathbf{a})$$



Attend to values to get weighted sum

$$\tilde{h}^\ell = W^O \alpha(V\mathbf{W}^V)$$

"query" $q = h_s^\ell$

"values"

$$K = V = \{h_t^\ell\}_{t=0}^T$$

"keys"

For each attention computation, every element is a key and value, and one element is a query

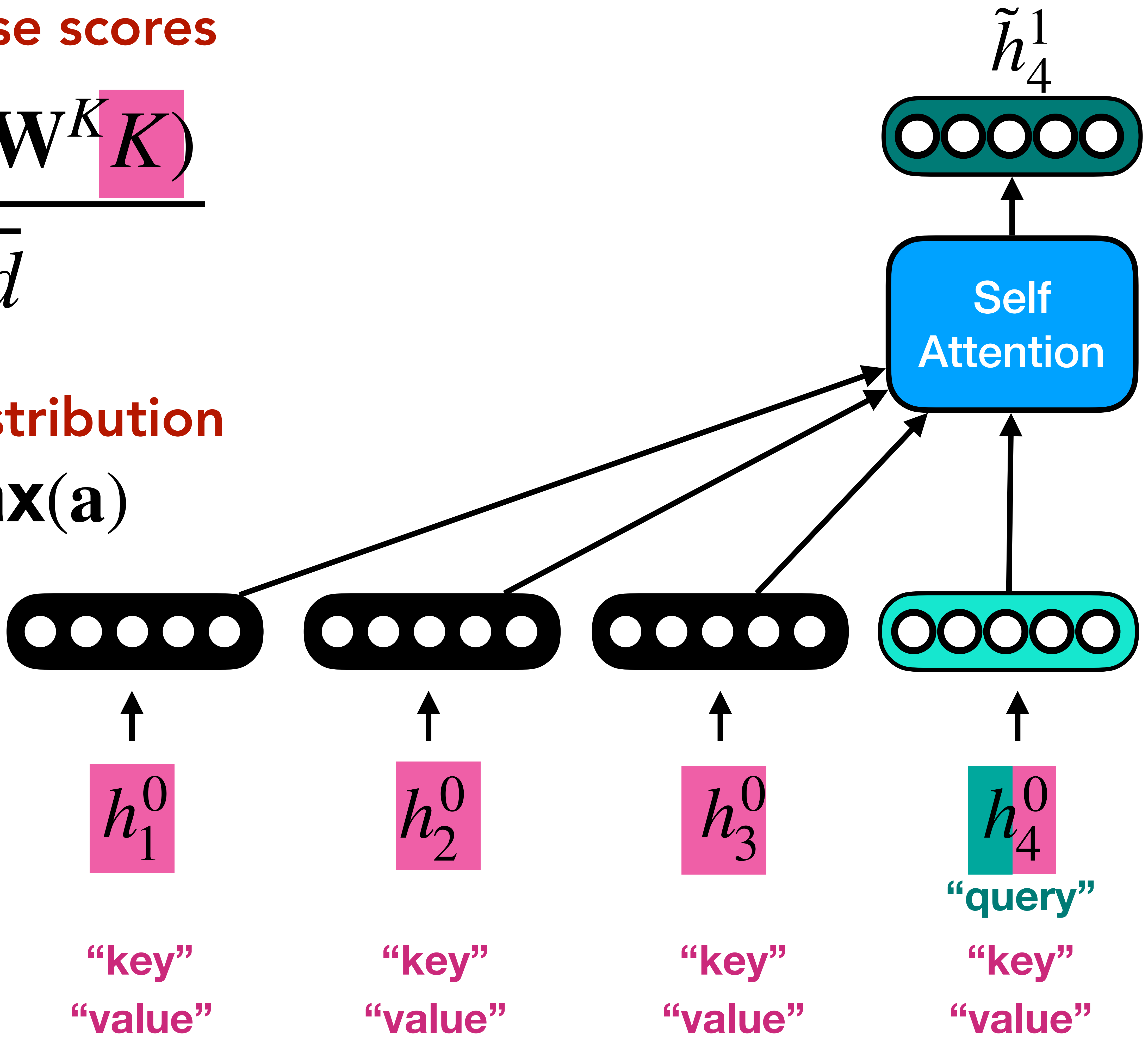
Self-Attention Toy Example

Compute pairwise scores

$$\mathbf{a} = \frac{(\mathbf{W}^Q q)(\mathbf{W}^K K)}{\sqrt{d}}$$

Get attention distribution

$$\alpha = \textbf{softmax}(\mathbf{a})$$



Attend to values to get weighted sum

$$\tilde{h}^\ell = W^O \alpha (V \mathbf{W}^V)$$

"query" $q = h_s^\ell$

"values"

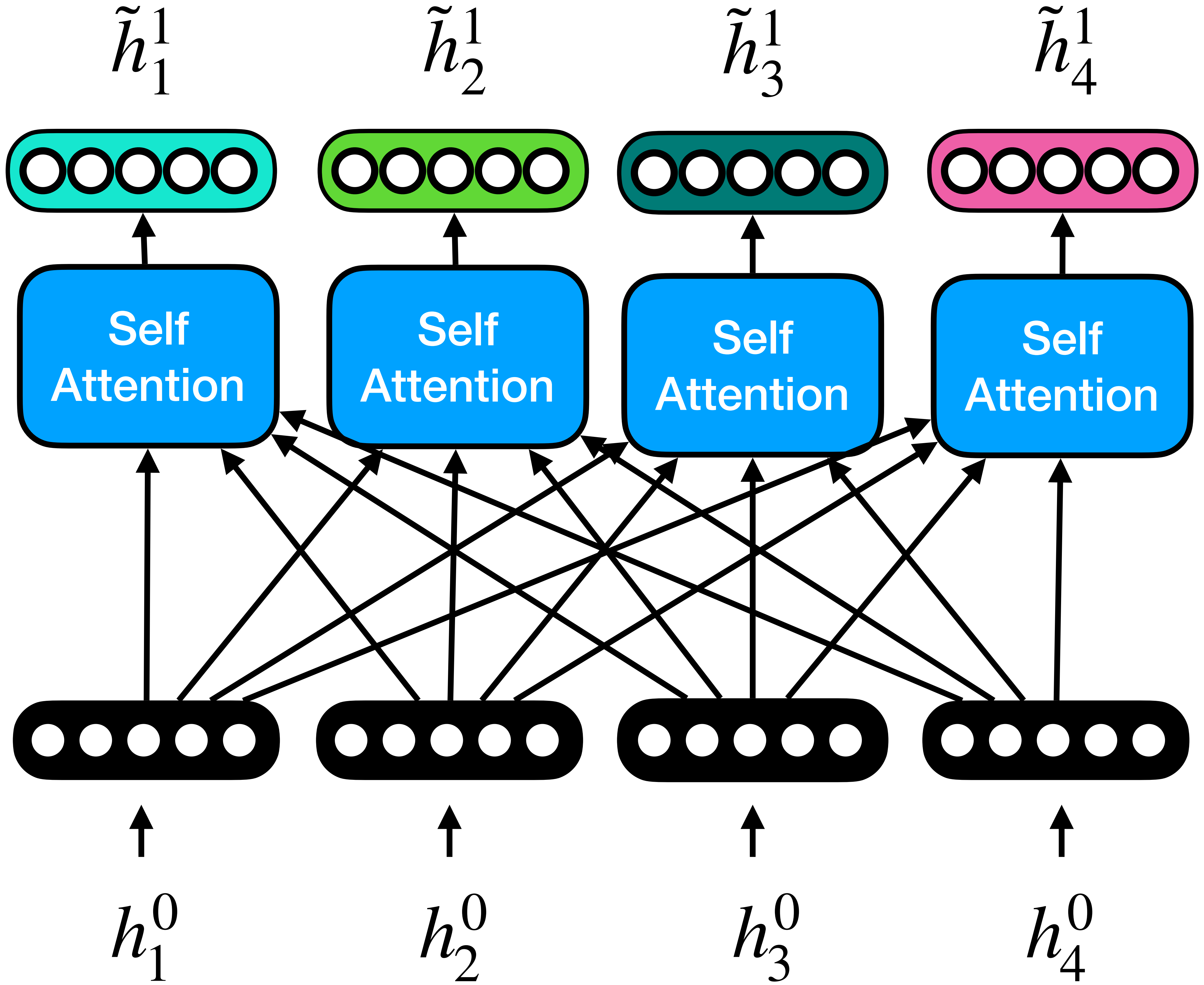
$K = V = \{h_t^\ell\}_{t=0}^T$

"keys"

For each attention computation, every element is a key and value, and one element is a query

Self-Attention Toy Example

- Every token is a query! Recompute self-attention value for each position in the sequence



$$\tilde{h}_1^1 = \textbf{Attention}\left(h_1^0, \{h_t^0\}_{t=0}^{t=3}\right)$$

$$\tilde{h}_1^2 = \textbf{Attention}\left(h_2^0, \{h_t^0\}_{t=0}^{t=3}\right)$$

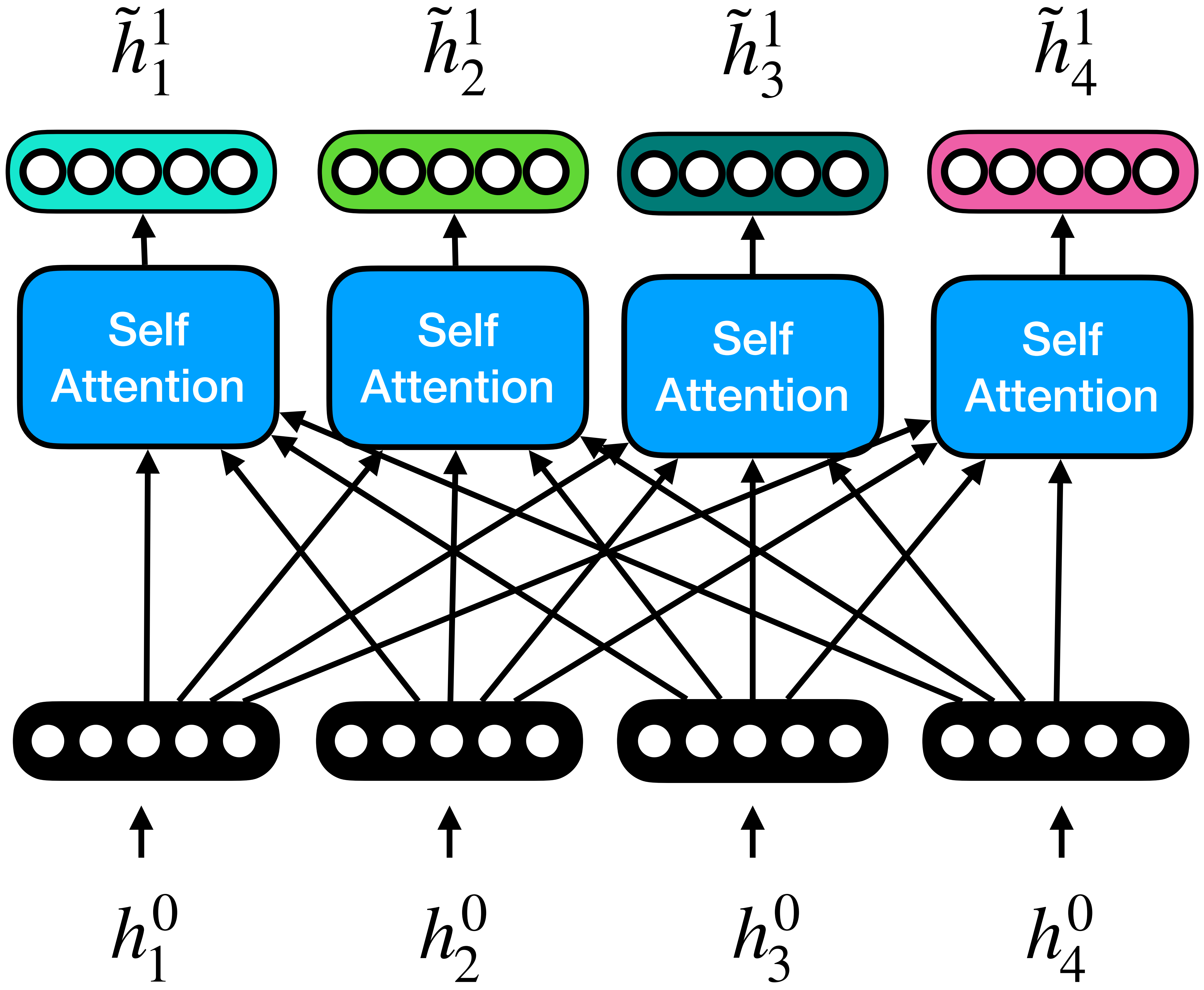
$$\tilde{h}_1^3 = \textbf{Attention}\left(h_3^0, \{h_t^0\}_{t=0}^{t=3}\right)$$

$$\tilde{h}_1^4 = \textbf{Attention}\left(h_4^0, \{h_t^0\}_{t=0}^{t=3}\right)$$

Question

What are two advantages of self-attention over recurrent models?

Self-Attention Recap



- **Computed in parallel** — no previous time step computation needed for the next one
- **No long-term dependencies** — direct connection between all time-steps in sequence

Multi-Headed Self-Attention

- Project V, K, Q into H sub-vectors where H is the number of "heads"

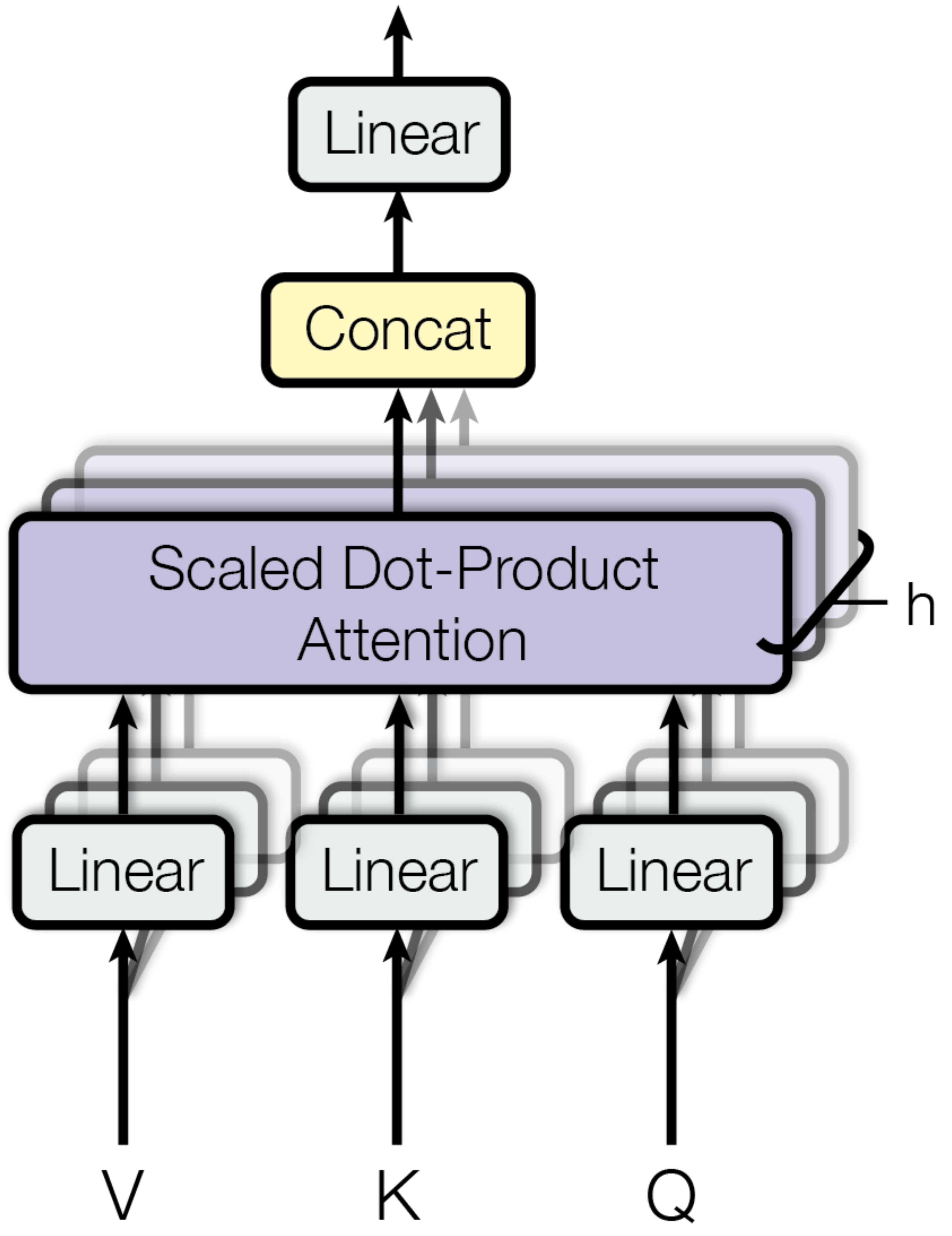
$$\mathbf{a}_i = \frac{(\mathbf{W}_i^Q q)(\mathbf{W}_i^K K)}{\sqrt{d/H}}$$

- Compute attention weights separately for each sub-vector

$$\alpha_i = \textbf{softmax}(\mathbf{a}_i) \qquad \tilde{h}_i^\ell = \alpha\left(V\mathbf{W}_i^V\right)$$

- Concatenate sub-vectors for each head and project

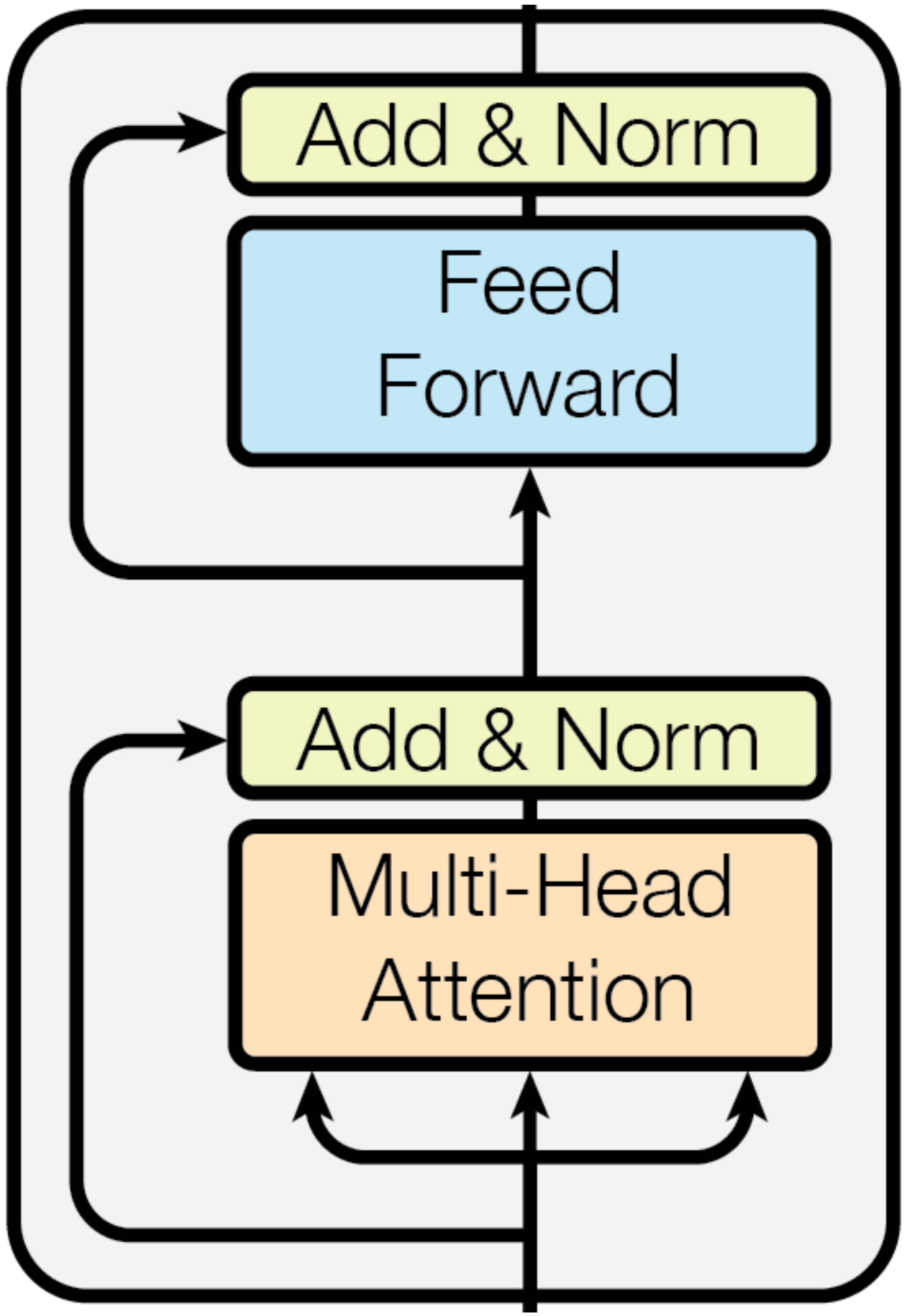
$$\tilde{h}^\ell = W^O[\ \tilde{h}_0^\ell;\ \ldots;\tilde{h}_i^\ell;\ \ldots;\tilde{h}_H^\ell\ ]$$



Vaswani et al., 2017

Transformer Block

- Self-attention is the main innovation of the popular **transformer** model!
- Each transformer block receives as input the outputs of the previous layer at every time step
- Each block is composed of a multi-headed attention, a layer normalisation, a feedforward network, and another layer normalisation
- There are residual connections before every normalisation layer
- Layer normalisation + residual connections don't add capacity, but make training easier



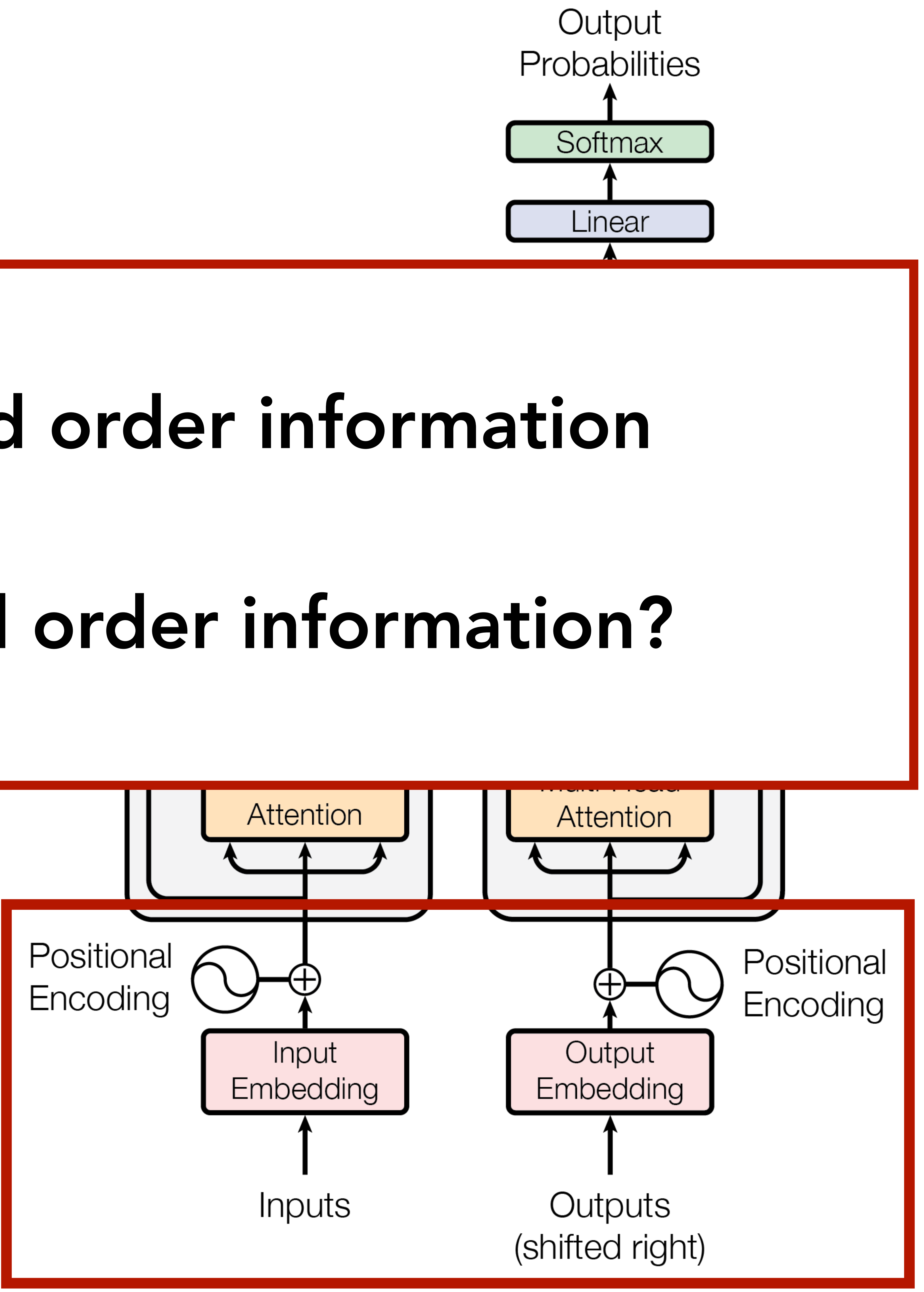
Vaswani et al., 2017

Full Transformer

- Full transformer encoder is multiple cascaded transformer blocks
  -
- N
  -
  parallel.

Recurrent models provided word order information

Does self-attention provide word order information?

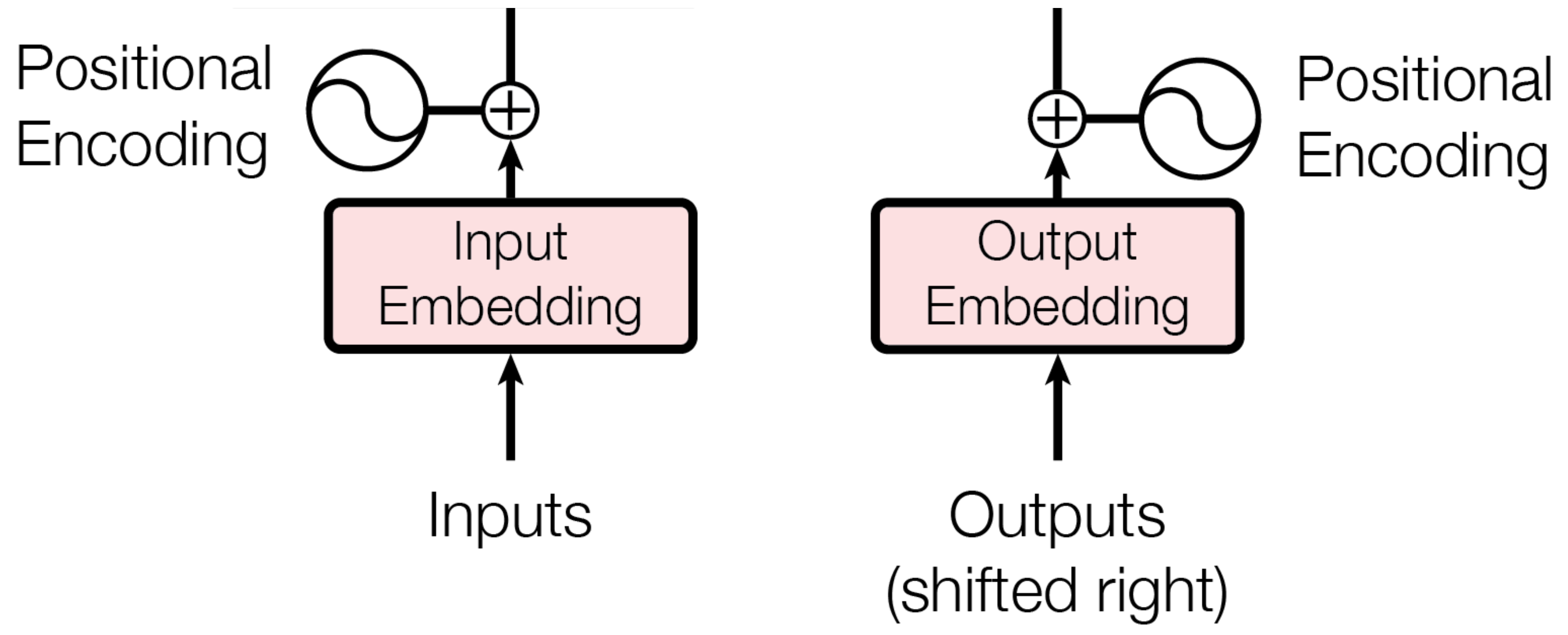


- Transformer decoder (right) similar to encoder
  - second attention layer to compute weighted average of encoder states before FFN

Vaswani et al., 2017

Position Embeddings

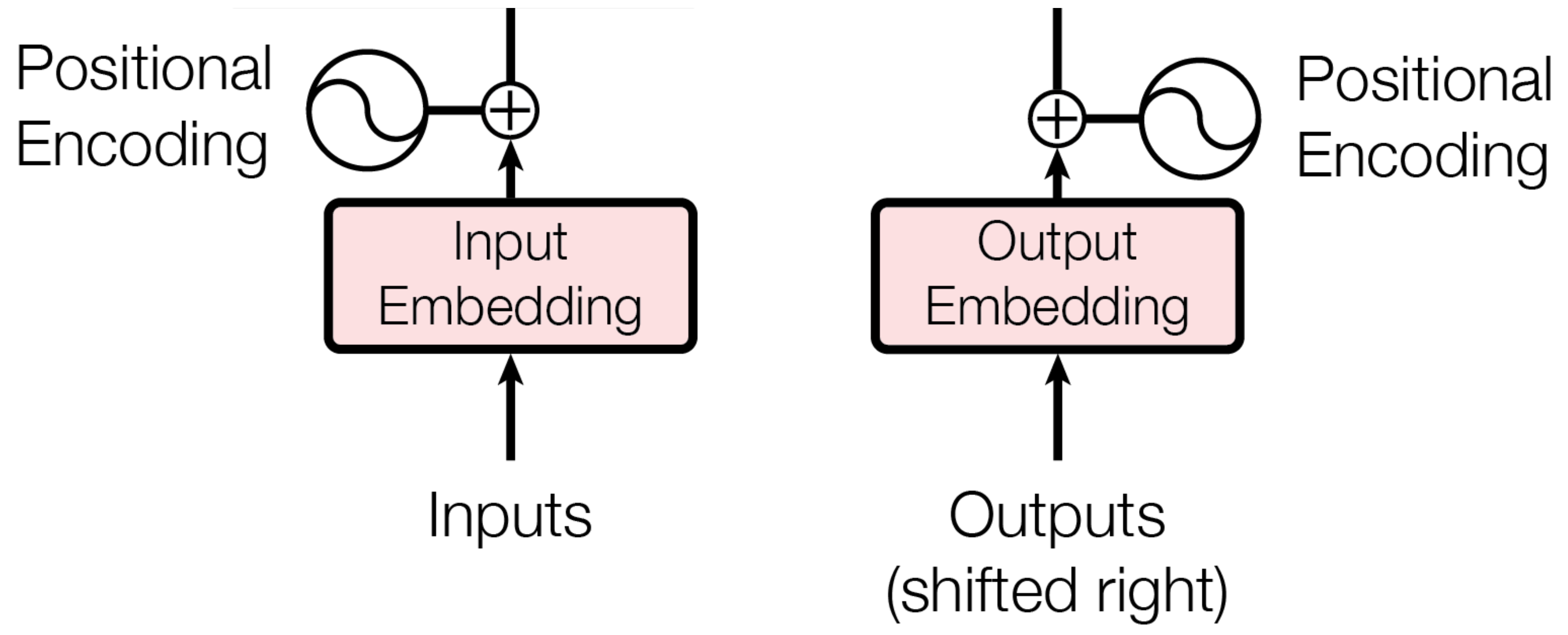
- Self-attention provides no word order information
  - Computes weighted average over set of vectors
- Word order is pretty crucial to understanding language
  - How do we fix this?
- Add an additional embedding to the input word that represents a position in the sequence



Vaswani et al., 2017

Position Embeddings

- Self-attention provides no word order information
  - Computes weighted average over set of vectors
- Word order is pretty crucial to understanding language
  - How do we fix this?
- Add an additional embedding to the input word that represents a position in the sequence



- Early position embeddings encoded a sinusoid function that was offset by a phase shift proportional to sequence position
- **In practice, position embeddings are learned scratch or more modern methods are used (e.g., Rotary position embeddings, AliBi)**

Vaswani et al., 2017

Other Resources of Interest

- The Annotated Transformer
  - https://nlp.seas.harvard.edu/2018/04/03/attention.html
- The Illustrated Transformer
  - https://jalammar.github.io/illustrated-transformer/
- Only basics presented here today! Many modifications to initial transformers exist

Recap

- **Temporal Bottleneck**: **Vanishing gradients** stop many RNN architectures from learning **long-range dependencies**

- **Parallelisation Bottleneck:** RNN states depend on previous time step hidden state, so must be **computed in series**

- **Attention**: Direct connections between output states and inputs (solves temporal bottleneck)

- **Self-Attention**: Remove recurrence, allowing parallel computation

- Modern **Transformers** use attention as primary function, but require position embeddings to capture sequence order

References

- Paperno, D., Kruszewski, G., Lazaridou, A., Pham, Q.N., Bernardi, R., Pezzelle, S., Baroni, M., Boleda, G., & Fernández, R. (2016). The LAMBADA dataset: Word prediction requiring a broad discourse context. *ArXiv, abs/1606.06031*.

- Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural Machine Translation by Jointly Learning to Align and Translate. *CoRR, abs/1409.0473*.

- Vaswani, A., Shazeer, N.M., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., & Polosukhin, I. (2017). Attention is All you Need. *ArXiv, abs/1706.03762*.

Deep Learning for

Natural Language Processing

Antoine Bosselut

Part 2: Recurrent Neural Networks for Sequence Modeling

Section Outline

- **Background**: Language Modeling, Feedforward Neural Networks, Backpropagation
- **Content - Models:** Recurrent Neural Networks, Encoder-Decoders
- **Content - Algorithms:** Backpropagation through Time, Vanishing Gradients

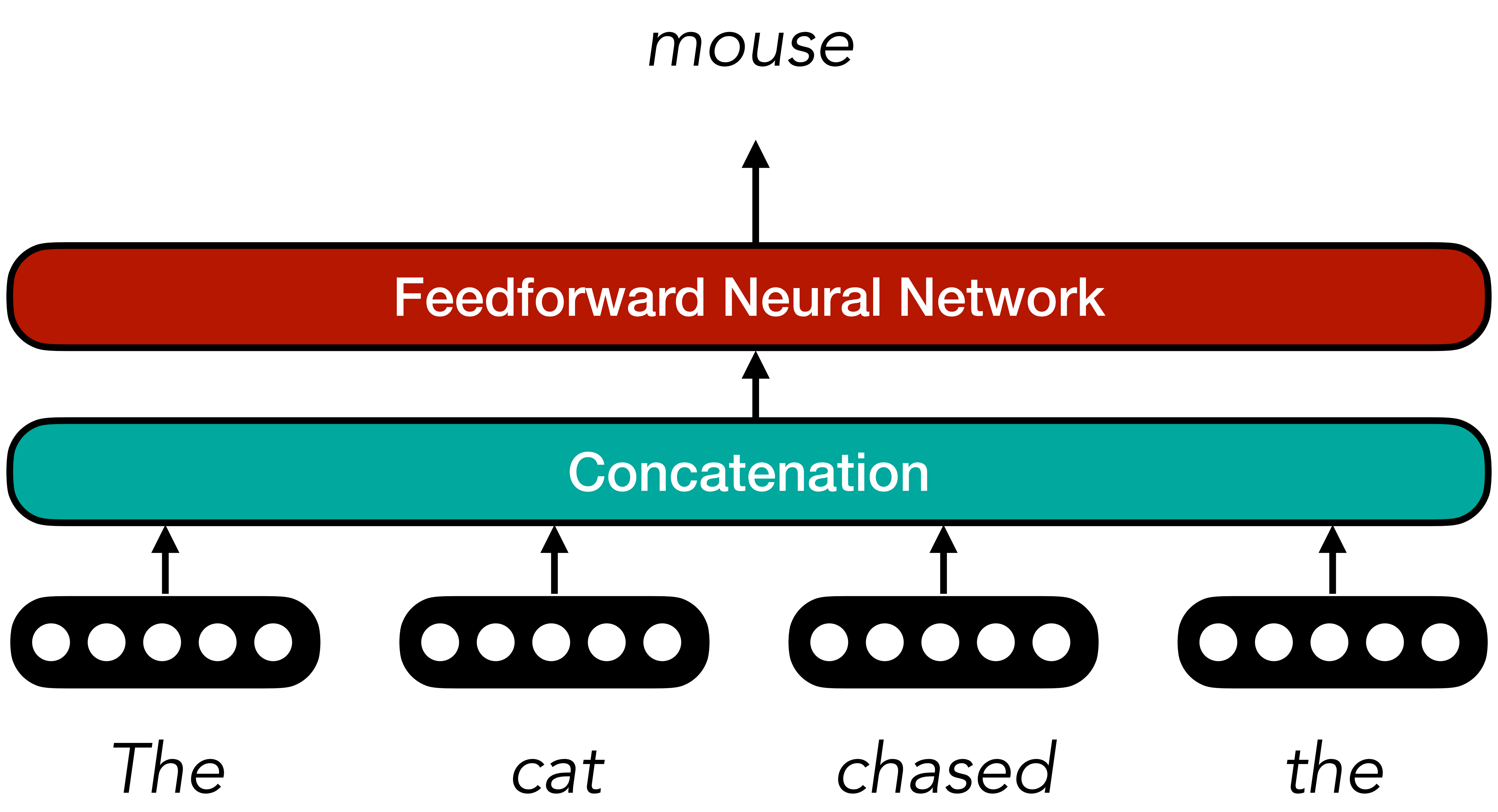
Language Modeling

- Given a subsequence, predict the next word: *The cat chased the* _____

Fixed Context Language Models

- Given a subsequence, predict the next word: *The cat chased the* _____

$$P(y) = \textbf{softmax}\left(b_o + \mathbf{W}_o \, \textbf{tanh}(b_h + \mathbf{W}_h x)\right)$$

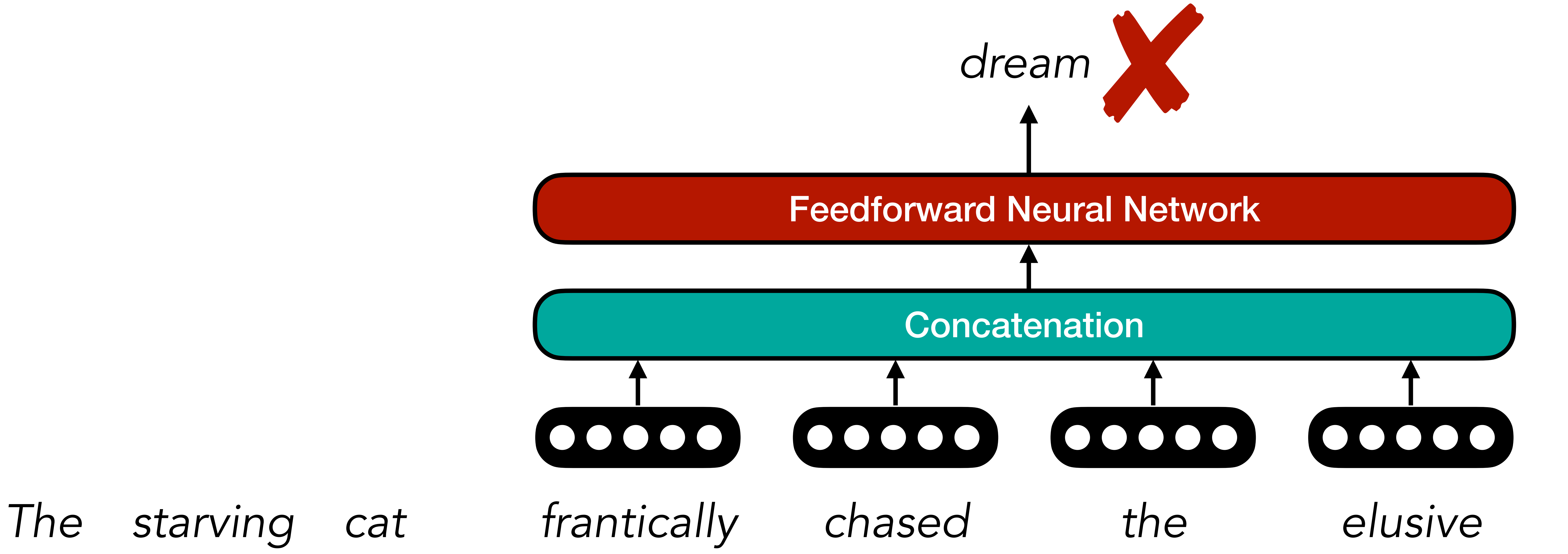


(Bengio et al., 2003)

Fixed Context Language Models

- Given a subsequence, predict the next word:

The starving cat frantically chased the elusive _____



(Bengio et al., 2003)

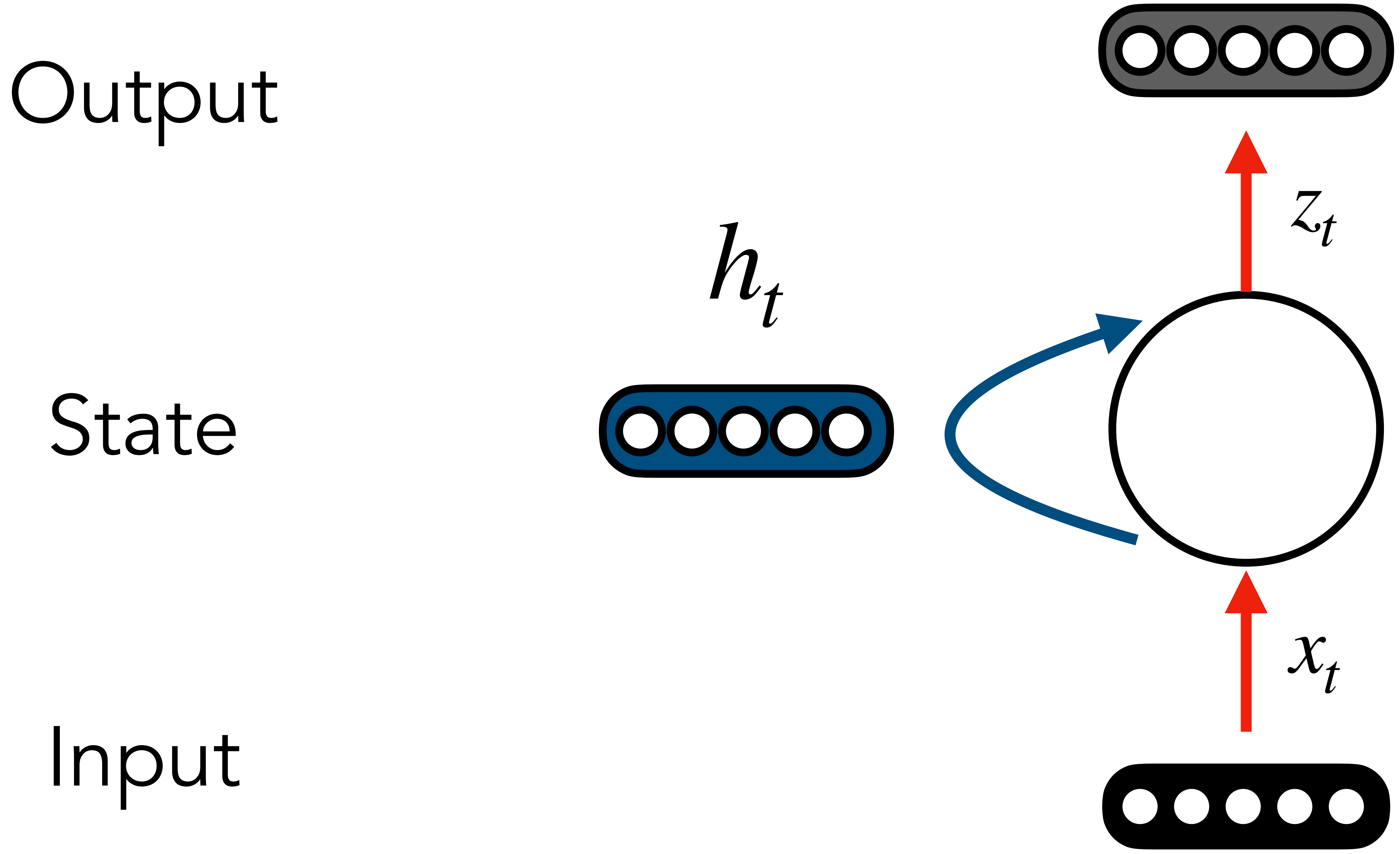
Problem

Fixed context windows limit language modelling capacity

How can we extend to arbitrary length sequences?

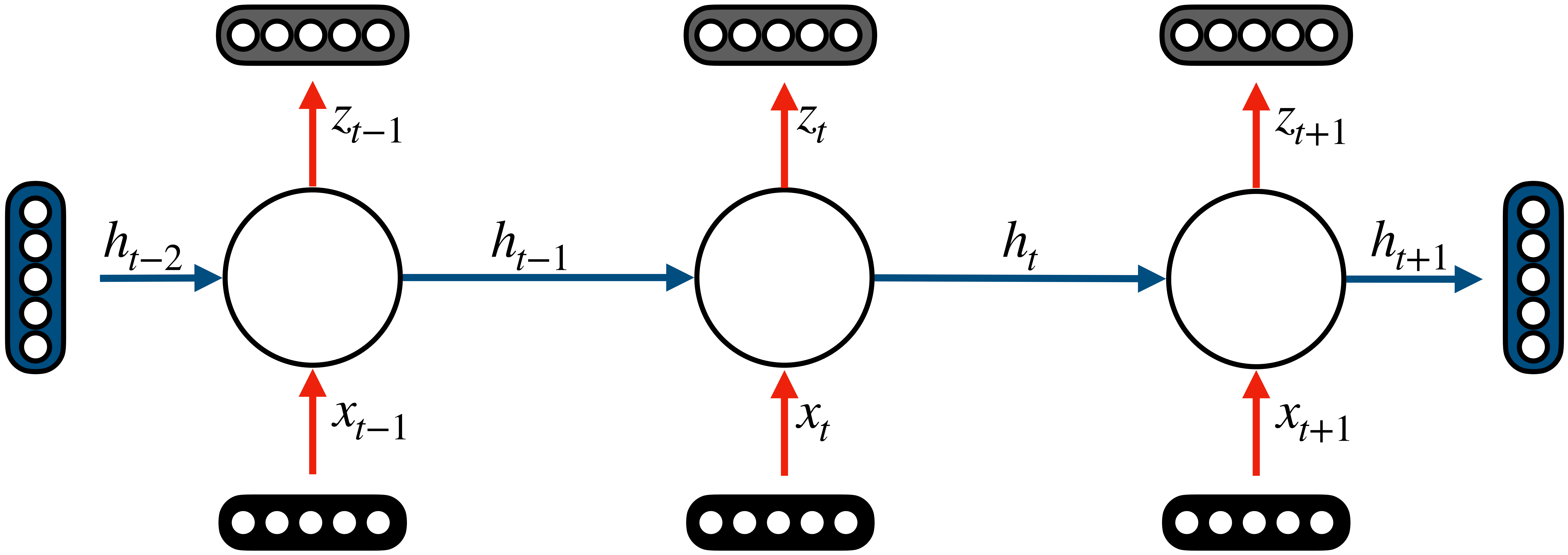
Recurrent Neural Networks

- **Solution:** Recurrent neural networks — NNs with feedback loops



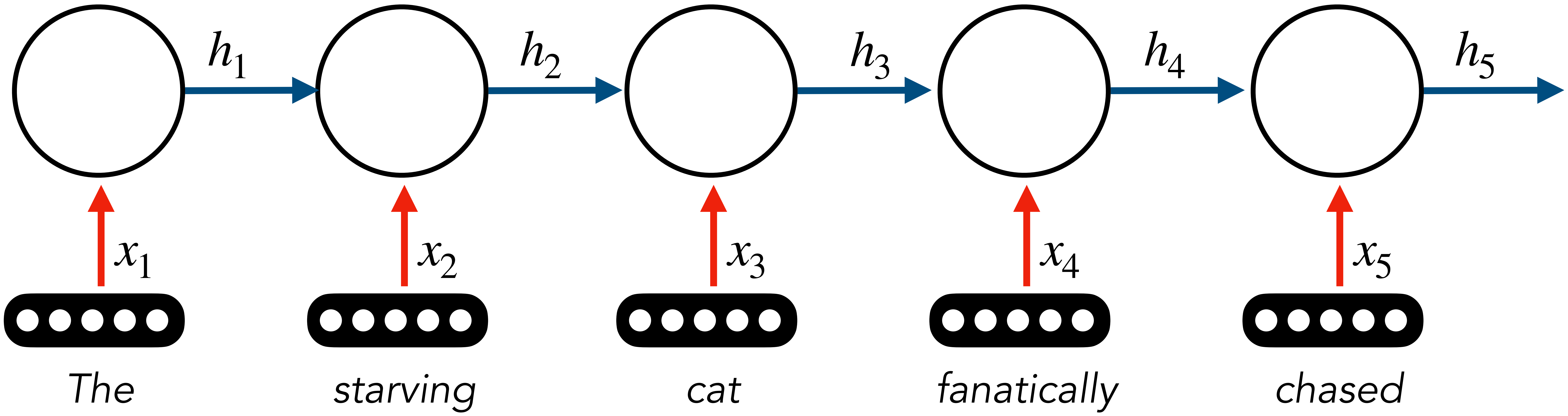
Unrolling the RNN

Unrolling the RNN across all time steps gives full computation graph

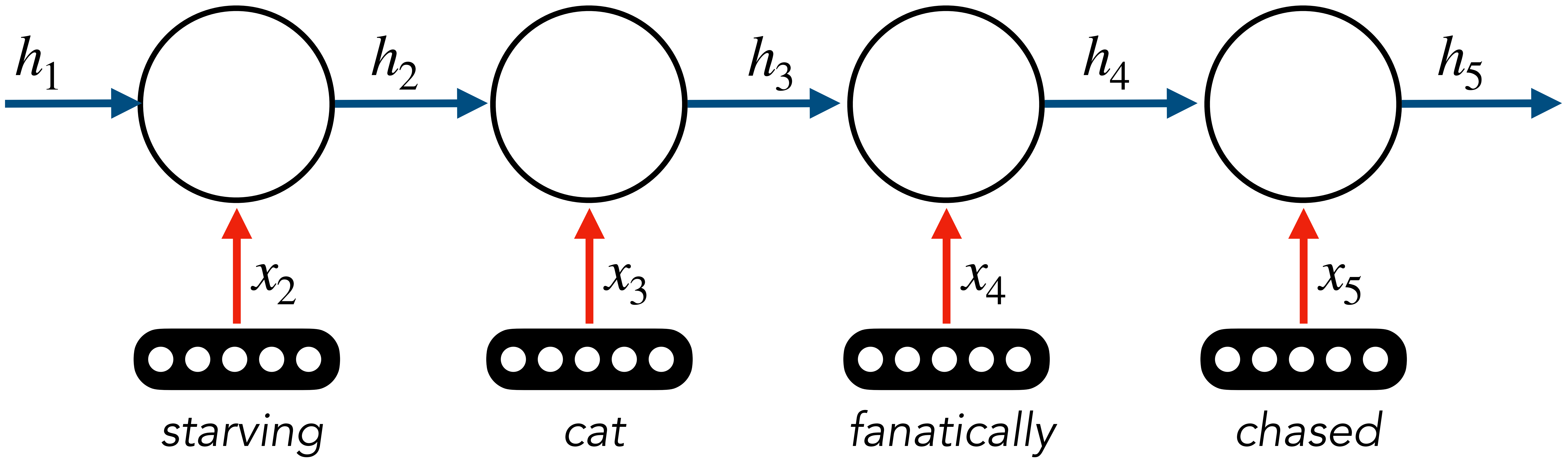


Allows for learning from entire sequence history, regardless of length

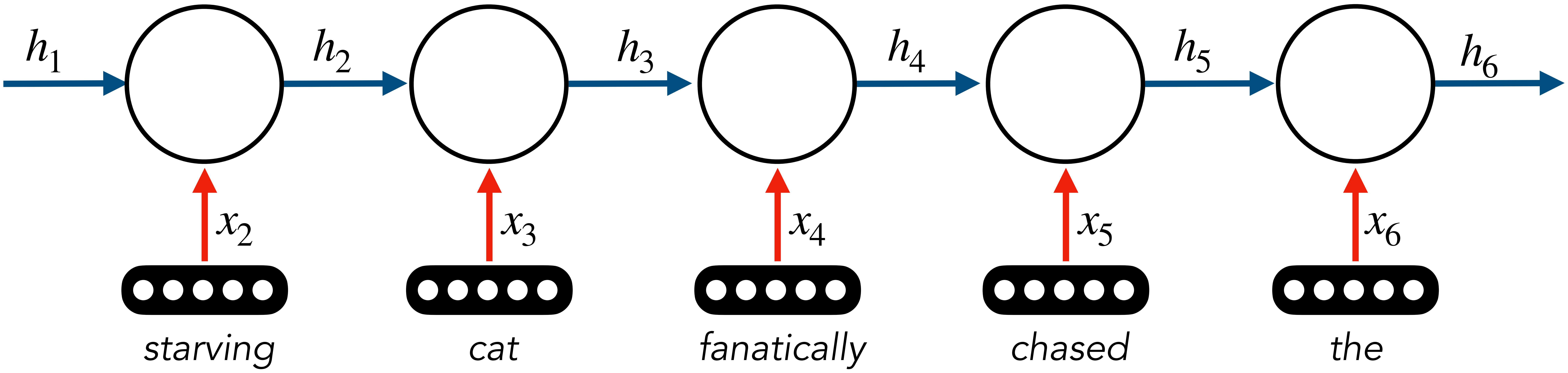
Unrolling the RNN



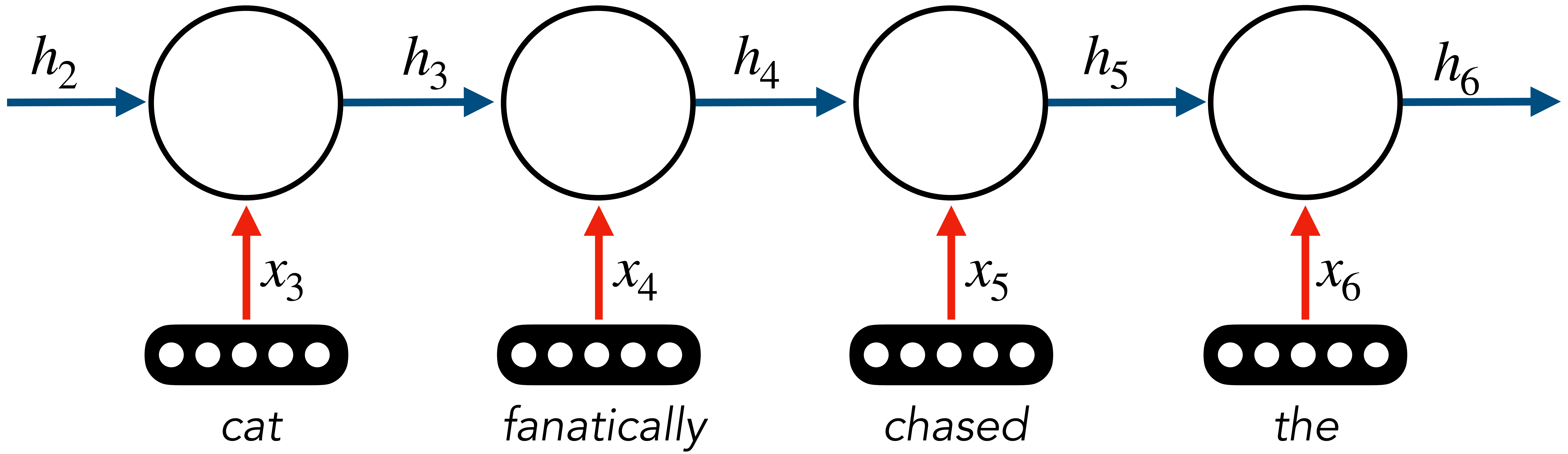
Unrolling the RNN



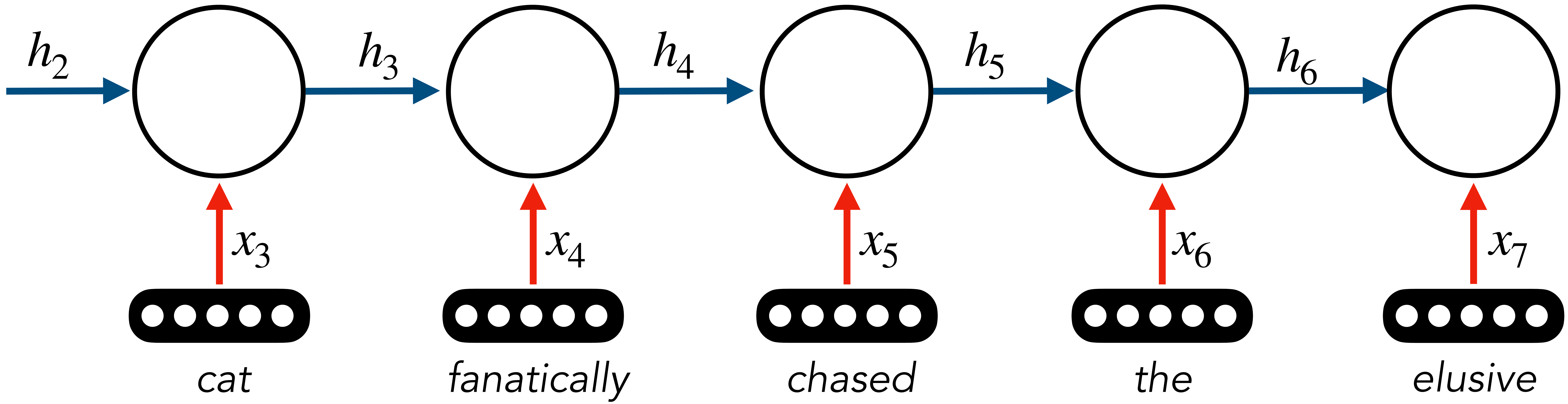
Unrolling the RNN



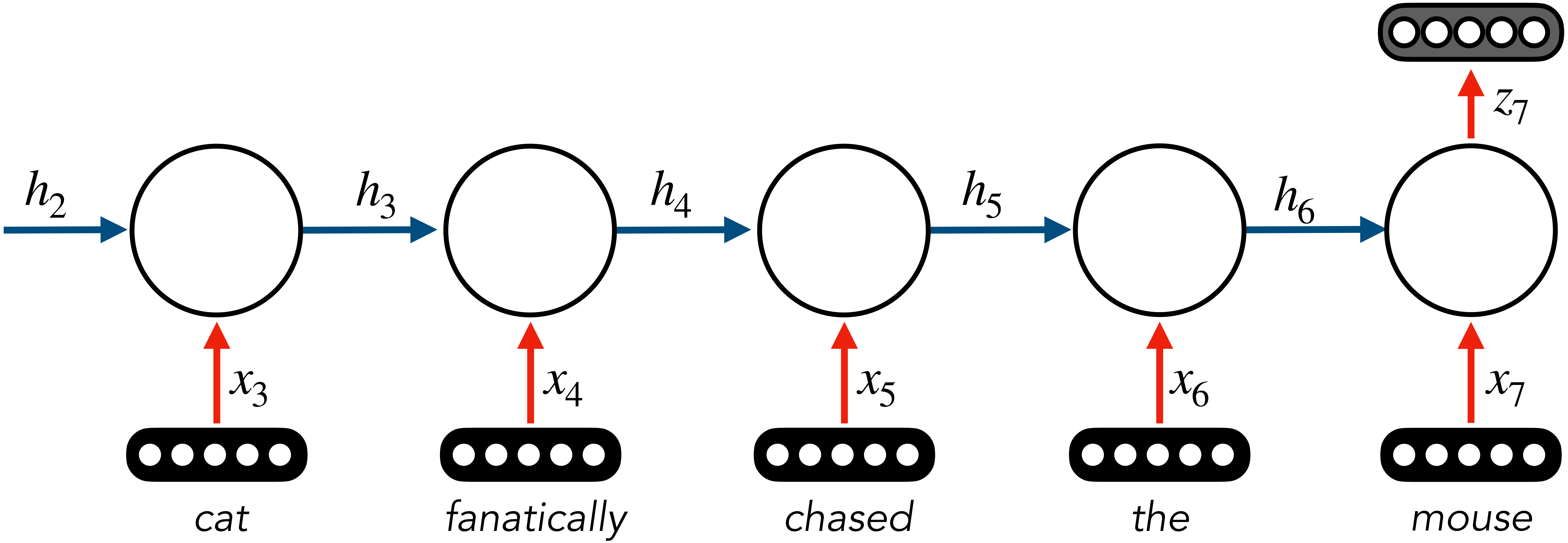
Unrolling the RNN



Unrolling the RNN



Unrolling the RNN



Classification

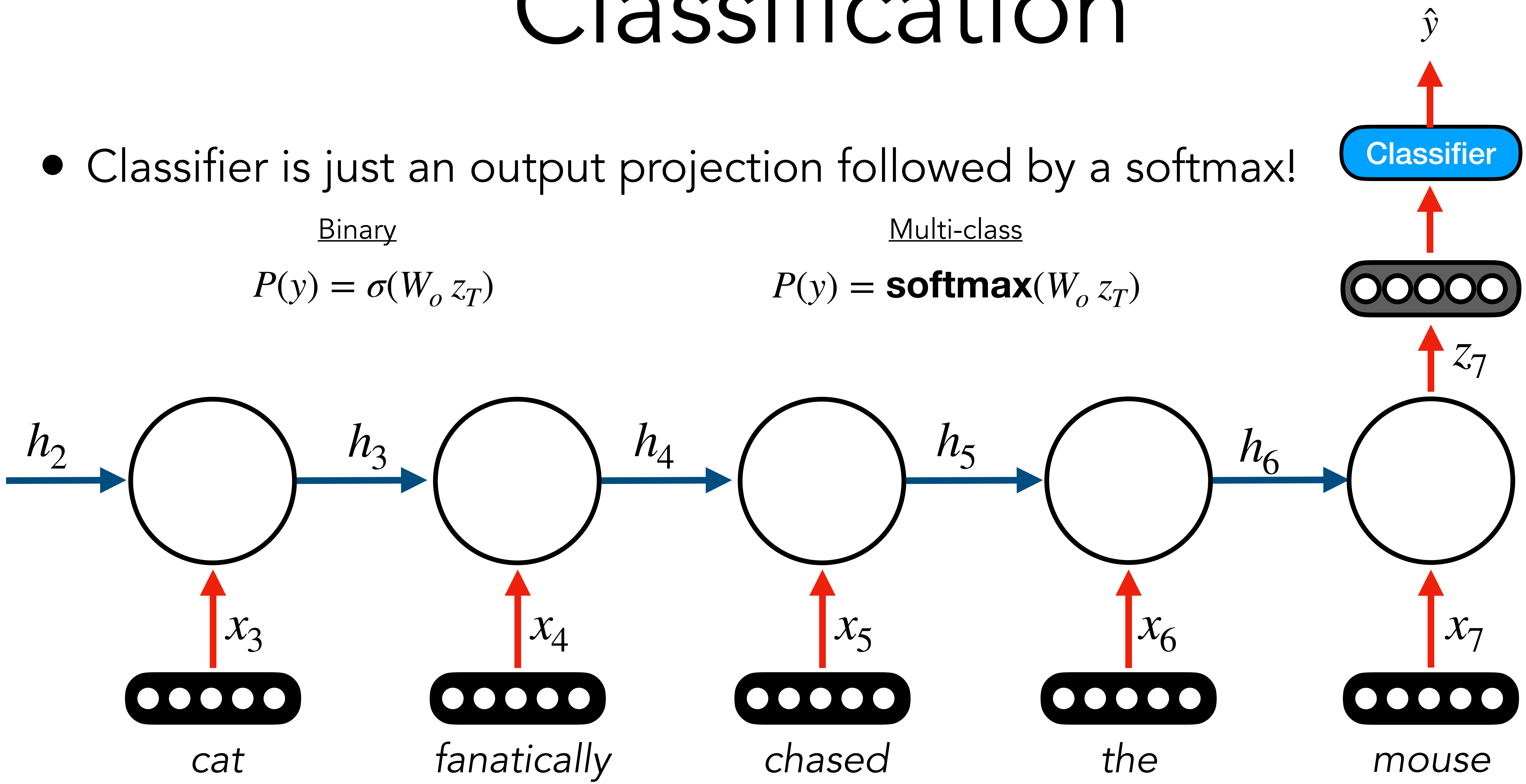
- Classifier is just an output projection followed by a softmax!

<u>Binary</u>

$$P(y) = \sigma(W_o\, z_T)$$

<u>Multi-class</u>

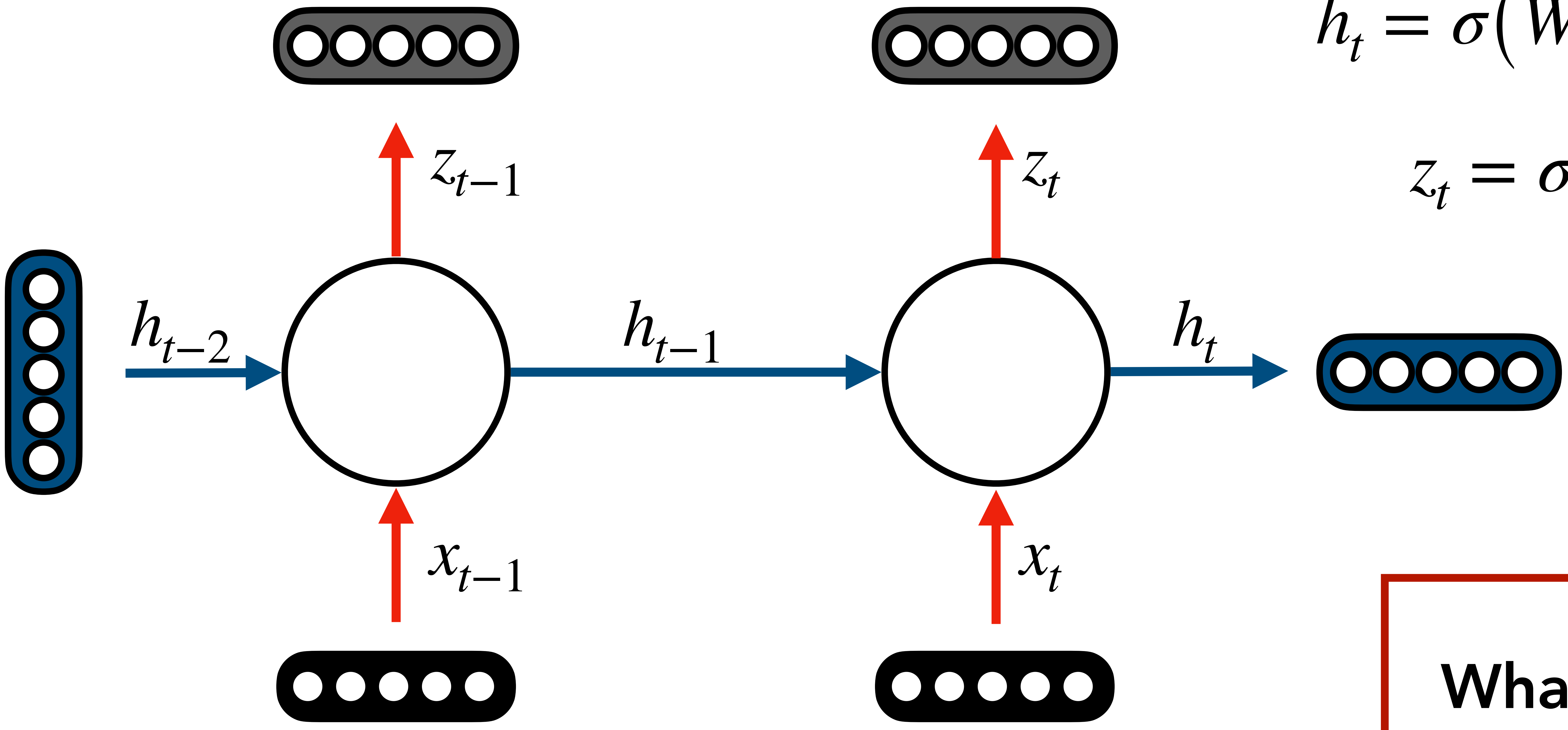
$$P(y) = \textbf{softmax}(W_o\, z_T)$$



Question

Why would you use the output of the last recurrent unit as the one to predict a label?

Classical RNN: Elman Network



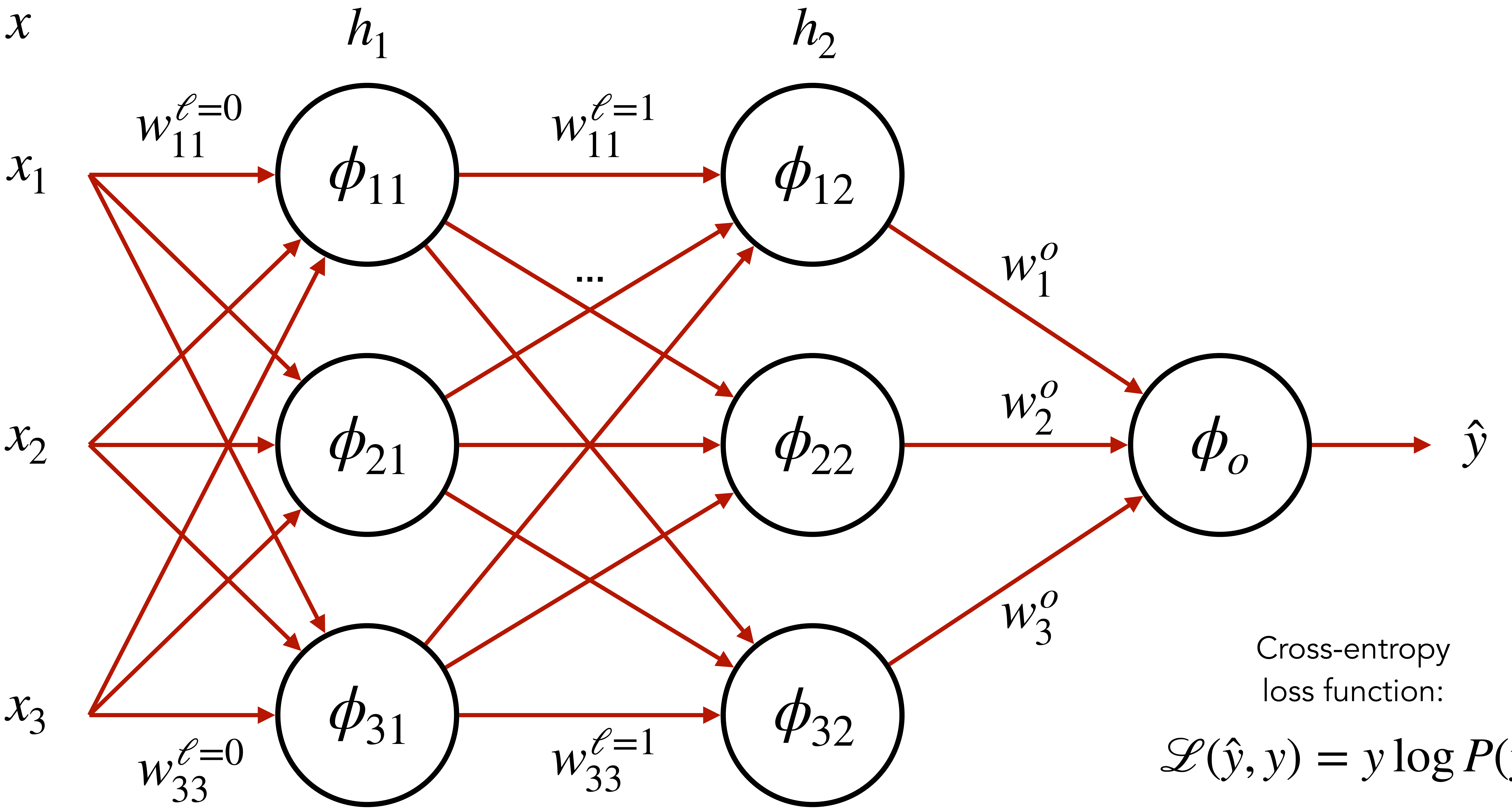
$$h_t = \sigma\left(W_{hx}x_t + W_{hh}h_{t-1} + b_h\right)$$

$$z_t = \sigma\left(W_{zh}h_t + b_z\right)$$

What should h_0 be?

(Elman, 1990)

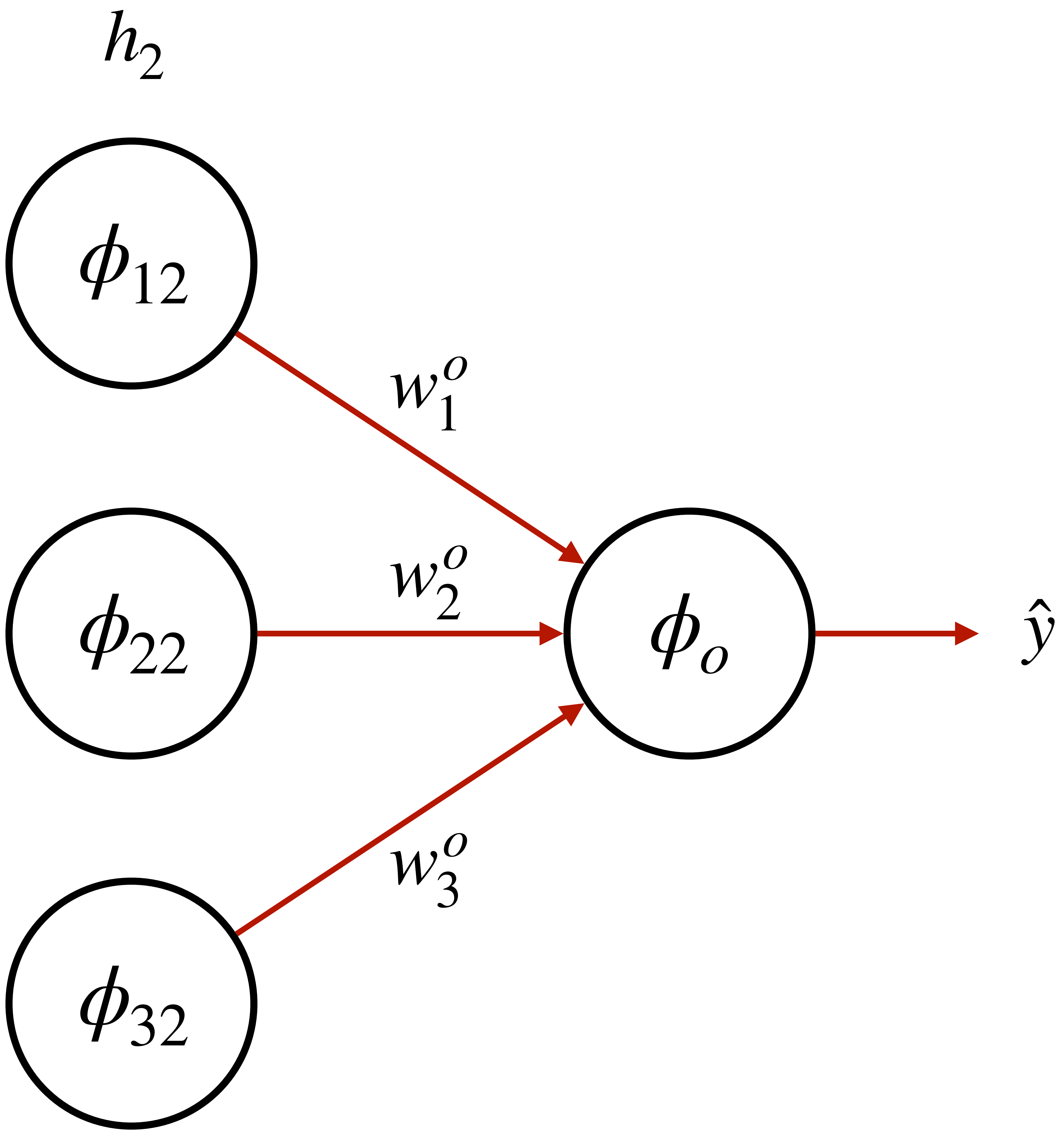
Backpropagation Review: FFNs



Cross-entropy loss function:

$$\mathscr{L}(\hat{y}, y) = y \log P(\hat{y})$$

Backpropagation Review: FFNs



$$\mathcal{L}(\hat{y}, y) = y \log P(\hat{y})$$

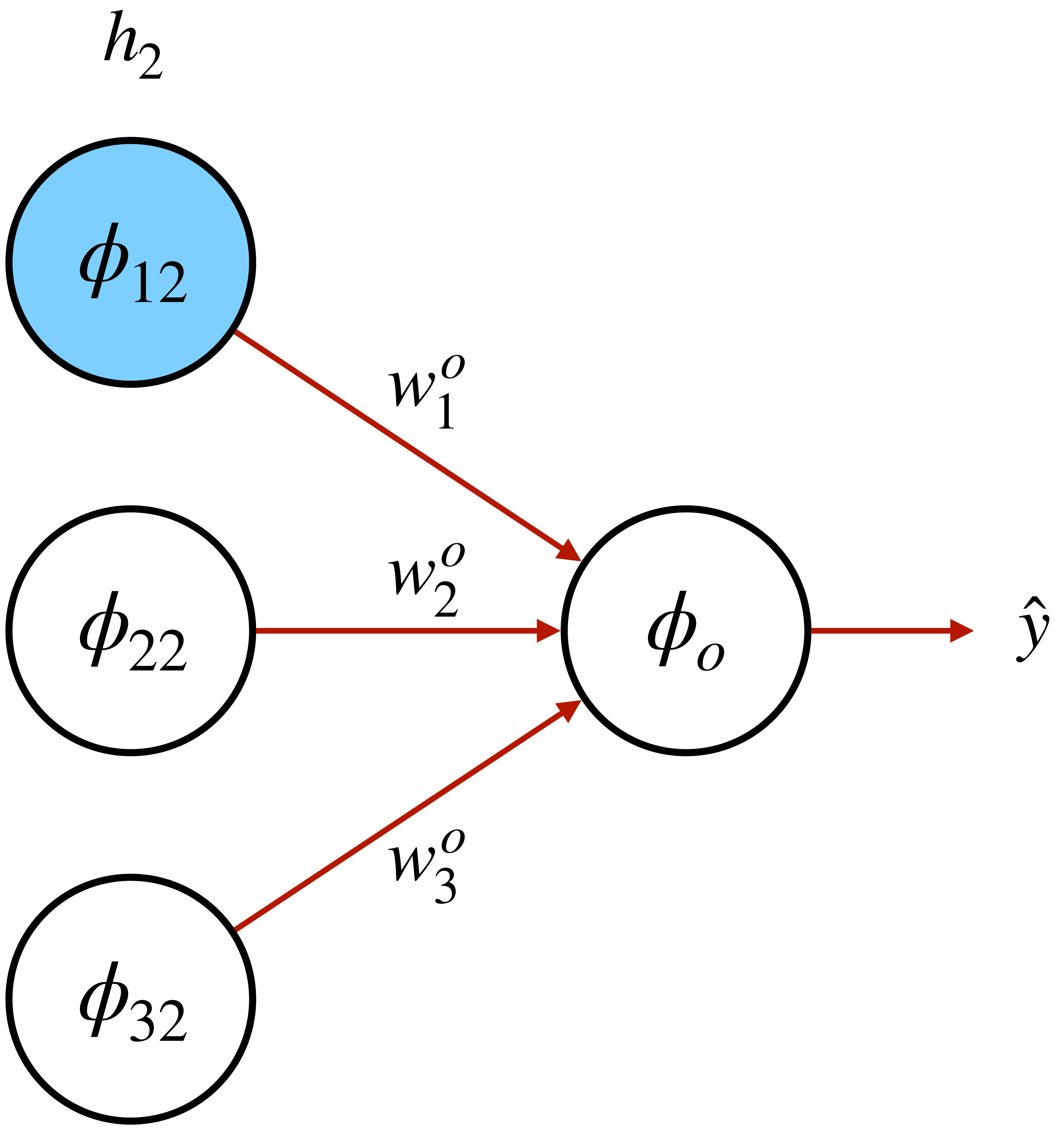
$$\hat{y} = \phi_o(u)$$

$$u = w_1^o \times \phi_{12}(\,.\,) + w_2^o \times \phi_{22}(\,.\,) + w_3^o \times \phi_{32}(\,.\,)$$

$$\frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \phi_{12}(\,.\,)} = \frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial u} \frac{\partial u}{\partial \phi_{12}(\,.\,)}$$

$$= \frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \phi_o(u)}{\partial u} w_1^0$$

Backpropagation Review: FFNs



$$\mathcal{L}(\hat{y}, y) = y \log P(\hat{y})$$

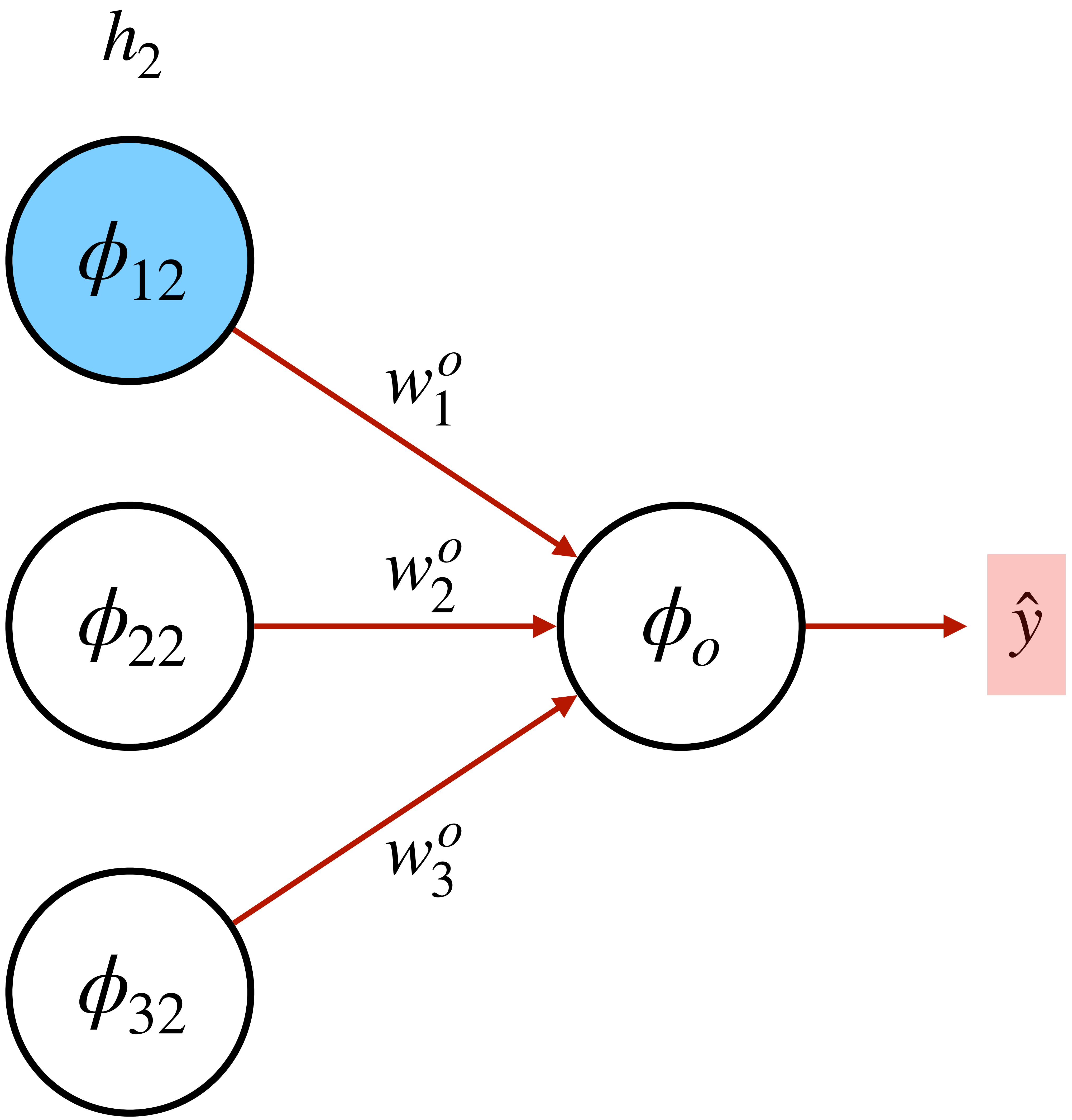
$$\hat{y} = \phi_o(u)$$

$$u = w_1^o \times \phi_{12}(\,.\,) + w_2^o \times \phi_{22}(\,.\,) + w_3^o \times \phi_{32}(\,.\,)$$

$$\frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \phi_{12}(\,.\,)} = \frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial u} \frac{\partial u}{\partial \phi_{12}(\,.\,)}$$

$$= \frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \phi_o(u)}{\partial u} w_1^0$$

Backpropagation Review: FFNs



$$\mathcal{L}(\hat{y}, y) = y \log P(\hat{y})$$

$$\hat{y} = \phi_o(u)$$

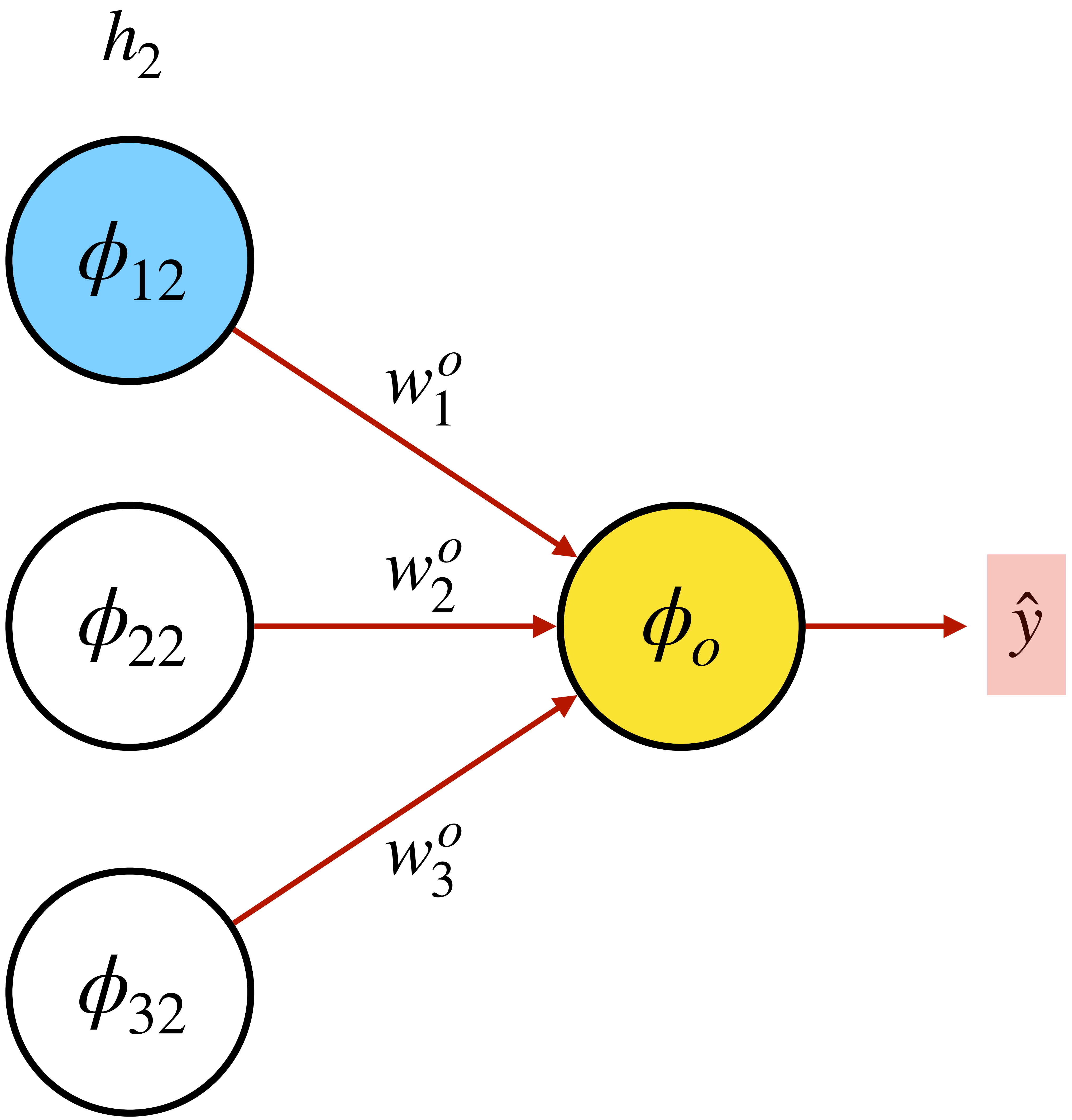
$$u = w_1^o \times \phi_{12}(\,.\,) + w_2^o \times \phi_{22}(\,.\,) + w_3^o \times \phi_{32}(\,.\,)$$

$$\frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \phi_{12}(\,.\,)} = \frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial u} \frac{\partial u}{\partial \phi_{12}(\,.\,)}$$

$$= \frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \phi_o(u)}{\partial u} w_1^0$$

Depends on label y

Backpropagation Review: FFNs



$$\mathcal{L}(\hat{y}, y) = y \log P(\hat{y}) + (1 - y) \log P(1 - \hat{y})$$

$$\hat{y} = \phi_o(u)$$

$$u = w_1^o \times \phi_{12}(\,.\,) + w_2^o \times \phi_{22}(\,.\,) + w_3^o \times \phi_{32}(\,.\,)$$

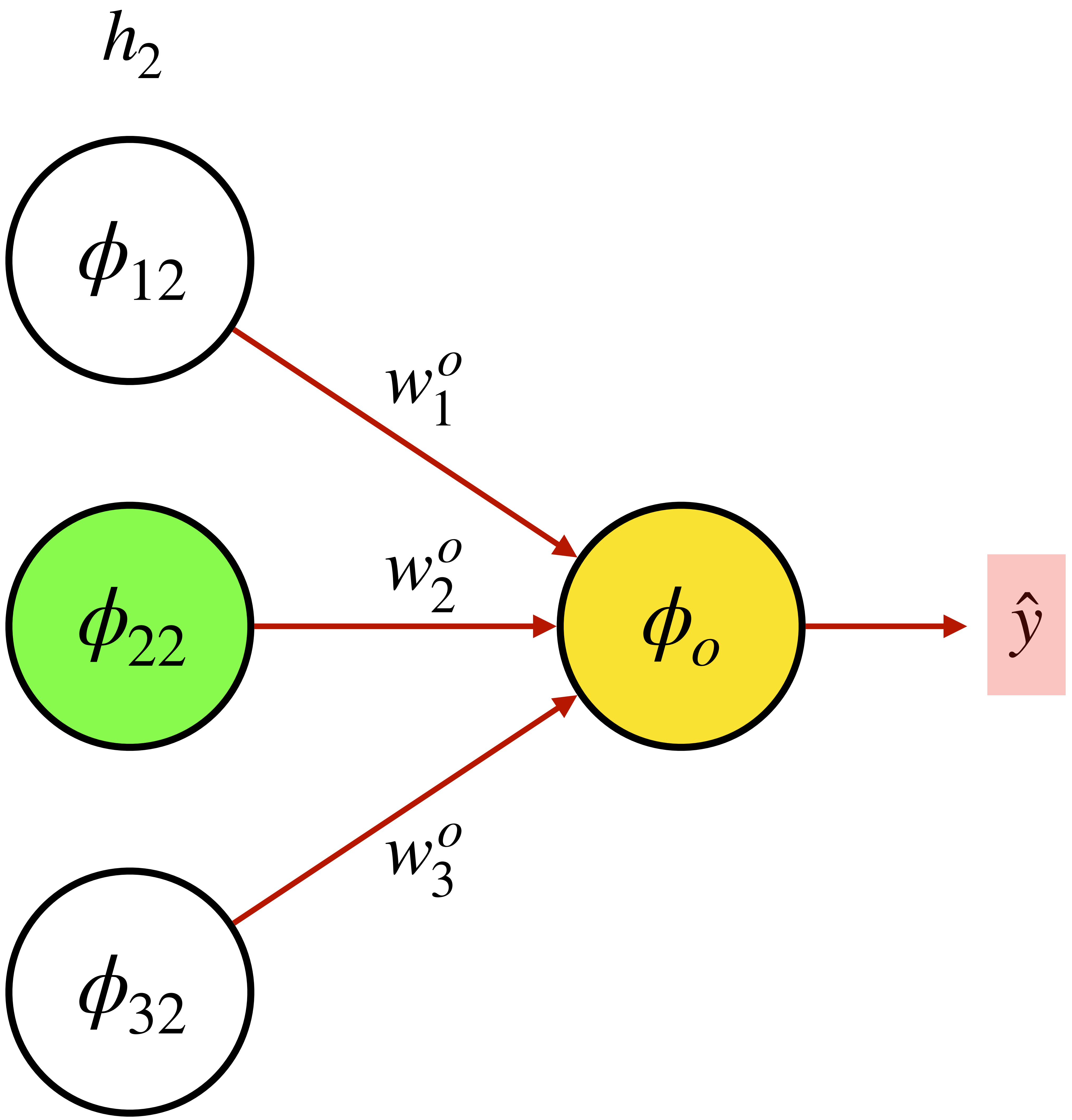
$$\frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \phi_{12}(\,.\,)} = \frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial u} \frac{\partial u}{\partial \phi_{12}(\,.\,)}$$

$$= \frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \phi_o(u)}{\partial u} w_1^0$$

Depends on label y

Depends on ϕ_o

Backpropagation Review: FFNs



$$\mathscr{L}(\hat{y}, y) = y \log P(\hat{y}) + (1 - y) \log P(1 - \hat{y})$$

$$\hat{y} = \phi_o(u)$$

$$u = w_1^o \times \phi_{12}(\,.\,) + w_2^o \times \phi_{22}(\,.\,) + w_3^o \times \phi_{32}(\,.\,)$$

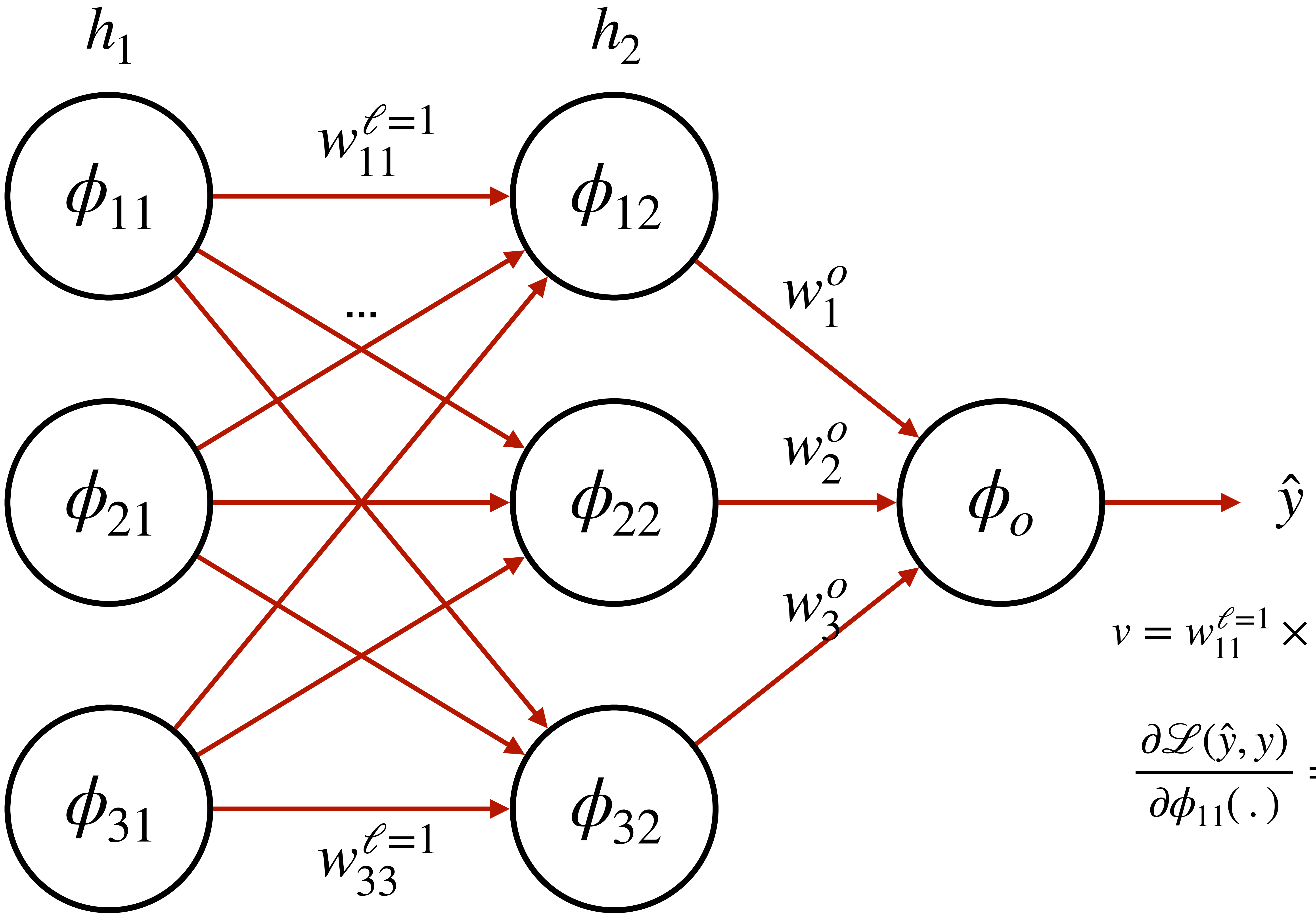
$$\frac{\partial \mathscr{L}(\hat{y}, y)}{\partial \phi_{22}(\,.\,)} = \frac{\partial \mathscr{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial u} \frac{\partial u}{\partial \phi_{22}(\,.\,)}$$

$$= \frac{\partial \mathscr{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \phi_o(u)}{\partial u} w_2^0$$

Depends on label y

Depends on ϕ_o

Backpropagation Review: FFNs



$$\frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \phi_{12}(\,.\,)} = \frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial u} \frac{\partial u}{\partial \phi_{12}(\,.\,)}$$

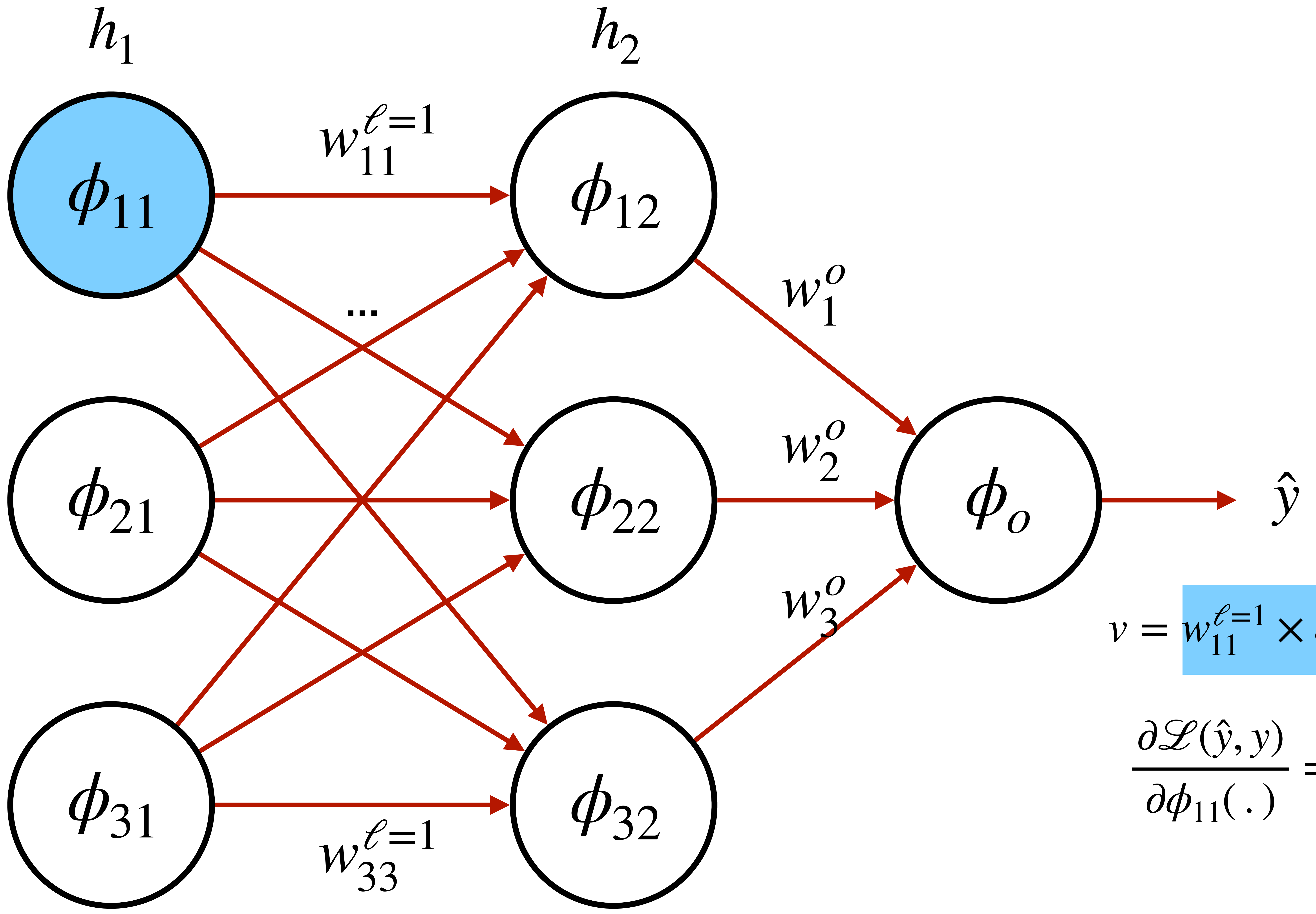
$$= \frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \phi_o(u)}{\partial u} w_1^0$$

$$v = w_{11}^{\ell=1} \times \phi_{11}(\,.\,) + w_{21}^{\ell=1} \times \phi_{21}(\,.\,) + w_{31}^{\ell=1} \times \phi_{31}(\,.\,)$$

$$\frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \phi_{11}(\,.\,)} = \frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial u} \frac{\partial u}{\partial \phi_{12}(v)} \frac{\partial \phi_{12}(v)}{\partial v} \frac{\partial v}{\partial \phi_{11}(\,.\,)}$$

$$= \frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \phi_o(u)}{\partial u} w_1^0 \frac{\partial \phi_{12}(v)}{\partial v} w_{11}^{\ell=1}$$

Backpropagation Review: FFNs



$$\frac{\partial \mathscr{L}(\hat{y}, y)}{\partial \phi_{12}(\,.\,)} = \frac{\partial \mathscr{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial u} \frac{\partial u}{\partial \phi_{12}(\,.\,)}$$

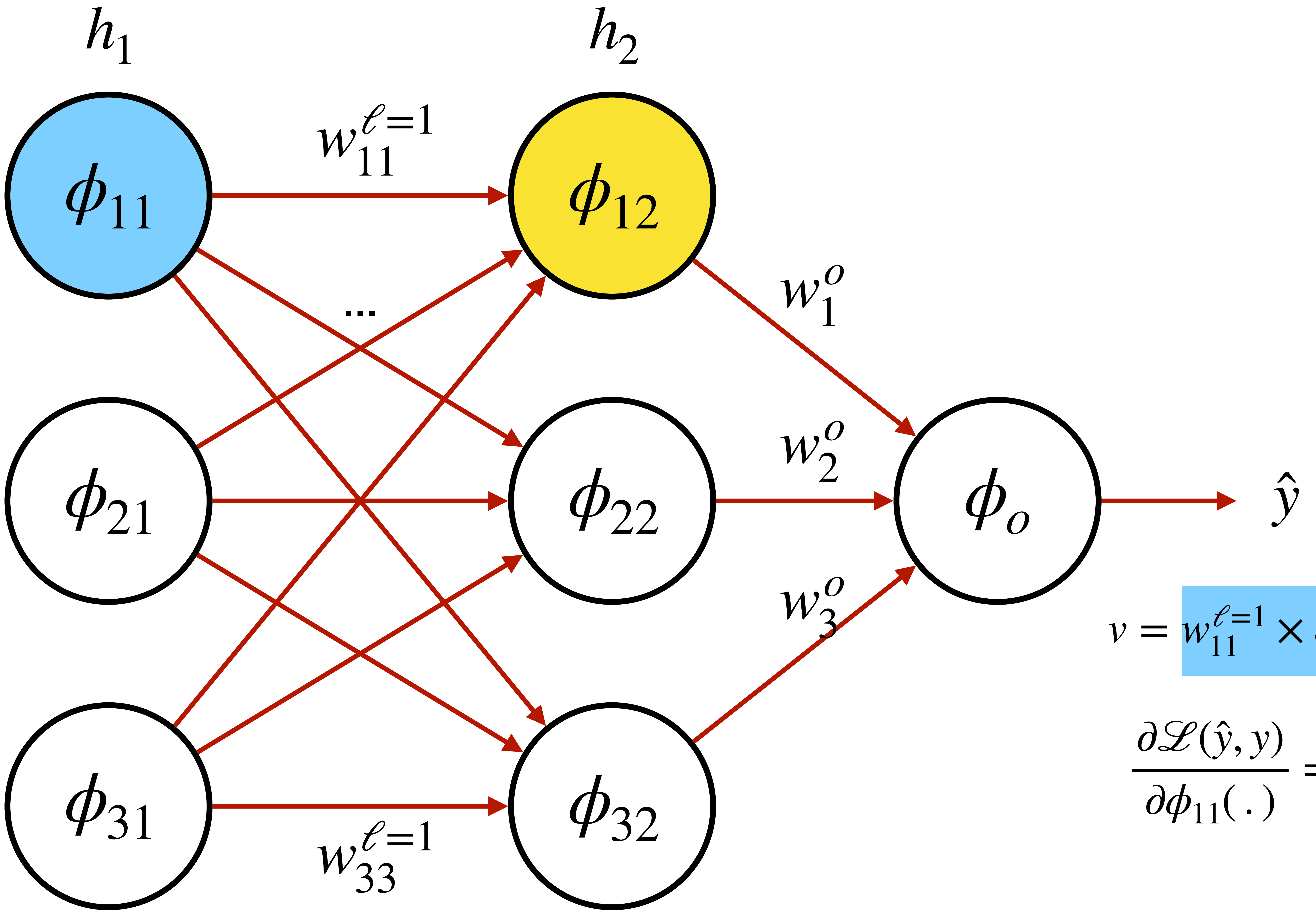
$$= \frac{\partial \mathscr{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \phi_o(u)}{\partial u} w_1^0$$

$$v = w_{11}^{\ell=1} \times \phi_{11}(\,.\,) + w_{21}^{\ell=1} \times \phi_{21}(\,.\,) + w_{31}^{\ell=1} \times \phi_{31}(\,.\,)$$

$$\frac{\partial \mathscr{L}(\hat{y}, y)}{\partial \phi_{11}(\,.\,)} = \frac{\partial \mathscr{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial u} \frac{\partial u}{\partial \phi_{12}(v)} \frac{\partial \phi_{12}(v)}{\partial v} \frac{\partial v}{\partial \phi_{11}(\,.\,)}$$

$$= \frac{\partial \mathscr{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \phi_o(u)}{\partial u} w_1^0 \frac{\partial \phi_{12}(v)}{\partial v} w_{11}^{\ell=1}$$

Backpropagation Review: FFNs



$$\frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \phi_{12}(\,.\,)} = \frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial u} \frac{\partial u}{\partial \phi_{12}(\,.\,)}$$

$$= \frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \phi_o(u)}{\partial u} w_1^0$$

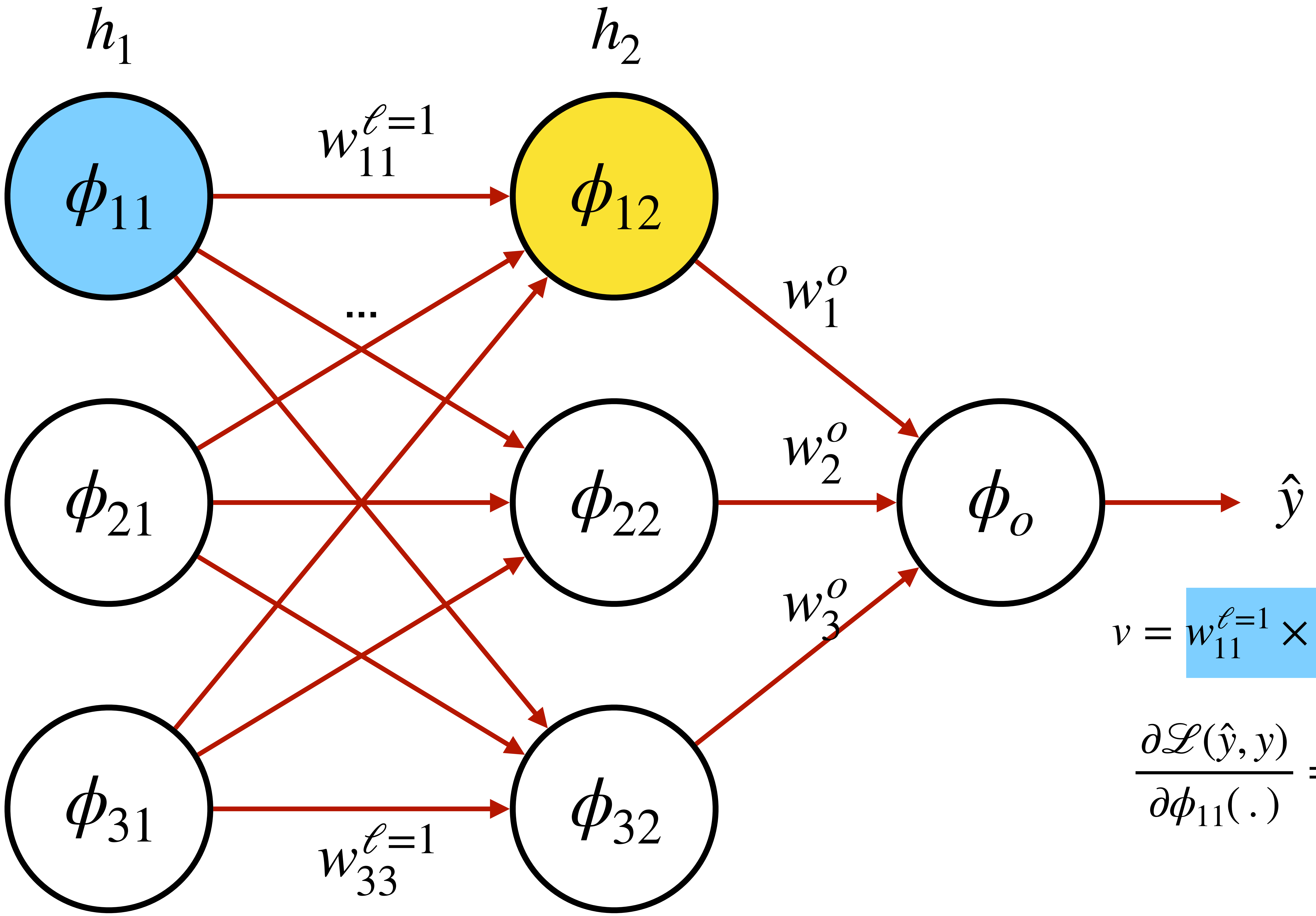
Depends on ϕ_{12}

$$v = \boxed{w_{11}^{\ell=1} \times \phi_{11}(\,.\,)} + w_{21}^{\ell=1} \times \phi_{21}(\,.\,) + w_{31}^{\ell=1} \times \phi_{31}(\,.\,)$$

$$\frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \phi_{11}(\,.\,)} = \frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial u} \frac{\partial u}{\partial \phi_{12}(v)} \boxed{\frac{\partial \phi_{12}(v)}{\partial v}} \boxed{\frac{\partial v}{\partial \phi_{11}(\,.\,)}}$$

$$= \frac{\partial \mathcal{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \phi_o(u)}{\partial u} w_1^0 \boxed{\frac{\partial \phi_{12}(v)}{\partial v}} \boxed{w_{11}^{\ell=1}}$$

Backpropagation Review: FFNs



$$\frac{\partial \mathscr{L}(\hat{y}, y)}{\partial \phi_{12}(\,.\,)} = \frac{\partial \mathscr{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial u} \frac{\partial u}{\partial \phi_{12}(\,.\,)}$$

$$= \frac{\partial \mathscr{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \phi_o(u)}{\partial u} w_1^0$$

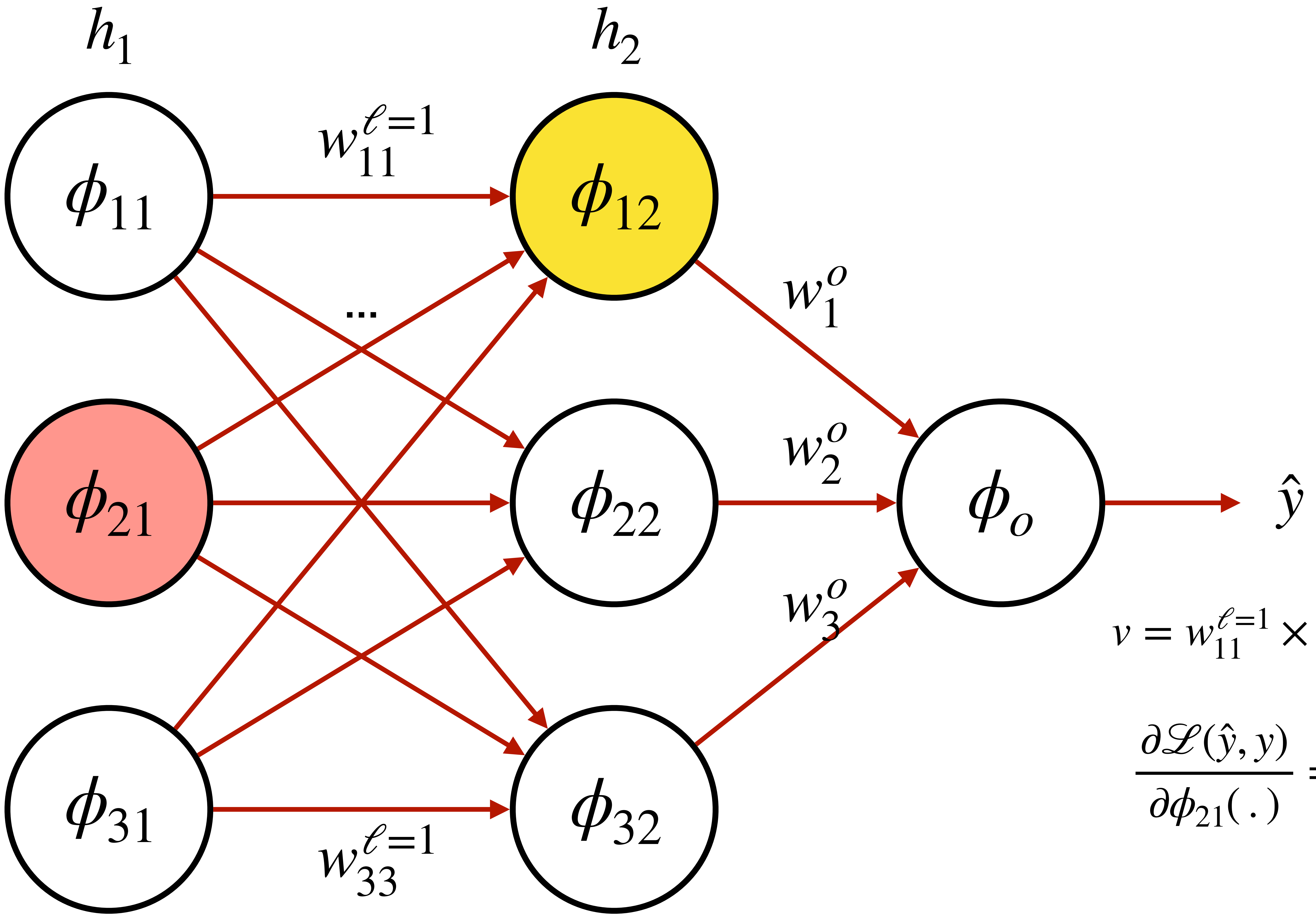
Depends on ϕ_{12}

$$v = w_{11}^{\ell=1} \times \phi_{11}(\,.\,) + w_{21}^{\ell=1} \times \phi_{21}(\,.\,) + w_{31}^{\ell=1} \times \phi_{31}(\,.\,)$$

$$\frac{\partial \mathscr{L}(\hat{y}, y)}{\partial \phi_{11}(\,.\,)} = \frac{\partial \mathscr{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial u} \frac{\partial u}{\partial \phi_{12}(v)} \frac{\partial \phi_{12}(v)}{\partial v} \frac{\partial v}{\partial \phi_{11}(\,.\,)}$$

$$= \frac{\partial \mathscr{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \phi_o(u)}{\partial u} w_1^0 \frac{\partial \phi_{12}(v)}{\partial v} w_{11}^{\ell=1}$$

Backpropagation Review: FFNs



$$\frac{\partial \mathscr{L}(\hat{y}, y)}{\partial \phi_{12}(\,.\,)} = \frac{\partial \mathscr{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial u} \frac{\partial u}{\partial \phi_{12}(\,.\,)}$$

$$= \frac{\partial \mathscr{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \phi_o(u)}{\partial u} w_1^0$$

Depends on ϕ_{12}

$$v = w_{11}^{\ell=1} \times \phi_{11}(\,.\,) + w_{21}^{\ell=1} \times \phi_{21}(\,.\,) + w_{31}^{\ell=1} \times \phi_{31}(\,.\,)$$

$$\frac{\partial \mathscr{L}(\hat{y}, y)}{\partial \phi_{21}(\,.\,)} = \frac{\partial \mathscr{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial u} \frac{\partial u}{\partial \phi_{12}(v)} \frac{\partial \phi_{12}(v)}{\partial v} \frac{\partial v}{\partial \phi_{21}(\,.\,)}$$

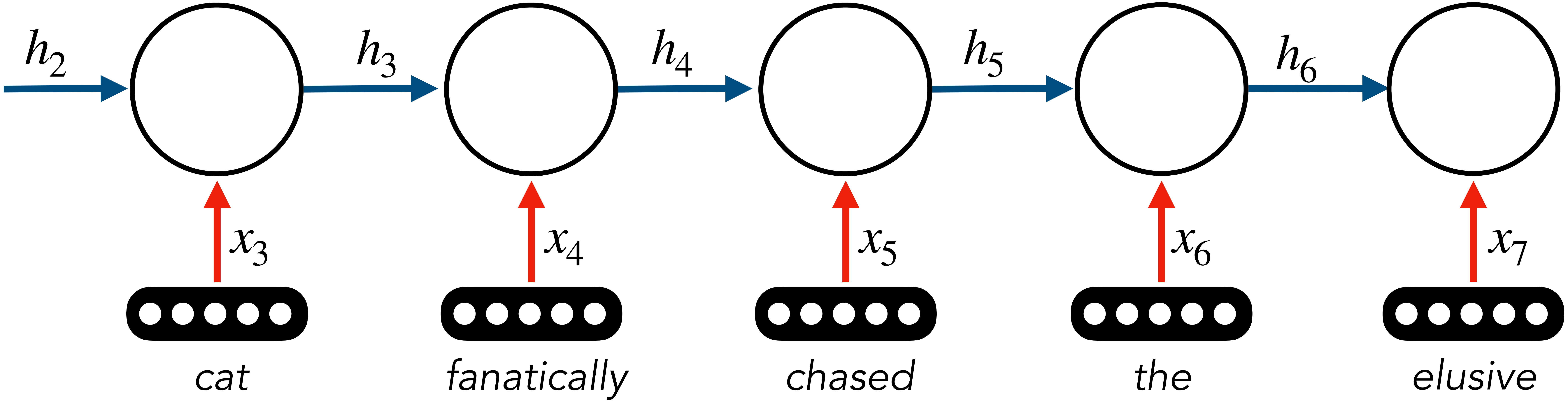
$$= \frac{\partial \mathscr{L}(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \phi_o(u)}{\partial u} w_1^0 \frac{\partial \phi_{12}(v)}{\partial v} w_{21}^{\ell=1}$$

Question

How would we extend backpropagation to a recurrent neural network?

Recall

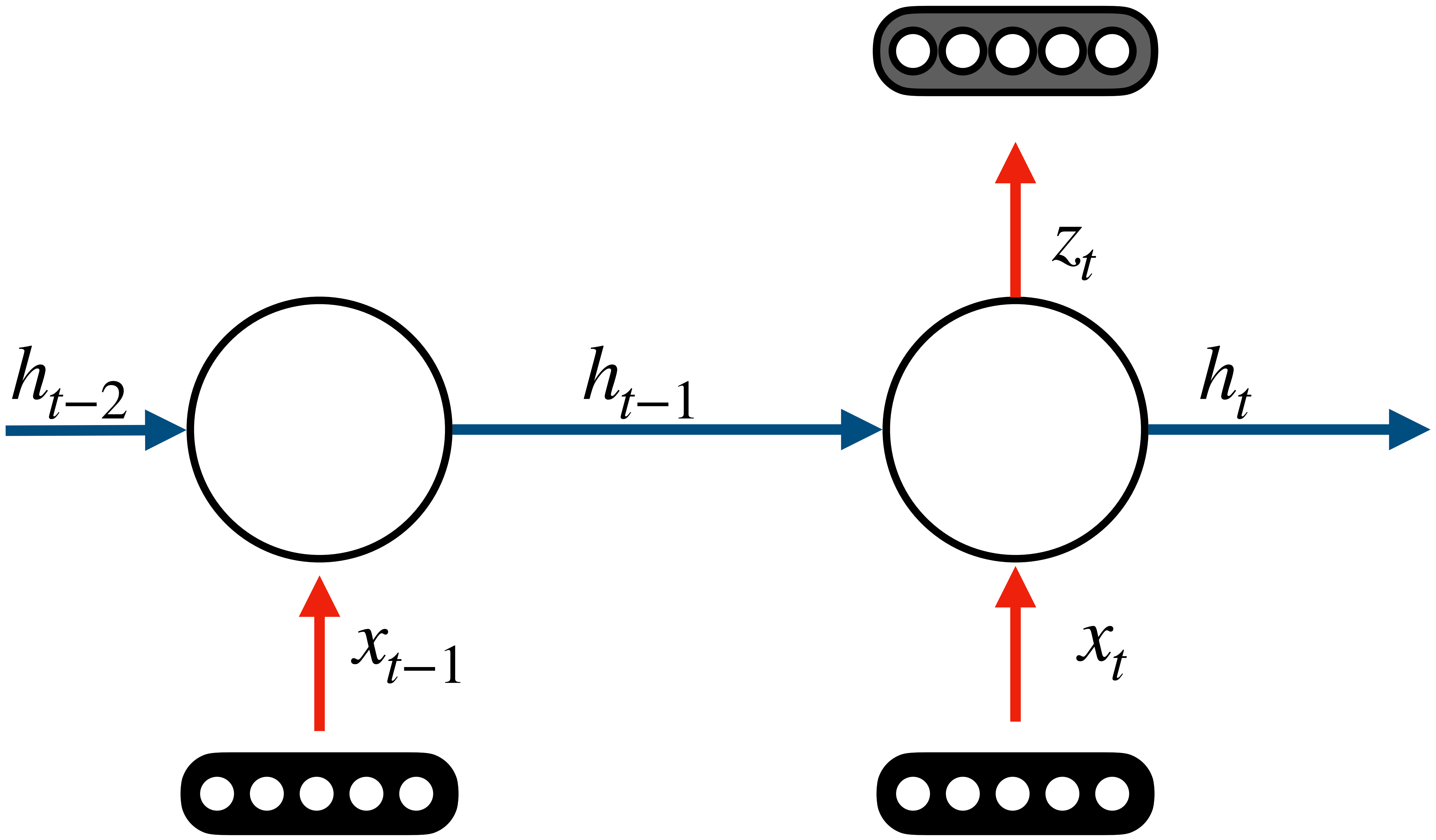
- RNN can be unrolled to a feedforward neural network
- Depth of feedforward neural network depends on length of the sequence



Backpropagation through Time

$$z_t = \sigma\left(W_{zh}h_t + b_z\right)$$

$$h_t = \sigma\left(W_{hx}x_t + W_{hh}h_{t-1} + b_h\right)$$



Backpropagation through Time

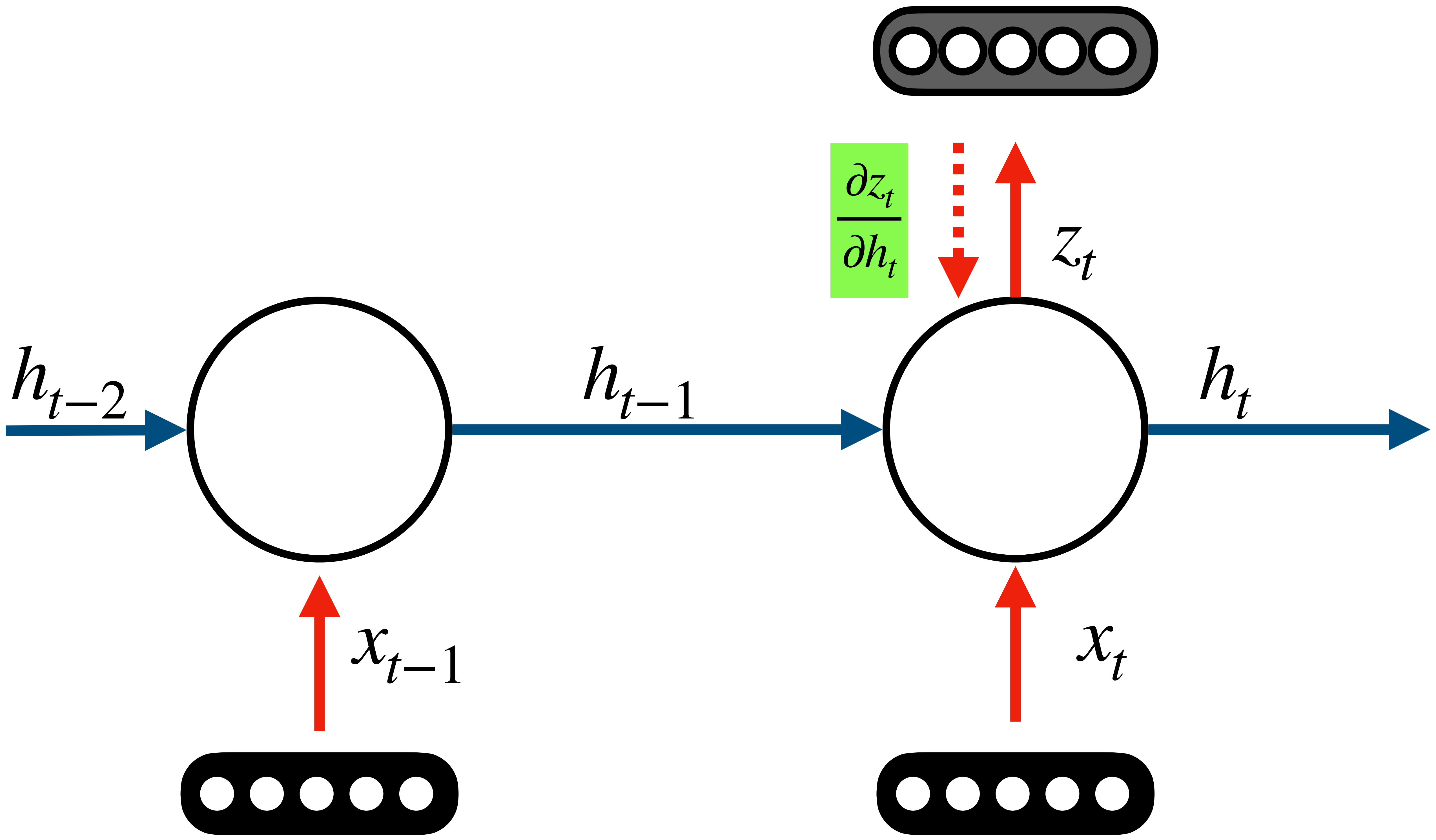
$$z_t = \sigma\left(W_{zh}h_t + b_z\right)$$

$$h_t = \sigma\left(W_{hx}x_t + W_{hh}h_{t-1} + b_h\right)$$

$$v = W_{zh}h_t + b_z \qquad z_t = \sigma(v)$$

$$u = W_{hx}x_t + W_{hh}h_{t-1} + b_h \qquad h_t = \sigma(u)$$

$$\frac{\partial z_t}{\partial h_t} = \frac{\partial \sigma(v)}{\partial v}\frac{\partial v}{\partial h_t} = \frac{\partial \sigma(v)}{\partial v} W_{zh}$$



Backpropagation through Time

$$z_t = \sigma\left(W_{zh}h_t + b_z\right)$$

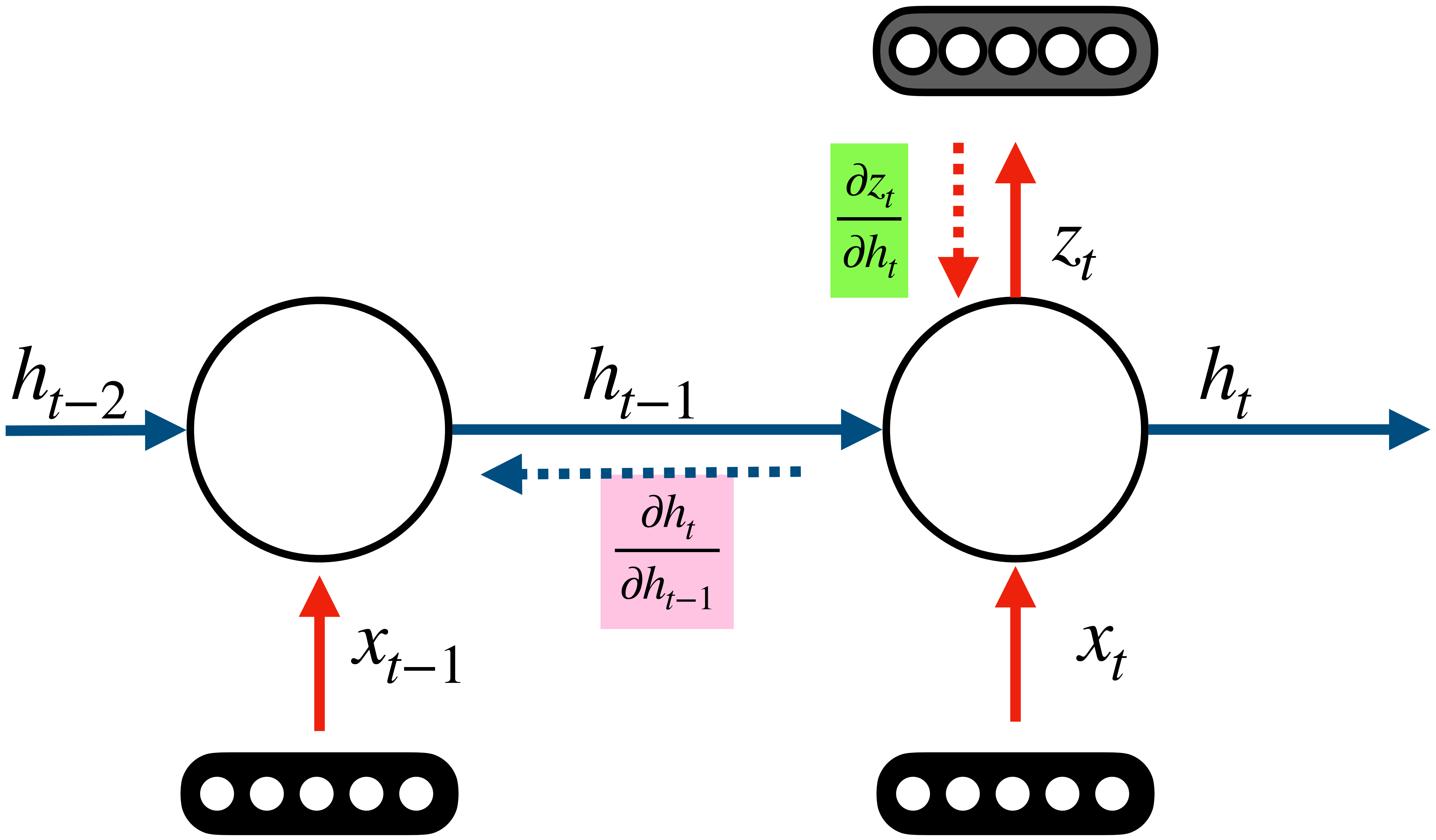
$$h_t = \sigma\left(W_{hx}x_t + W_{hh}h_{t-1} + b_h\right)$$

$$v = W_{zh}h_t + b_z \qquad z_t = \sigma(v)$$

$$u = W_{hx}x_t + W_{hh}h_{t-1} + b_h \qquad h_t = \sigma(u)$$

$$\frac{\partial z_t}{\partial h_t} = \frac{\partial \sigma(v)}{\partial v}\frac{\partial v}{\partial h_t} = \frac{\partial \sigma(v)}{\partial v}W_{zh}$$

$$\frac{\partial h_t}{\partial h_{t-1}} = \frac{\partial \sigma(u)}{\partial u}\frac{\partial u}{\partial h_{t-1}} = \frac{\partial \sigma(u)}{\partial u}W_{hh}$$



Backpropagation through Time

$$z_t = \sigma\left(W_{zh} h_t + b_z\right)$$

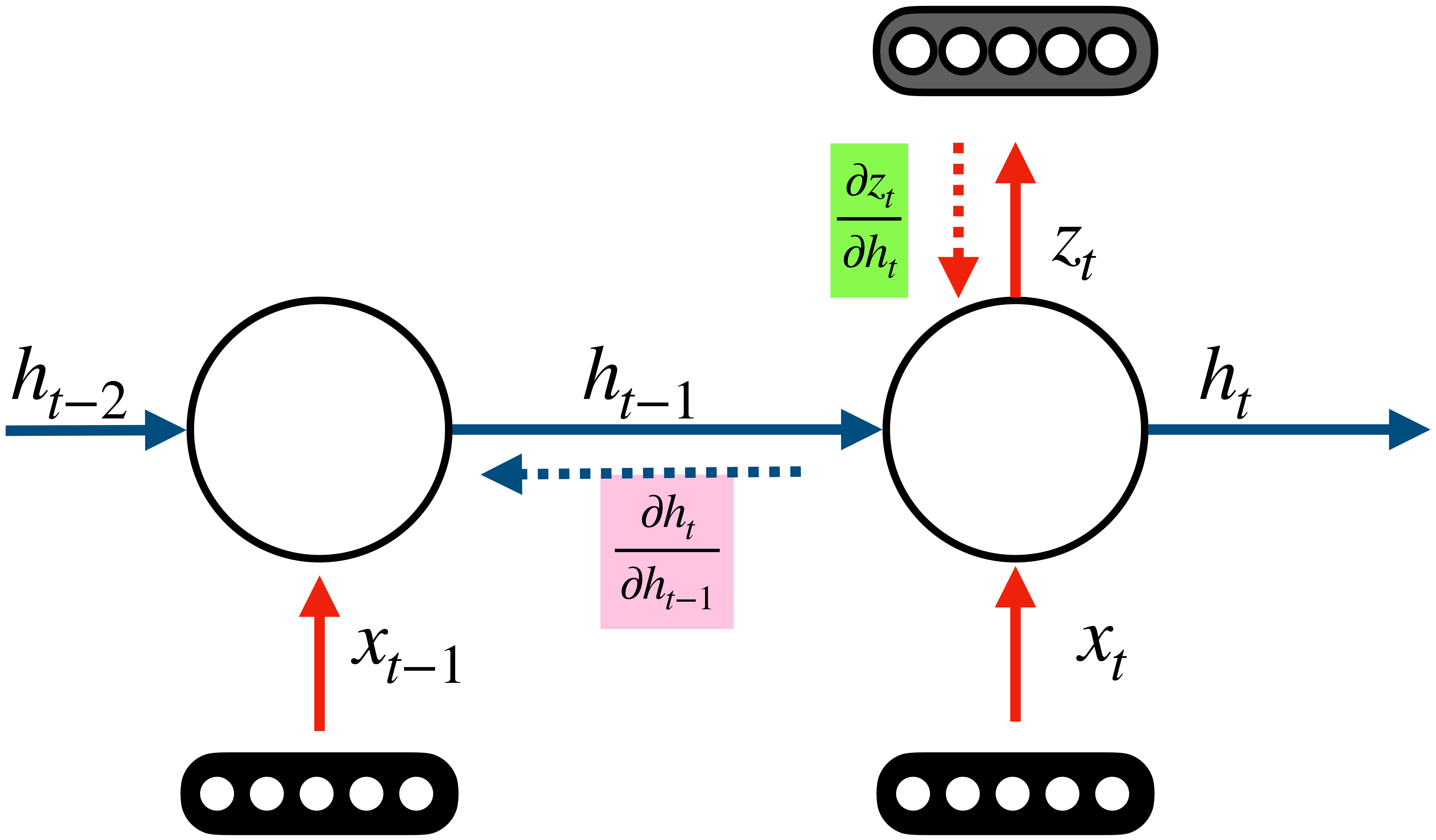
$$h_t = \sigma\left(W_{hx} x_t + W_{hh} h_{t-1} + b_h\right)$$

$$v = W_{zh} h_t + b_z \qquad z_t = \sigma(v)$$

$$u = W_{hx} x_t + W_{hh} h_{t-1} + b_h \qquad h_t = \sigma(u)$$

$$\frac{\partial z_t}{\partial h_t} = \frac{\partial \sigma(v)}{\partial v} \frac{\partial v}{\partial h_t} = \frac{\partial \sigma(v)}{\partial v} W_{zh}$$

$$\frac{\partial h_t}{\partial h_{t-1}} = \frac{\partial \sigma(u)}{\partial u} \frac{\partial u}{\partial h_{t-1}} = \frac{\partial \sigma(u)}{\partial u} W_{hh}$$



$$\frac{\partial z_t}{\partial h_{t-1}} = \frac{\partial z_t}{\partial h_t} \frac{\partial h_t}{\partial h_{t-1}}$$

Backpropagation through Time

$$z_t = \sigma\left(W_{zh}h_t + b_z\right)$$

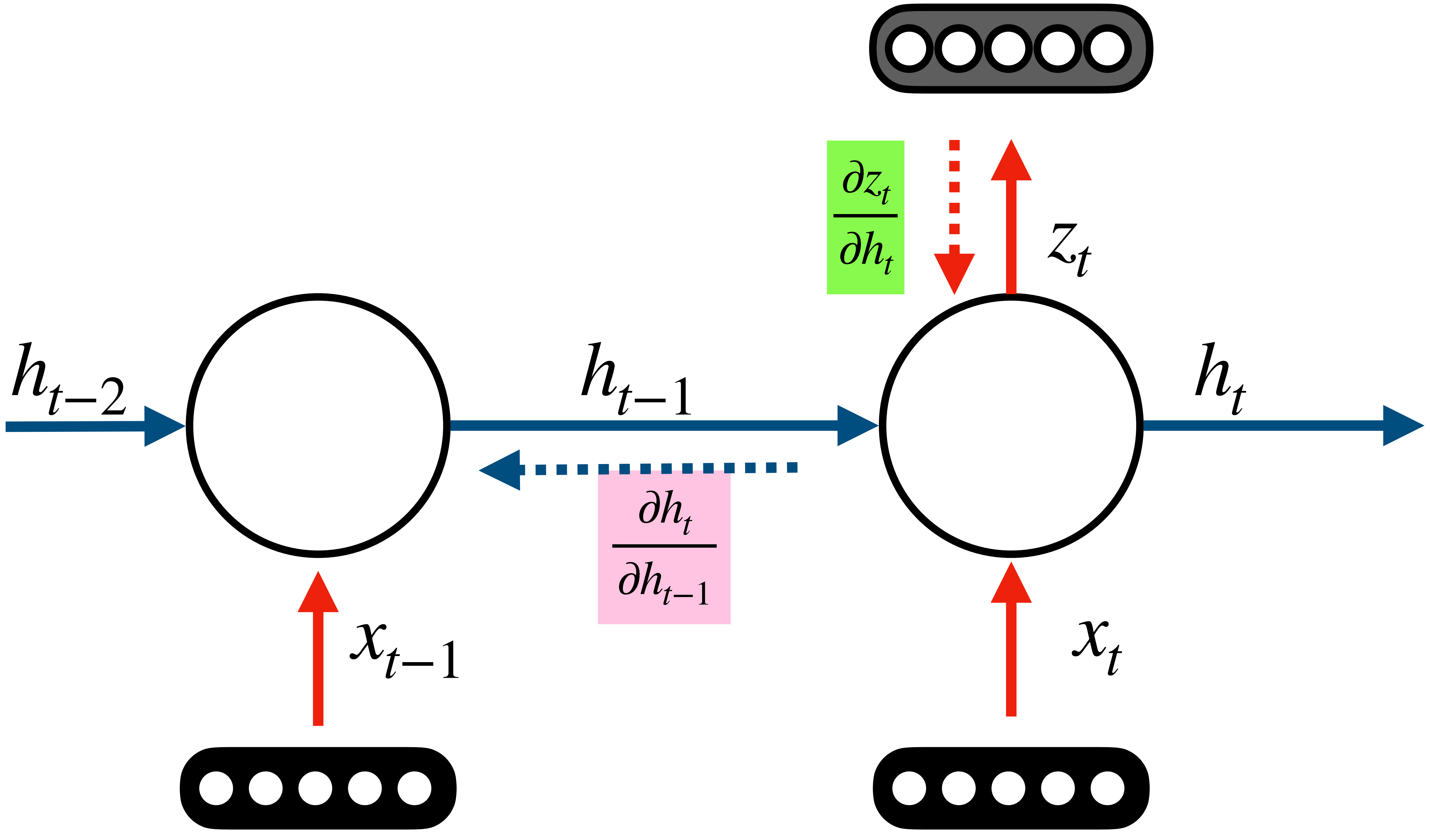
$$h_t = \sigma\left(W_{hx}x_t + W_{hh}h_{t-1} + b_h\right)$$

$$v = W_{zh}h_t + b_z \qquad z_t = \sigma(v)$$

$$u = W_{hx}x_t + W_{hh}h_{t-1} + b_h \qquad h_t = \sigma(u)$$

$$\frac{\partial z_t}{\partial h_t} = \frac{\partial \sigma(v)}{\partial v}\frac{\partial v}{\partial h_t} = \frac{\partial \sigma(v)}{\partial v}W_{zh}$$

$$\frac{\partial h_t}{\partial h_{t-1}} = \frac{\partial \sigma(u)}{\partial u}\frac{\partial u}{\partial h_{t-1}} = \frac{\partial \sigma(u)}{\partial u}W_{hh}$$



$$\frac{\partial z_t}{\partial h_{t-1}} = \frac{\partial z_t}{\partial h_t}\frac{\partial h_t}{\partial h_{t-1}} = \frac{\partial \sigma(v)}{\partial v}\frac{\partial v}{\partial h_t}\frac{\partial \sigma(u)}{\partial u}\frac{\partial u}{\partial h_{t-1}} = \frac{\partial \sigma(v)}{\partial v}W_{zh}\frac{\partial \sigma(u)}{\partial u}W_{hh}$$

Backpropagation through Time

$$z_t = \sigma\left(W_{zh}h_t + b_z\right)$$

$$h_t = \sigma\left(W_{hx}x_t + W_{hh}h_{t-1} + b_h\right)$$

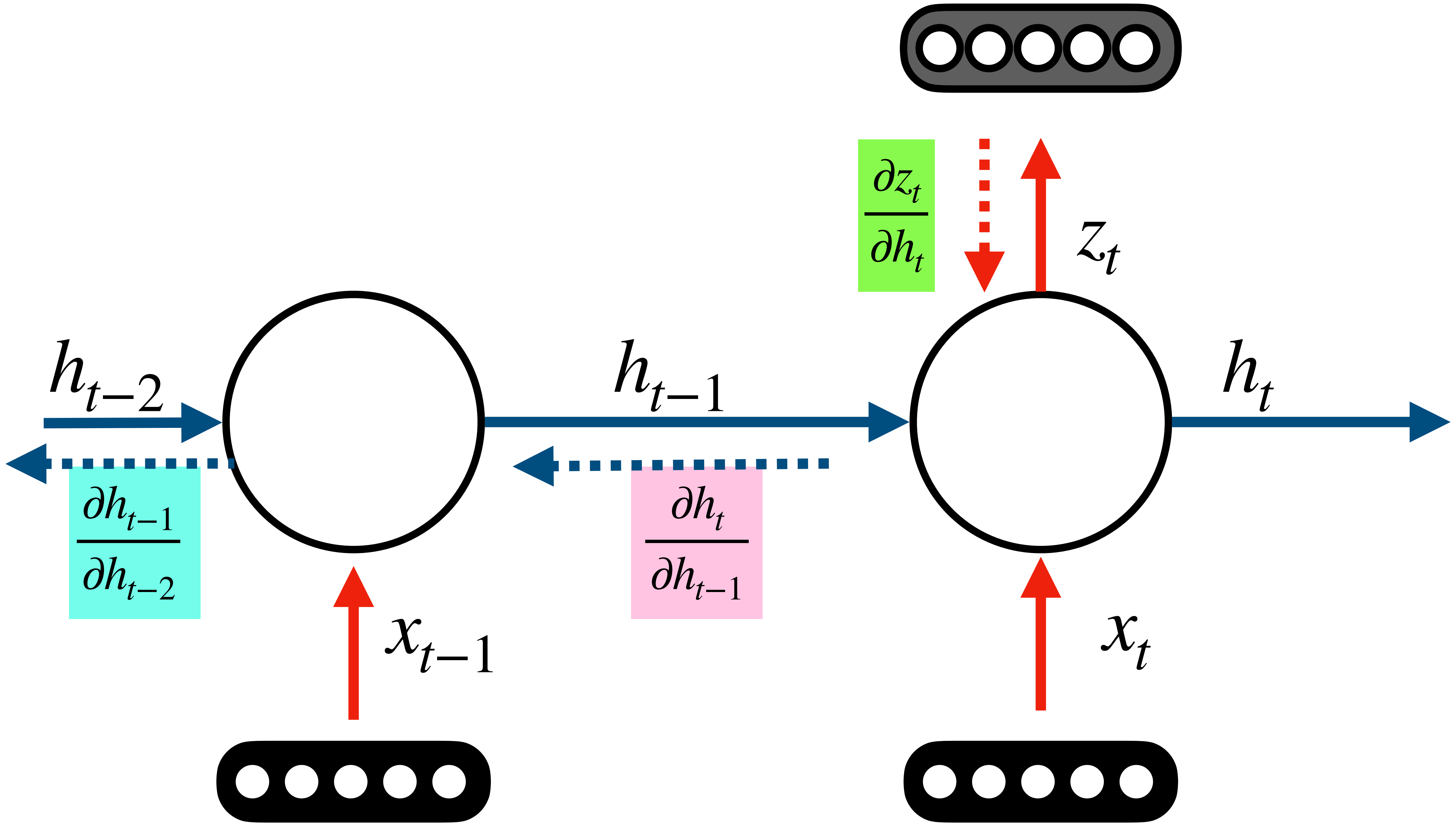
$$v_t = W_{zh}h_t + b_z \qquad z_t = \sigma(v_t)$$

$$u_t = W_{hx}x_t + W_{hh}h_{t-1} + b_h \qquad h_t = \sigma(u_t)$$

$$\frac{\partial z_t}{\partial h_t} = \frac{\partial \sigma(v_t)}{\partial v_t}\frac{\partial v_t}{\partial h_t} = \frac{\partial \sigma(v_t)}{\partial v_t}W_{zh}$$

$$\frac{\partial h_t}{\partial h_{t-1}} = \frac{\partial \sigma(u_t)}{\partial u_t}\frac{\partial u_t}{\partial h_{t-1}} = \frac{\partial \sigma(u_t)}{\partial u_t}W_{hh}$$

$$\frac{\partial h_{t-1}}{\partial h_{t-2}} = \frac{\partial \sigma(u_{t-1})}{\partial u_{t-1}}\frac{\partial u_{t-1}}{\partial h_{t-2}} = \frac{\partial \sigma(u_{t-1})}{\partial u_{t-1}}W_{hh}$$



$$\frac{\partial z_t}{\partial h_{t-1}} = \frac{\partial z_t}{\partial h_t}\frac{\partial h_t}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial h_{t-2}} = \frac{\partial \sigma(v_t)}{\partial v_t}W_{zh}\frac{\partial \sigma(u_t)}{\partial u_t}W_{hh}\frac{\partial \sigma(u_{t-1})}{\partial u_{t-1}}W_{hh}$$

Backpropagation through Time

$$z_t = \sigma\left(W_{zh} h_t + b_z\right)$$

$$h_t = \sigma\left(W_{hx} x_t + W_{hh} h_{t-1} + b_h\right)$$

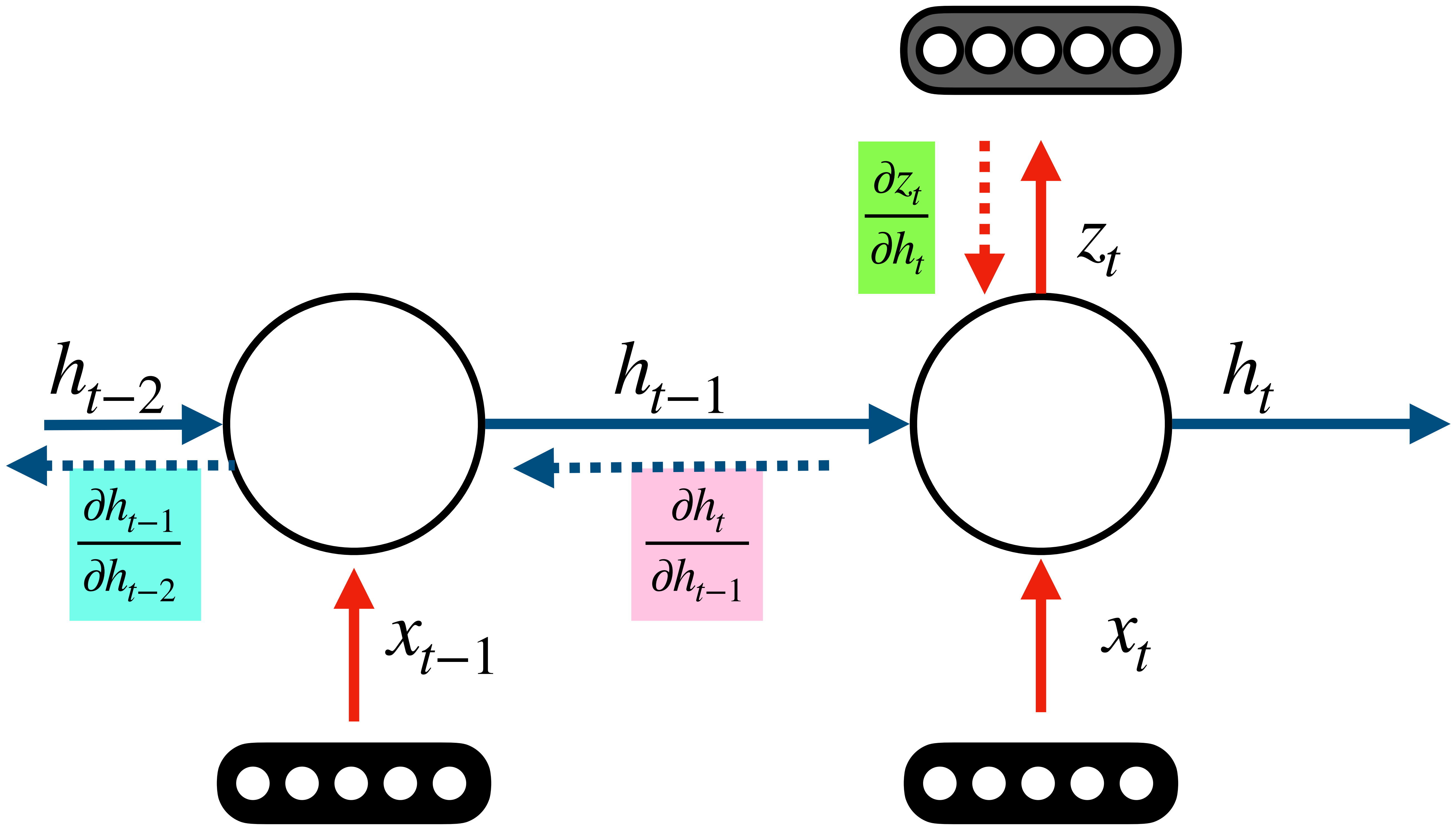
$$v_t = W_{zh} h_t + b_z \qquad z_t = \sigma(v_t)$$

$$u_t = W_{hx} x_t + W_{hh} h_{t-1} + b_h \qquad h_t = \sigma(u_t)$$

$$\frac{\partial z_t}{\partial h_t} = \frac{\partial \sigma(v_t)}{\partial v_t} \frac{\partial v_t}{\partial h_t} = \frac{\partial \sigma(v_t)}{\partial v_t} W_{zh}$$

$$\frac{\partial h_t}{\partial h_{t-1}} = \frac{\partial \sigma(u_t)}{\partial u_t} \frac{\partial u_t}{\partial h_{t-1}} = \frac{\partial \sigma(u_t)}{\partial u_t} W_{hh}$$

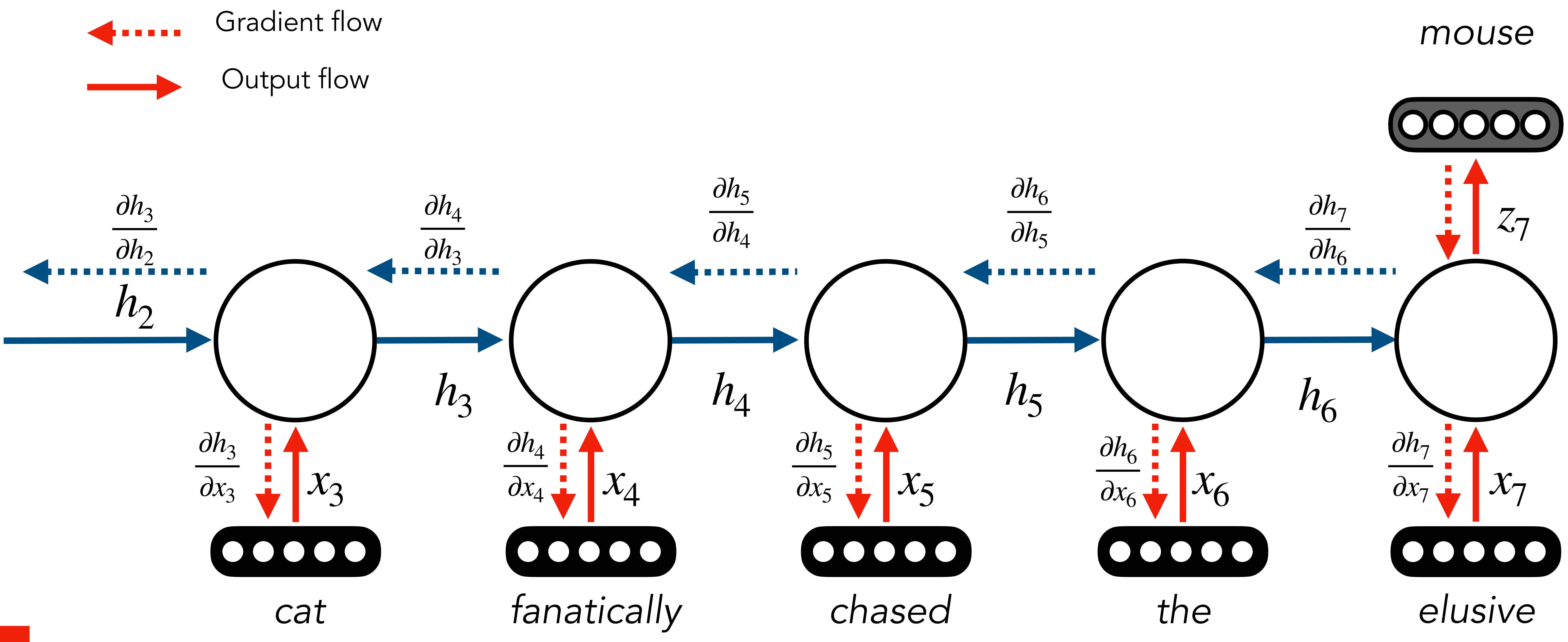
$$\frac{\partial h_{t-1}}{\partial h_{t-2}} = \frac{\partial \sigma(u_{t-1})}{\partial u_{t-1}} \frac{\partial u_{t-1}}{\partial h_{t-2}} = \frac{\partial \sigma(u_{t-1})}{\partial u_{t-1}} W_{hh}$$



$$\frac{\partial z_t}{\partial h_{t-1}} = \frac{\partial z_t}{\partial h_t} \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} = \frac{\partial \sigma(v_t)}{\partial v_t} W_{zh} \frac{\partial \sigma(u_t)}{\partial u_t} W_{hh} \frac{\partial \sigma(u_{t-1})}{\partial u_{t-1}} W_{hh}$$

Note that these are
actually the same matrix

Backpropagation through time



Summary

- Neural language models allow us to *share information* among similar sequences by learning neural representations that similarly represent them
- **Problem:** Fixed context language models can only process a limited window of the word history at a time
- **Solution:** recurrent neural networks can **theoretically** learn to model an **unbounded context length**

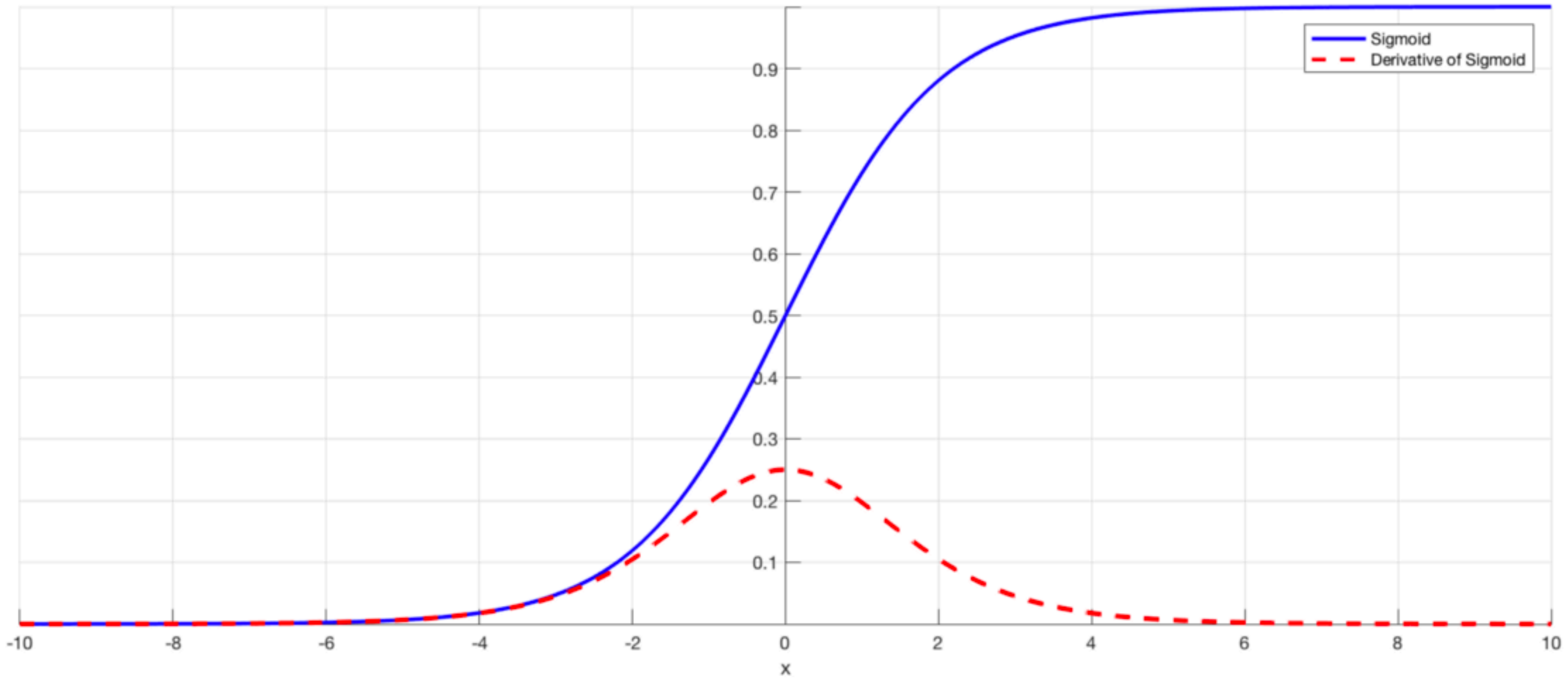
Vanishing Gradients

- **Learning Problem:** Long unrolled networks will crush gradients that backpropagate to earlier time steps

$$h_t = \boxed{\sigma}(W_{hx}x_t + W_{hh}\boxed{h_{t-1}} + b_h)$$

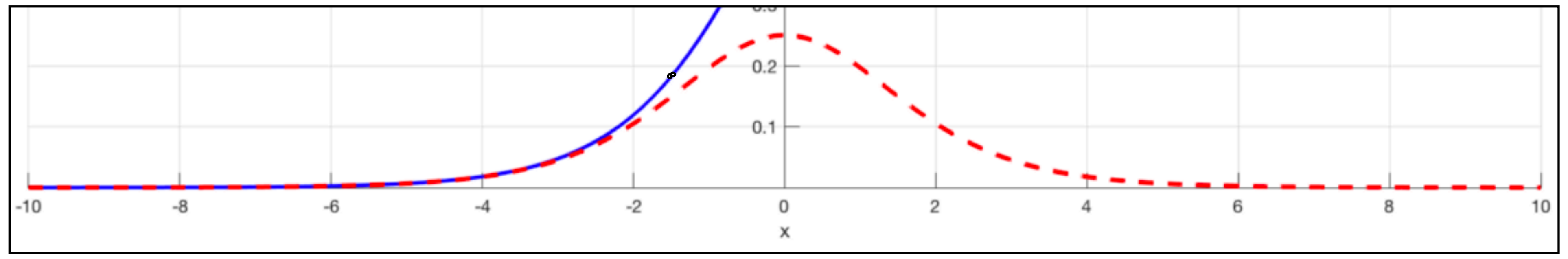
$$u = W_{hx}x_t + W_{hh}h_{t-1} + b_h$$

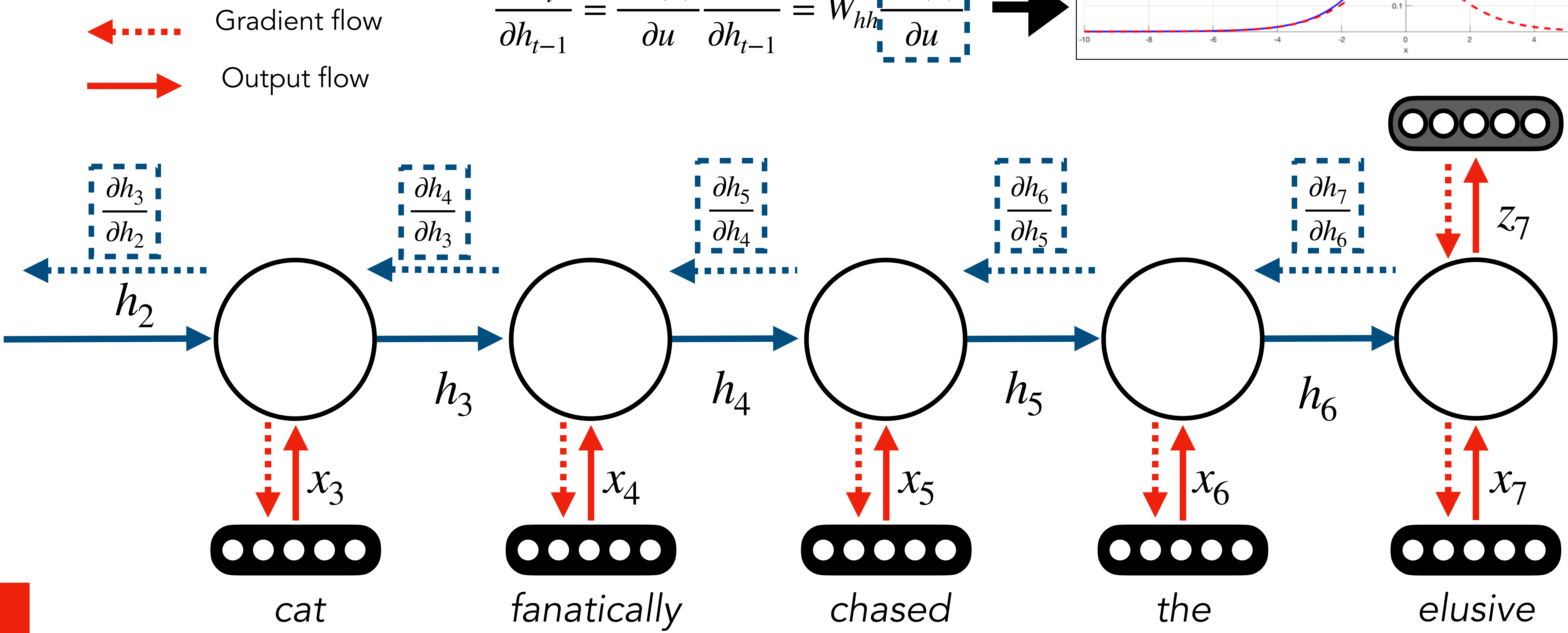
$$\frac{\partial h_t}{\partial h_{t-1}} = \frac{\partial \sigma(u)}{\partial u}\frac{\partial u}{\partial h_{t-1}} = W_{hh}\boxed{\frac{\partial \sigma(u)}{\partial u}}$$



Backpropagation through time

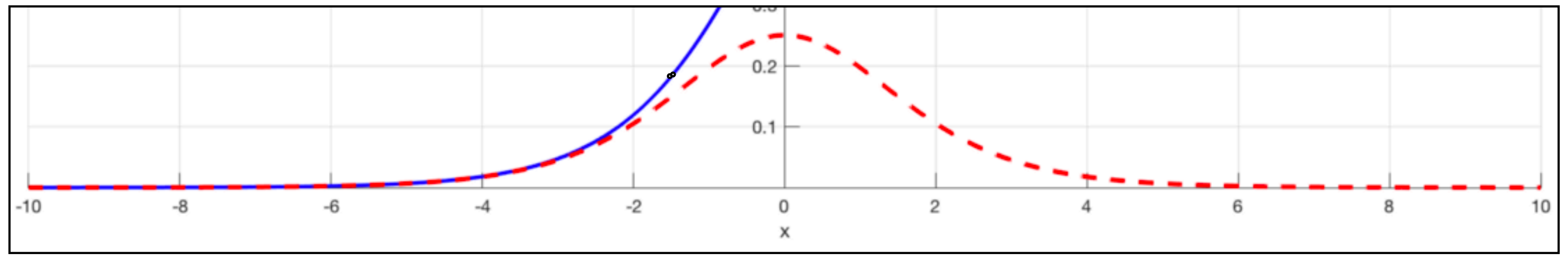
$$\frac{\partial h_t}{\partial h_{t-1}} = \frac{\partial \sigma(u)}{\partial u}\frac{\partial u}{\partial h_{t-1}} = W_{hh}\boxed{\frac{\partial \sigma(u)}{\partial u}}$$

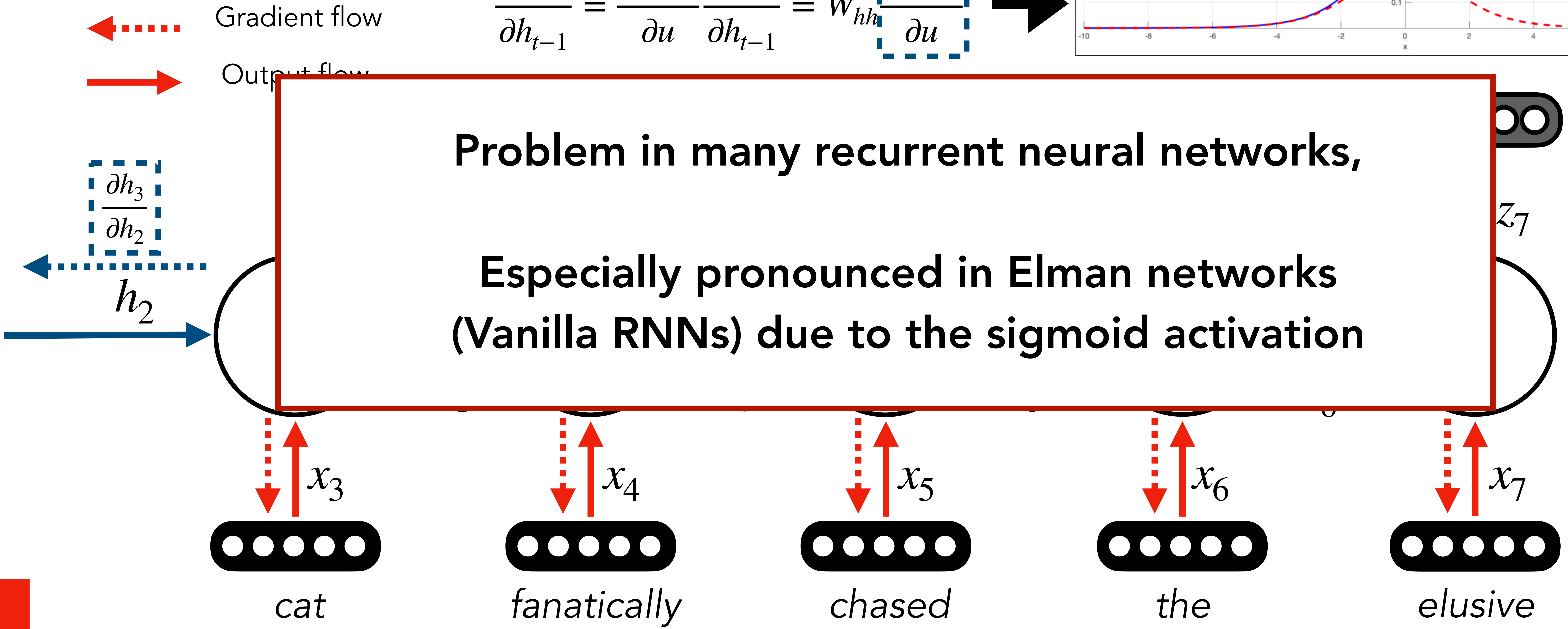




Backpropagation through time

$$\frac{\partial h_t}{\partial h_{t-1}} = \frac{\partial \sigma(u)}{\partial u}\frac{\partial u}{\partial h_{t-1}} = W_{hh}\boxed{\frac{\partial \sigma(u)}{\partial u}}$$





Problem in many recurrent neural networks,

Especially pronounced in Elman networks (Vanilla RNNs) due to the sigmoid activation

Issue with Recurrent Models

- Multiple steps of state overwriting makes it challenging to learn long-range dependencies.

They tuned, discussed for a moment, then struck up a lively ***jig****. Everyone joined in, turning the courtyard into an even more chaotic scene, people now* ***dancing*** *in circles,* ***swinging*** *and* ***spinning*** *in circles, everyone making up their own* ***dance steps****. I felt my feet tapping, my body wanting to move.*
Aside from writing, I 've always loved **<u>dancing</u>** *.*

- Nearby words should affect each other more than farther ones, but RNNs make it challenging to learn **<u>any</u>** long-range interactions

LAMBADA dataset, 2016

Gated Recurrent Neural Networks

- Use gates to avoid dampening gradient signal every time step

$$h_t = \sigma\left(W_{hx}x_t + W_{hh}h_{t-1} + b_h\right)$$

Elman Network

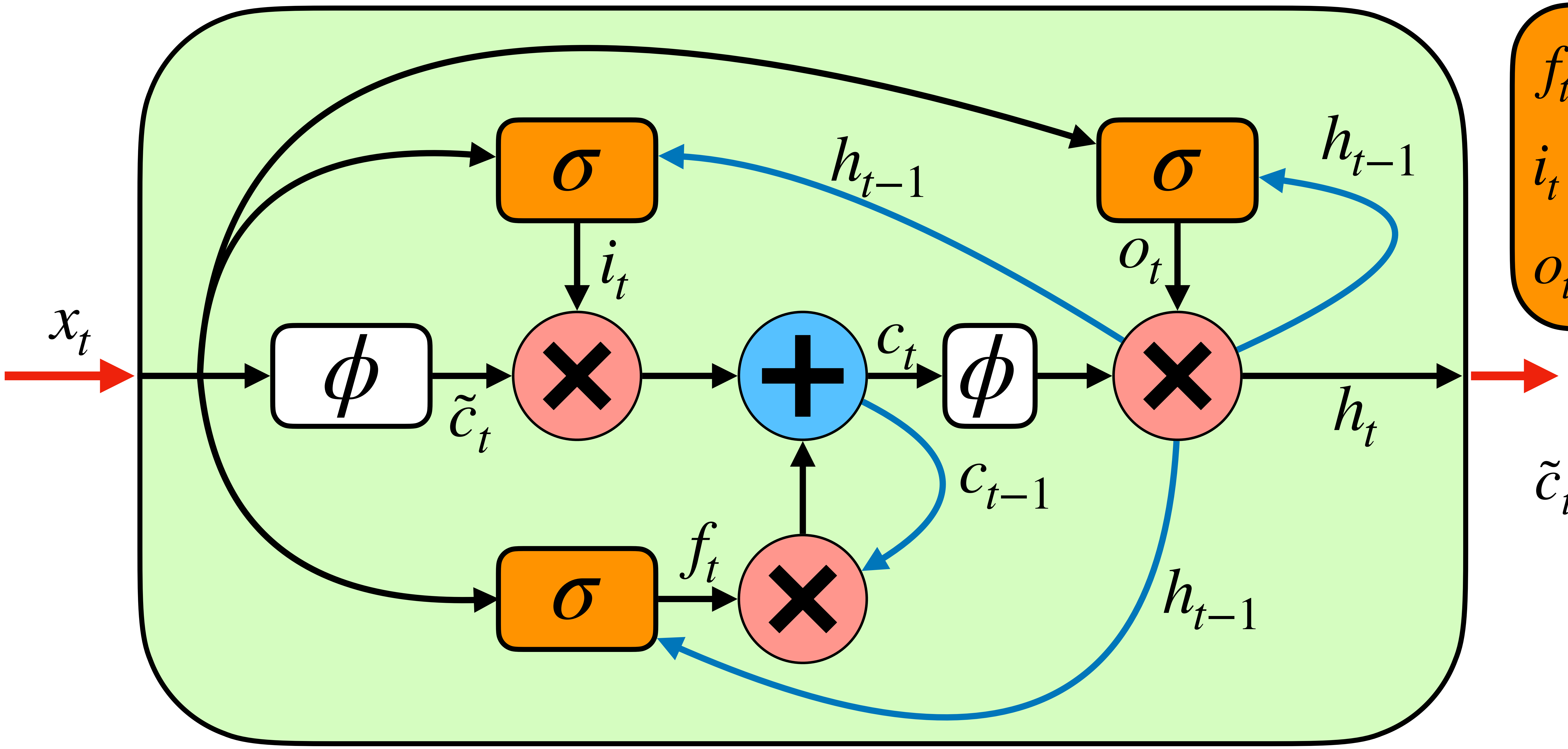
$$h_t = h_{t-1} \odot \mathbf{f} + \mathbf{func}(x_t)$$

Gated Network Abstraction

- Gate value **f** computes how much information from previous hidden state moves to the next time step —> 0 < **f** < 1

- Because h_{t-1} is no longer inside the activation function, it is not automatically constrained, reducing vanishing gradients!

Long Short Term Memory (LSTM)



Gates:

$$f_t = \sigma\left(W_{fx}x_t + W_{fh}h_{t-1} + b_f\right)$$
$$i_t = \sigma\left(W_{ix}x_t + W_{ih}h_{t-1} + b_i\right)$$
$$o_t = \sigma\left(W_{ox}x_t + W_{oh}h_{t-1} + b_o\right)$$

$$\tilde{c}_t = \phi\left(W_{cx}x_t + W_{ch}h_{t-1} + b_c\right)$$
$$c_t = i_t \times \tilde{c}_t + f_t \times c_{t-1}$$
$$h_t = o_t \times \phi(c_t)$$

(Hochreiter and Schmidhuber, 1997)

Question

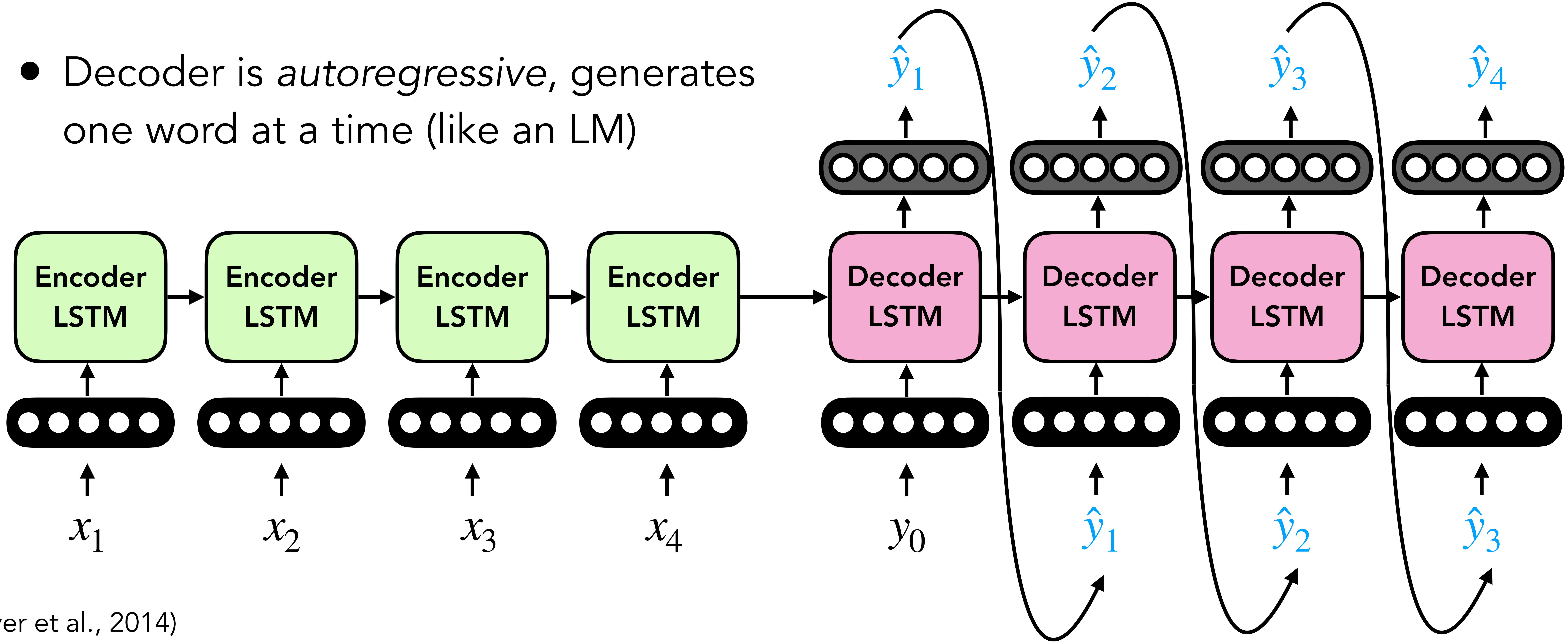
How can we use recurrent neural networks in practice?

Machine Translation involves more than estimating the probability next word; requires generating a full translation of a given context into another language

Encoder-Decoder Models

- Encode a sequence fully with one model (**encoder**) and use its representation to seed a second model that decodes another sequence (**decoder**)

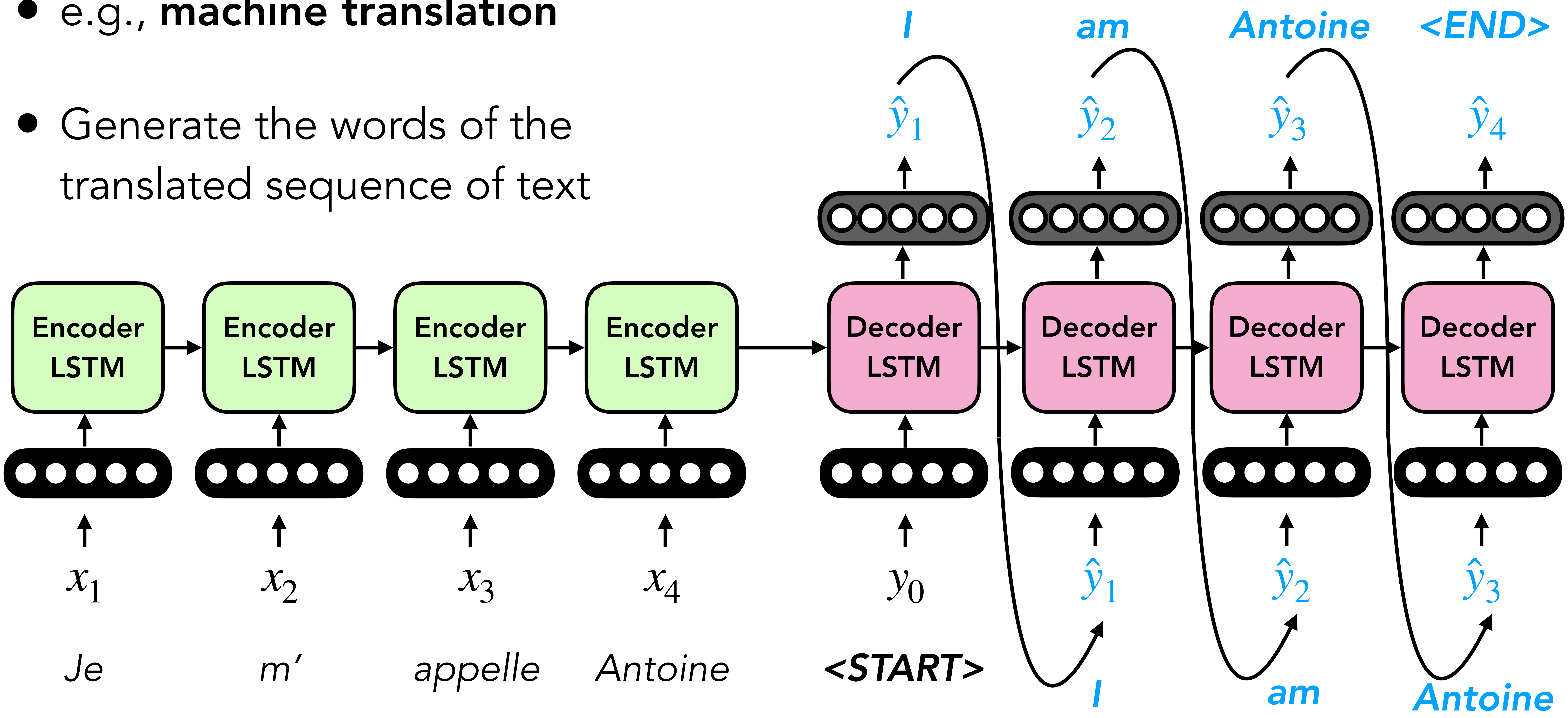
- Decoder is *autoregressive*, generates one word at a time (like an LM)



(Sutskever et al., 2014)

Encoder-Decoder Models

- e.g., **machine translation**
- Generate the words of the translated sequence of text

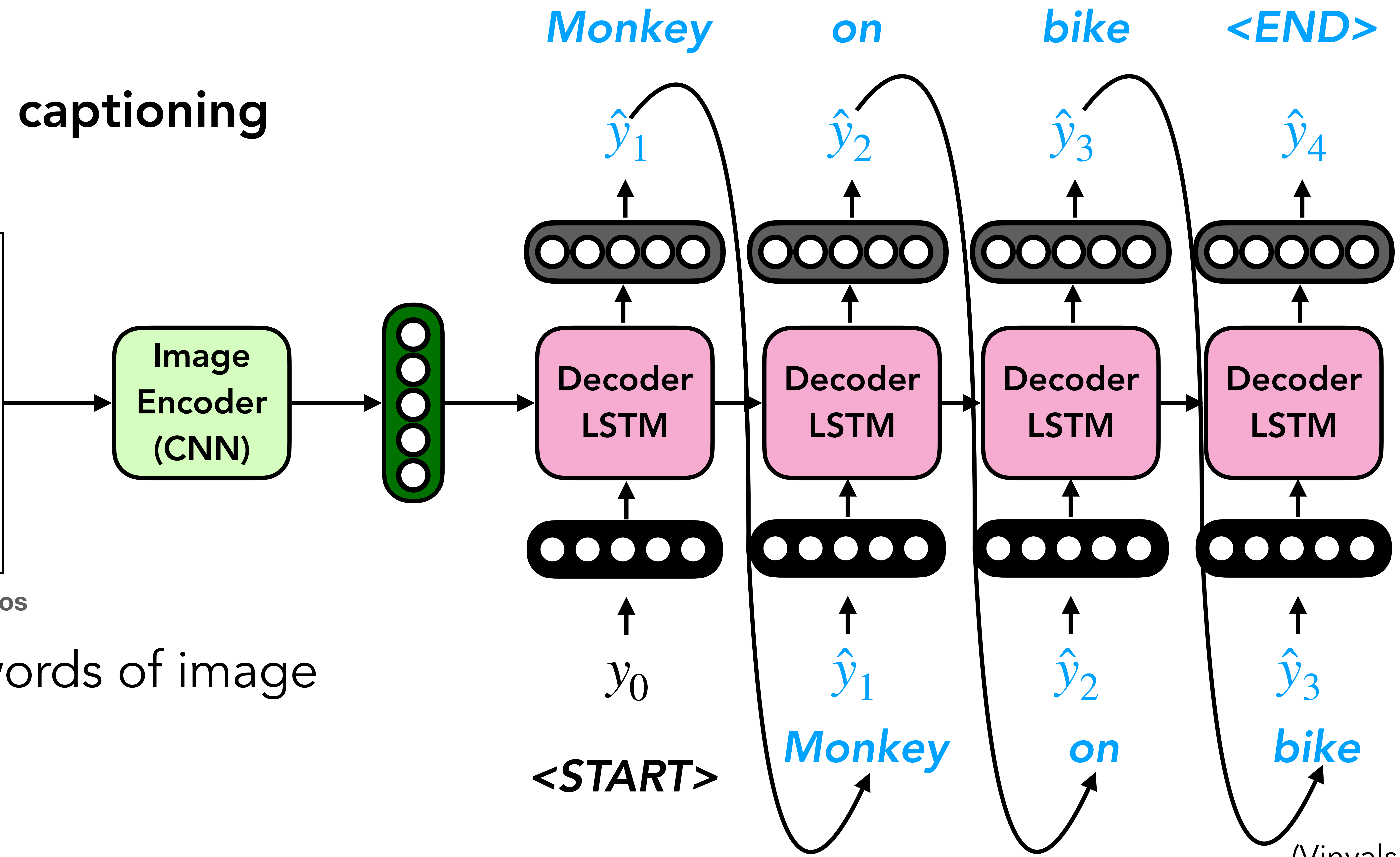


(Sutskever et al., 2014)

Encoder-Decoder Models

- Input doesn't need to be text
- e.g., **image captioning**

Photo credit: J Hovenstine Studios

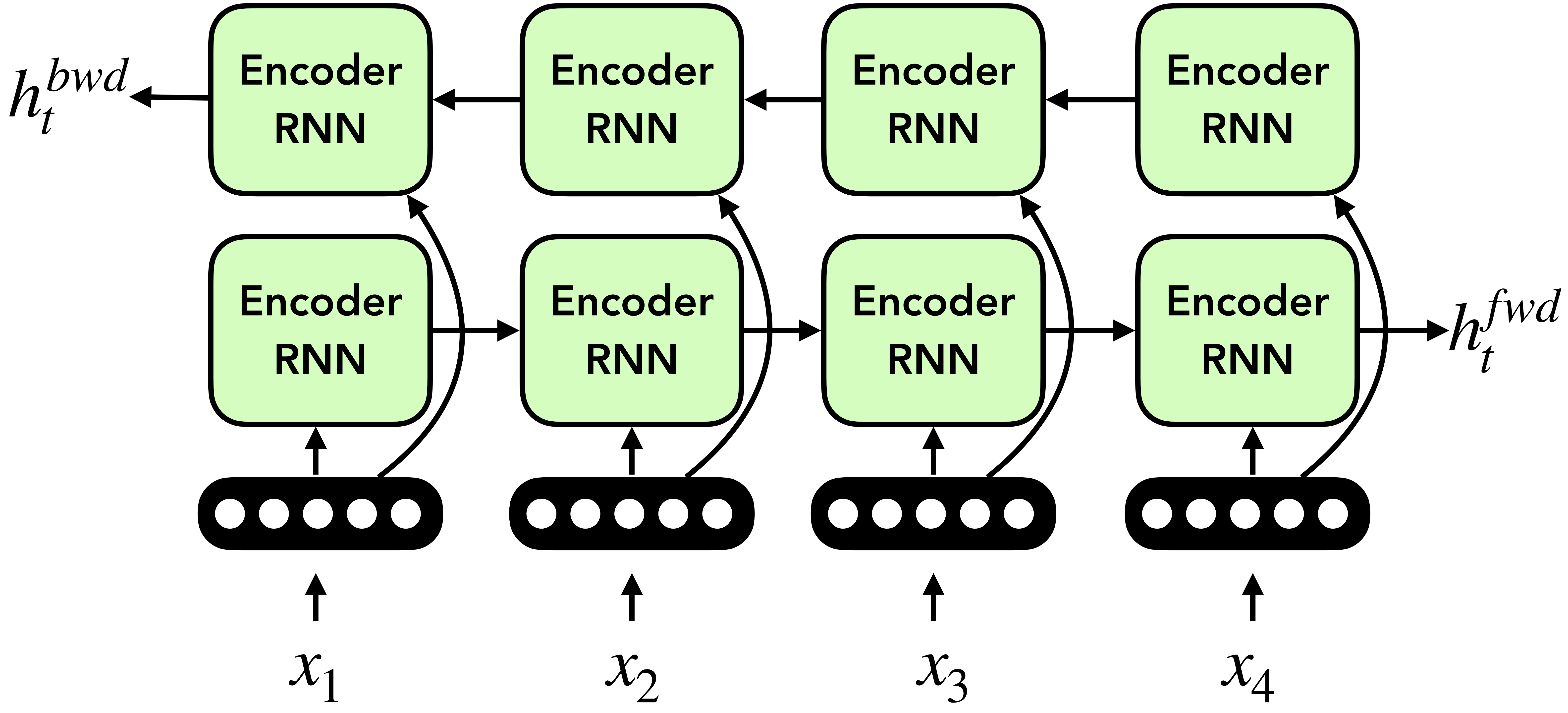


- Generate words of image description

(Vinyals et al., 2014)

Bidirectional Encoders

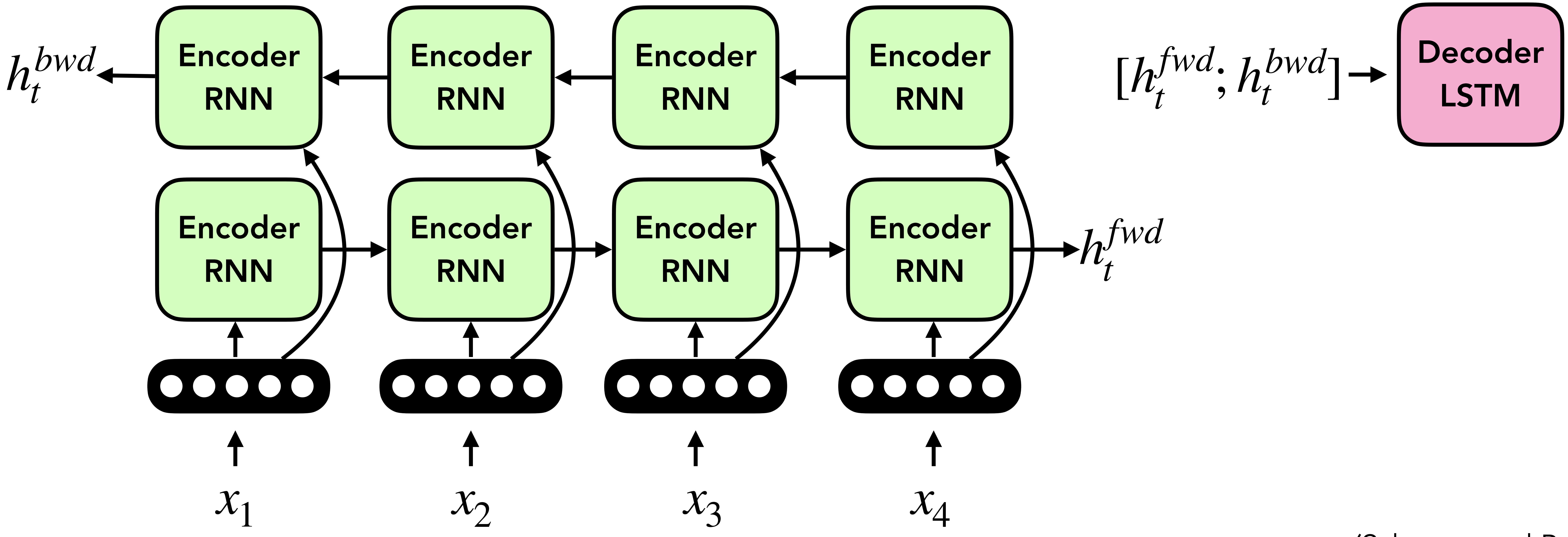
- Decoder needs to be unidirectional (can't know the future…)
- Encoder sequence representation augmented by encoding in both directions



(Schuster and Paliwal, 1997)

Bidirectional Encoders

- Decoder needs to be unidirectional (can't know the future…)
- Encoder sequence representation augmented by encoding in both directions



(Schuster and Paliwal, 1997)

Other Resources of Interest

- Approaches for maintaining state and avoiding vanishing gradients
  - Long Short-Term Memory (Hochreiter and Schmidhuber, 1997):
  - Gated Recurrent Units (Cho et al., 2014):
- LSTM: A Search Space Odyssey (Greff et al., 2015)
  - Examine 5000 different modifications to LSTMs — none significantly better than original architecture
- Only basics presented here today! Many offshoots of these techniques!

Recap

- Early neural language models (and n-gram models) suffer from **fixed context windows**

- Recurrent neural networks can **theoretically** learn to model an **unbounded context length** using back propagation through time (BPTT)

- Practically, however, **vanishing gradients** stop many RNN architectures from learning **long-range dependencies**

- RNNs (and modern variants) remain useful for **many sequence-to-sequence tasks**

References

- Bengio, Y., Ducharme, R., Vincent, P., & Janvin, C. (2003). A Neural Probabilistic Language Model. *Journal of machine learning research*.
- Elman, J.L. (1990). Finding Structure in Time. *Cogn. Sci., 14*, 179-211.
- Schuster, M., & Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, *45*(11), 2673–2681.
- Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation, 9*, 1735-1780.
- Cho, K., Merrienboer, B.V., Gülçehre, Ç., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. *Conference on Empirical Methods in Natural Language Processing*.
- Sutskever, I., Vinyals, O., & Le, Q.V. (2014). Sequence to Sequence Learning with Neural Networks. *NIPS*.
- Vinyals, O., Toshev, A., Bengio, S., & Erhan, D. (2014). Show and tell: A neural image caption generator. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 3156-3164.
- Greff, K., Srivastava, R.K., Koutník, J., Steunebrink, B.R., & Schmidhuber, J. (2015). LSTM: A Search Space Odyssey. *IEEE Transactions on Neural Networks and Learning Systems, 28*, 2222-2232.