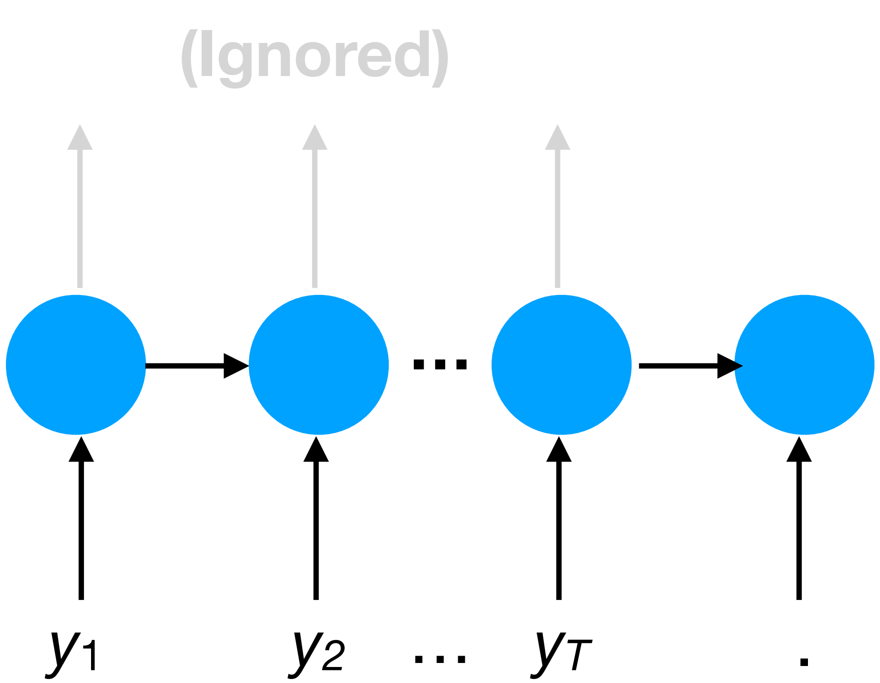
CS/DS 552: Class 15

Jacob Whitehill

RNNs for machine translation

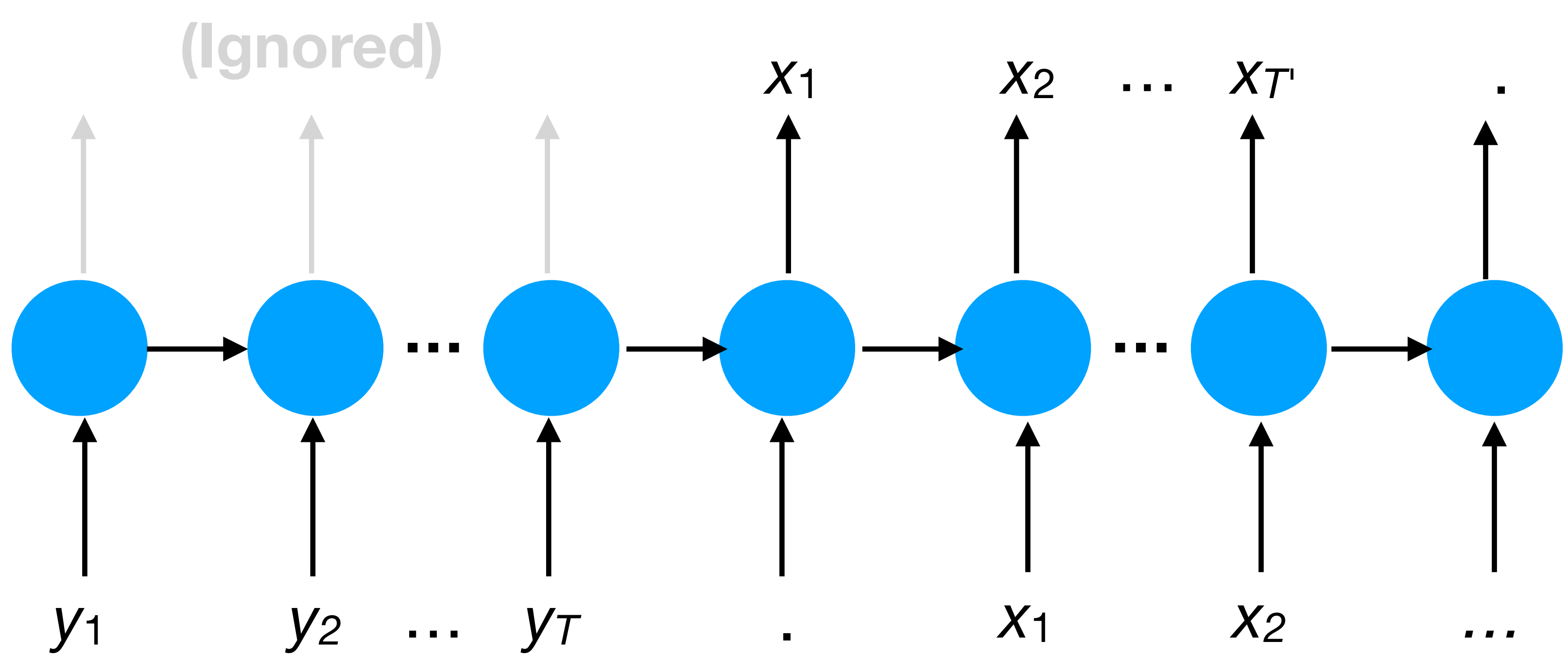
RNNs for machine translation

- We can construct an RNN to translate from a source language to the target language:
  1. We first input the T words of the input sentence as y_1, …, y_T, followed by .



RNNs for machine translation

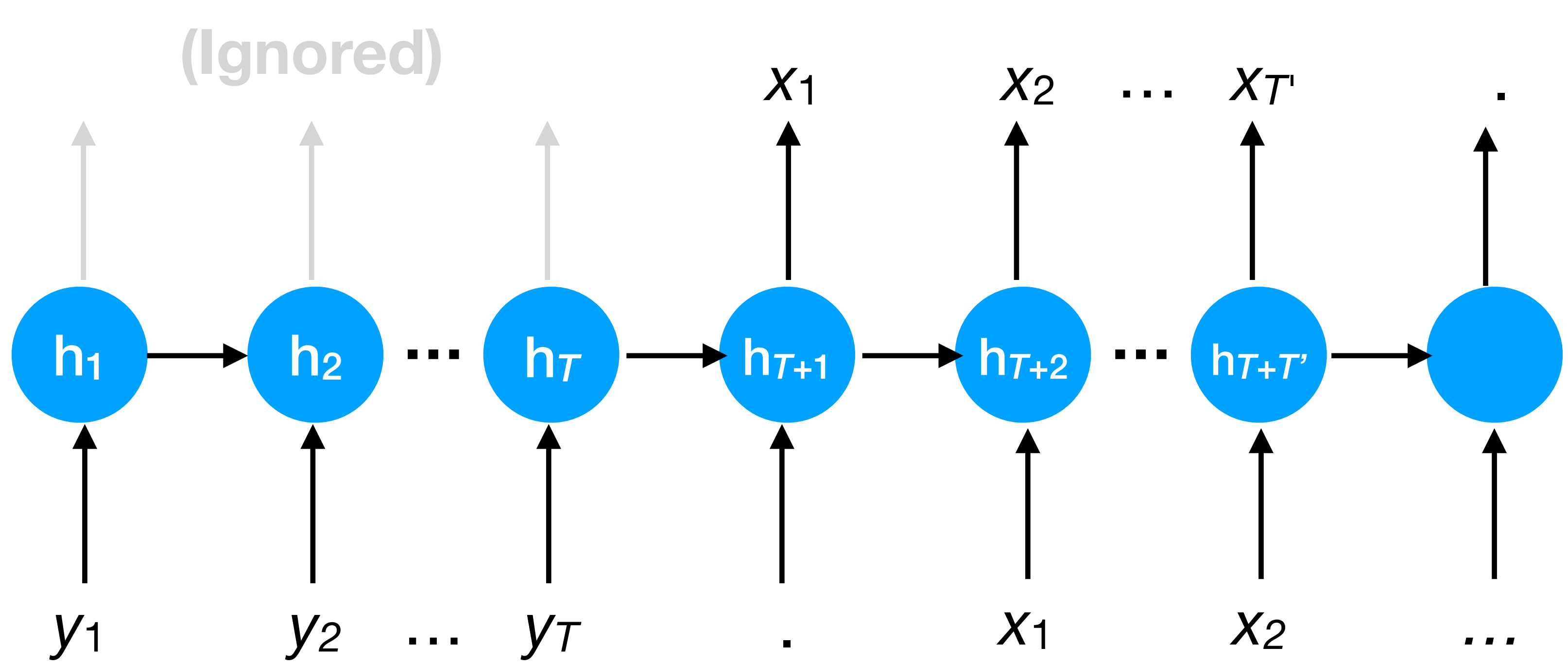
- We can construct an RNN to translate from a source language to the target language:
  1. We first input the T words of the input sentence as y_1, …, y_T, followed by .
  2. We then obtain the T' words of the output sentence autoregressively as x_1, …, $x_{T'}$, until the model outputs .



RNNs for machine translation

- The RNN's hidden state $\mathbf{h}_t$ captures both the **meaning of the input y** and the **summary of the output up to time *t***.

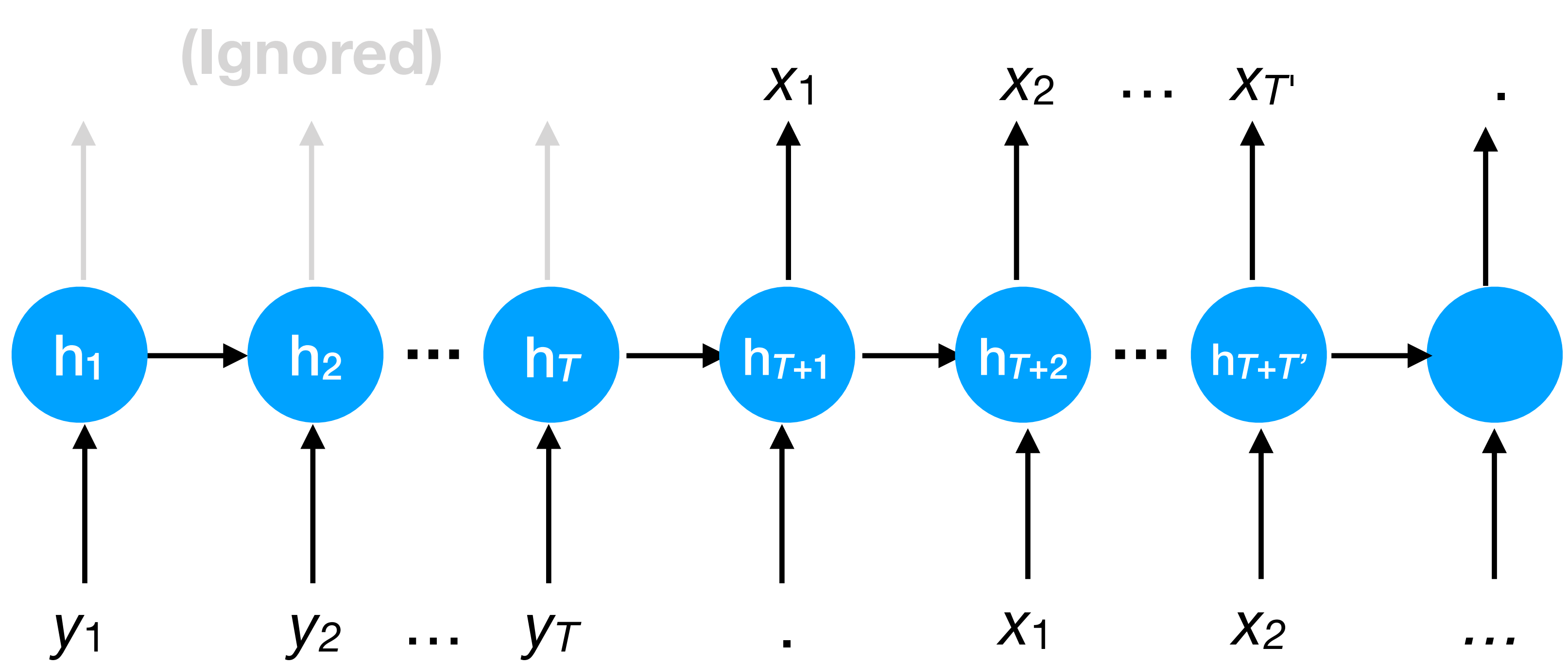
- $\mathbf{h}_t$ tries to "compress" the variable-length history into a fixed-length representation (i.e., $O(1)$ space).



RNNs for machine translation

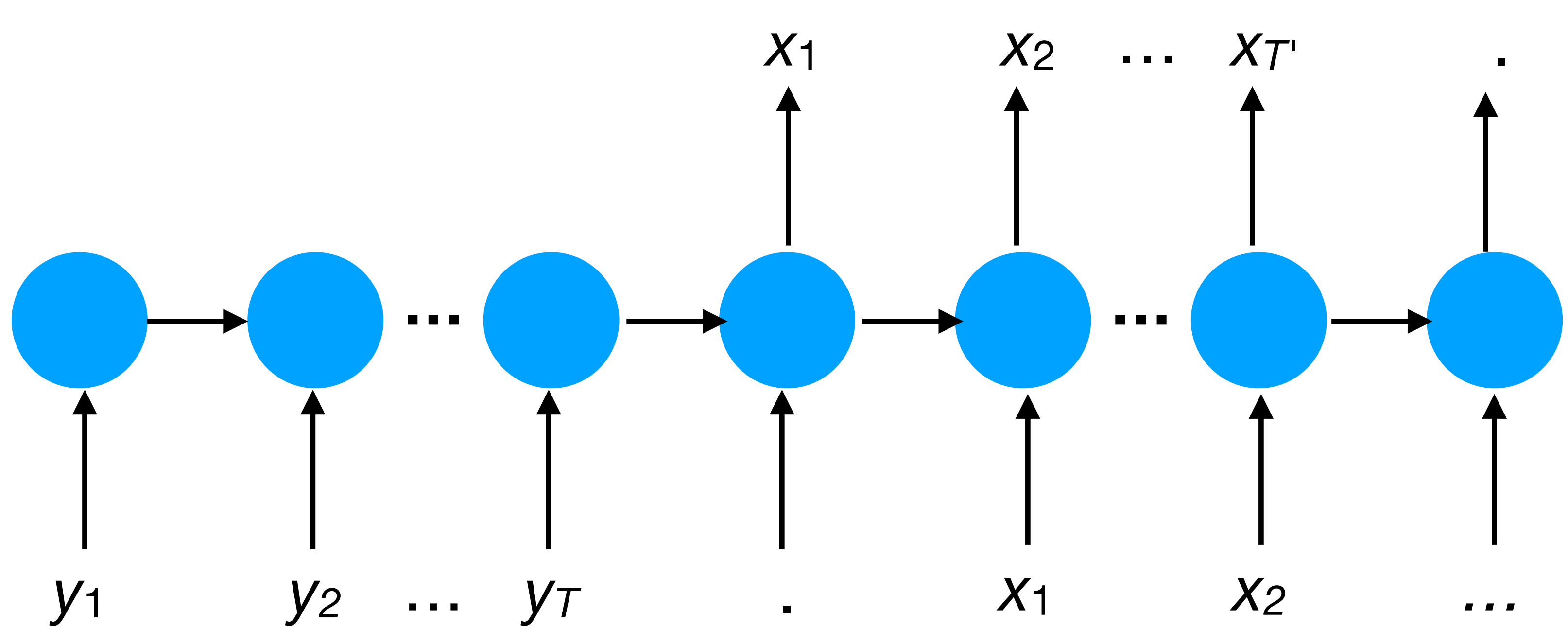
- This approach can work (and is the idea behind many LLMs) but typically requires a very large model to be successful.

- Instead, it can be beneficial to break the machine translation problem into subtasks: “encoding” and “decoding”.



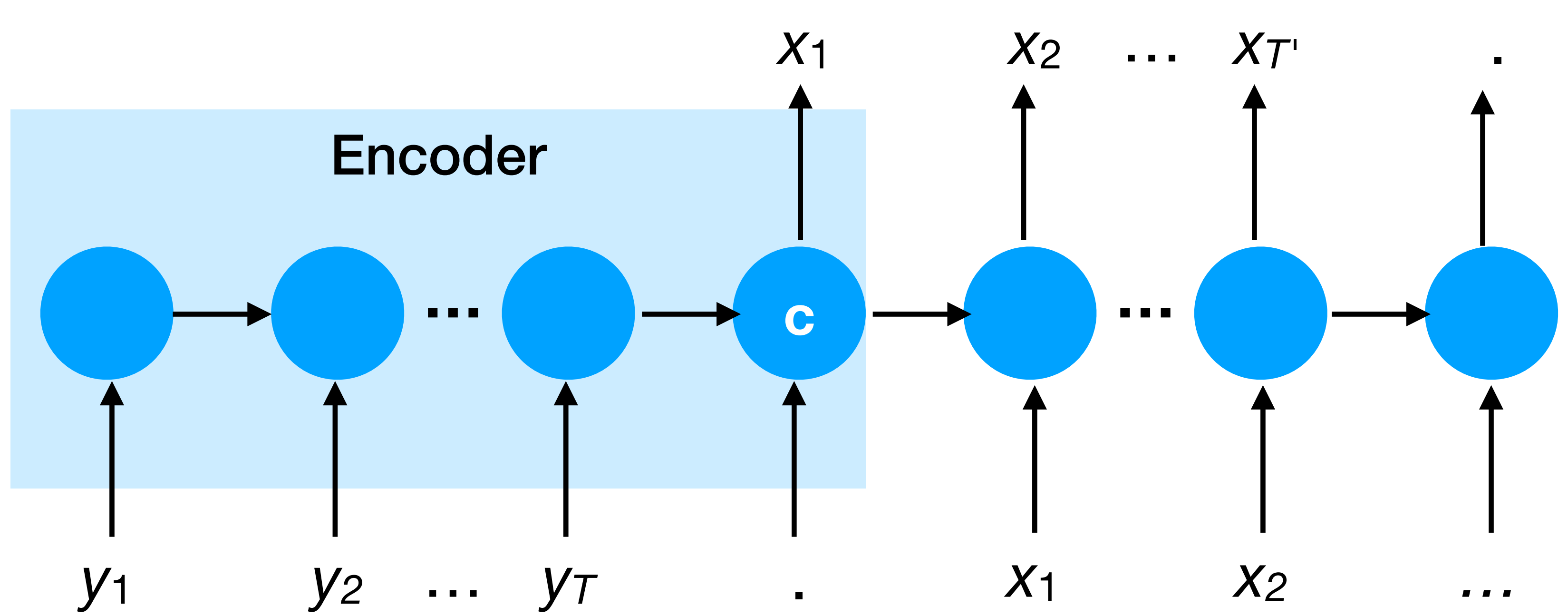
Encoder-Decoder models

- We construct a sequence-to-sequence model consisting of an **encoder** RNN and a **decoder** RNN:



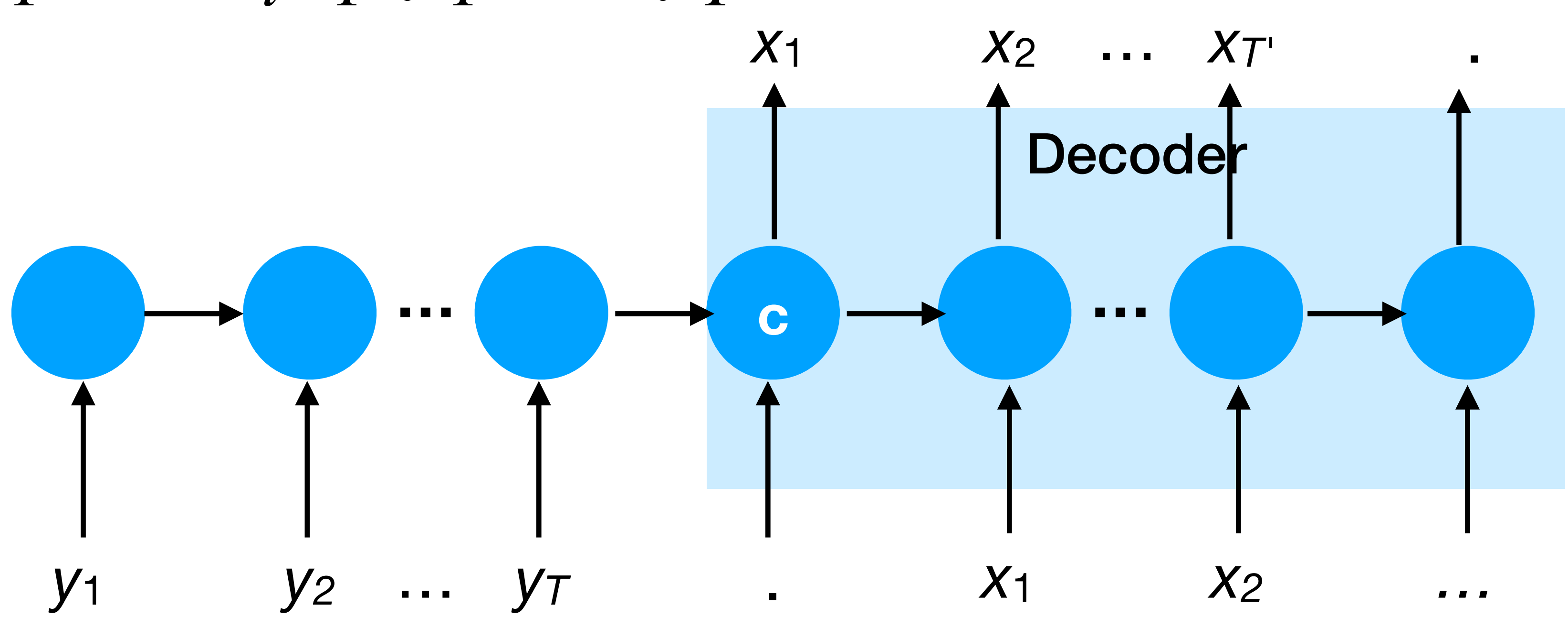
Encoder-Decoder models

- The encoder ingests the input sequence $y_1, \ldots, y_T$ and produces a context vector **c** that captures **y**'s meaning.



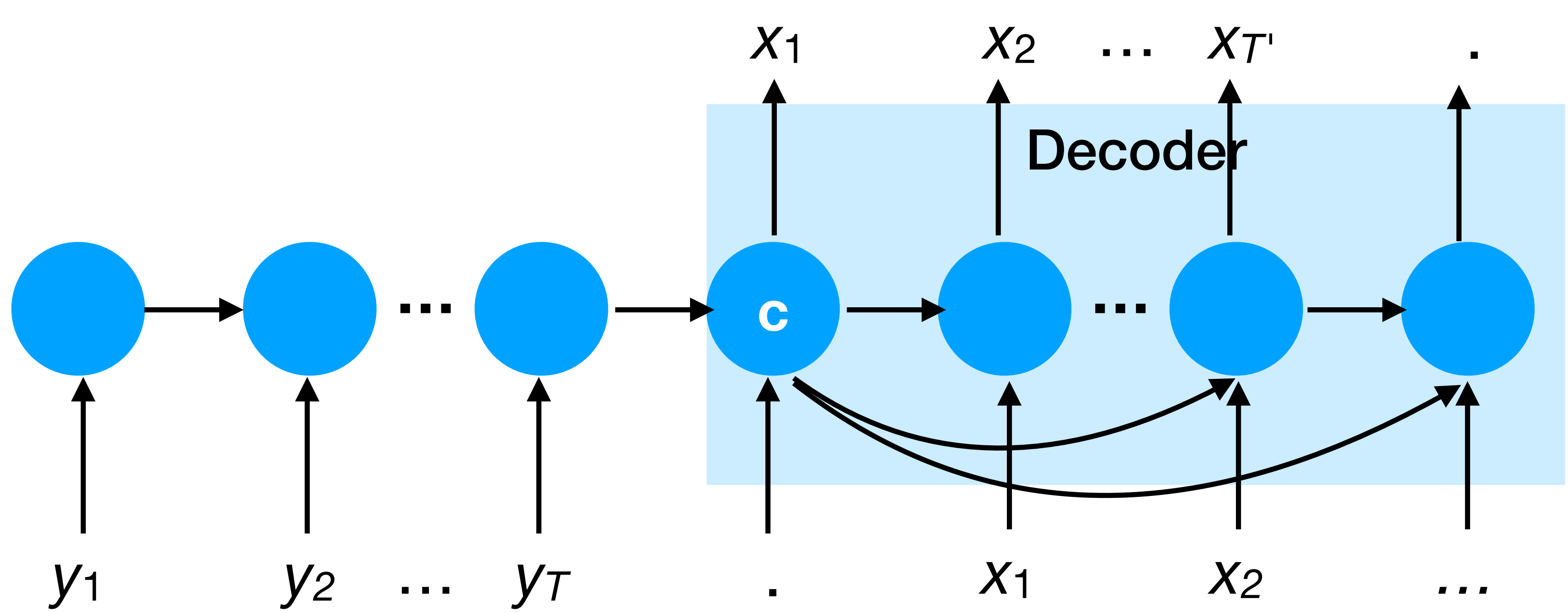
Encoder-Decoder models

- The encoder ingests the input sequence $y_1, \ldots, y_T$ and produces a context vector **c** that captures **y**'s meaning.
- The decoder uses the context vector to estimate $P(x_t \mid x_1, \ldots, x_{t-1}, y_1, \ldots, y_T)$ at each timestep t.



Encoder-Decoder models

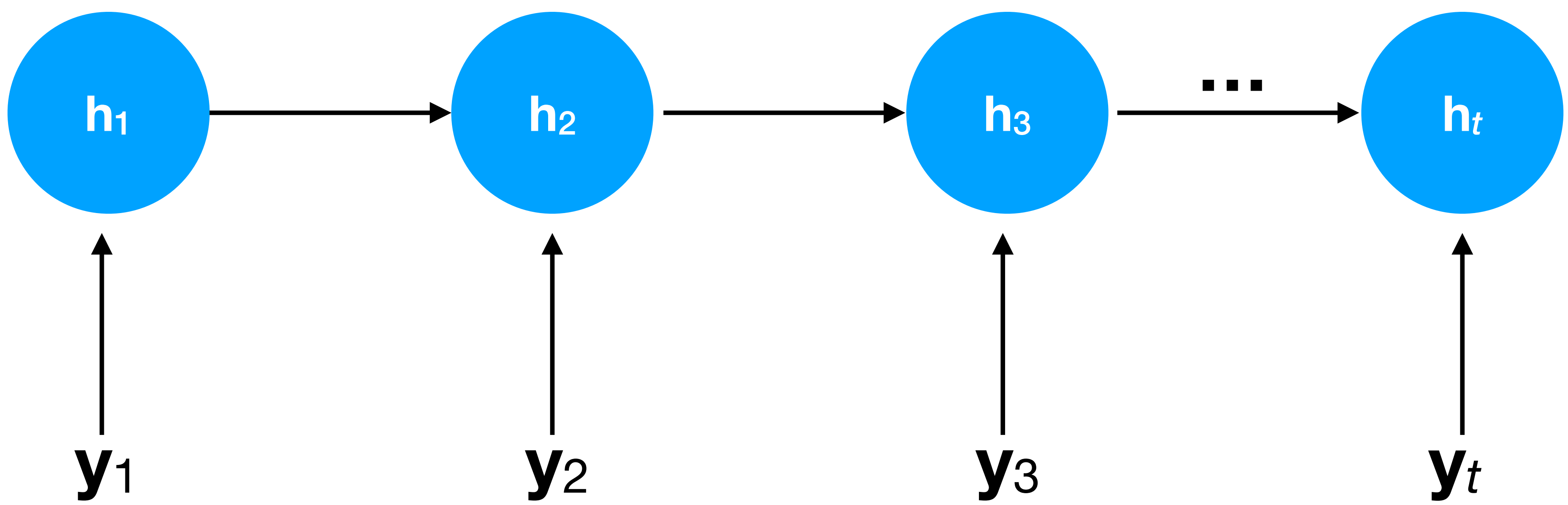
- Alternatively, we can feed **c** explicitly to *all* timesteps of the decoder, thus giving more direct access to the input:



Limitations of basic RNNs

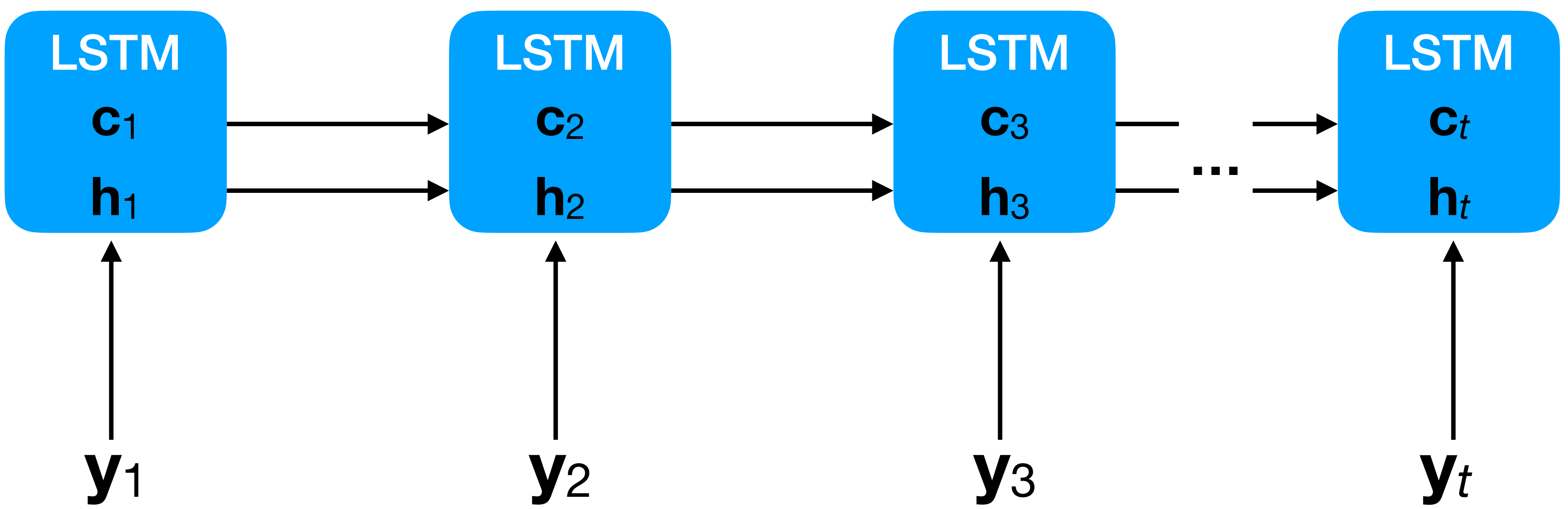
- In practice, the hidden states { $\mathbf{h}_t$ } of basic RNNs have difficulty storing information long-term.

- Basic RNNs are also highly unstable to train.



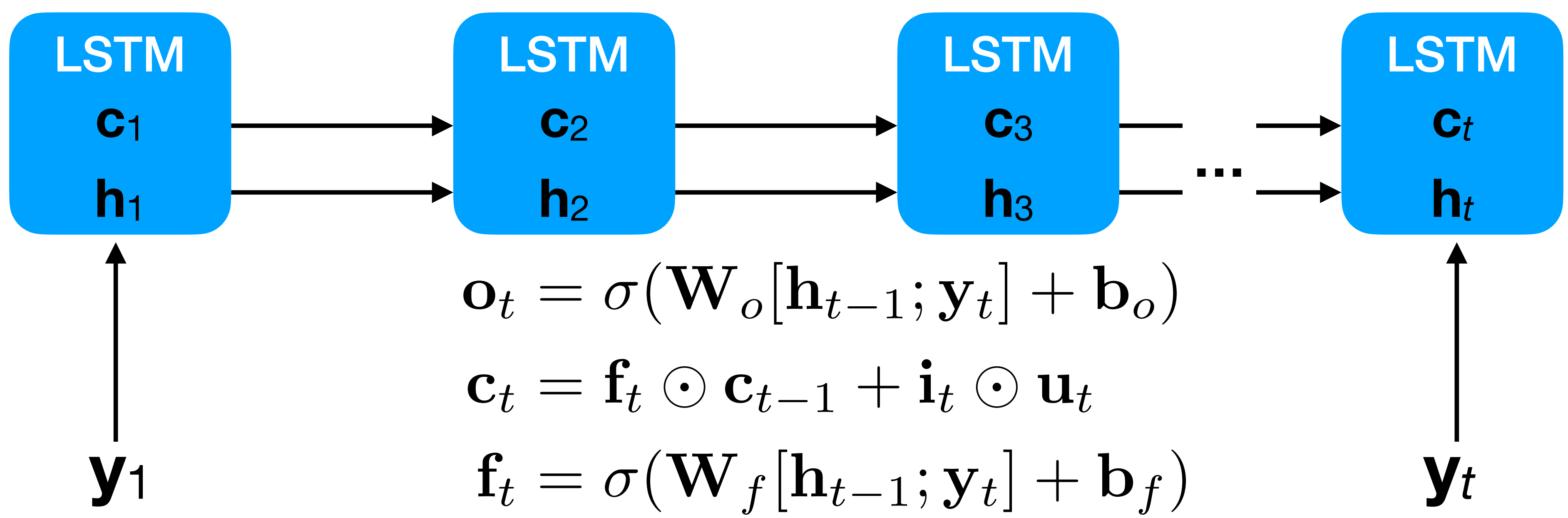
LSTM RNNs

- A **long short-term memory (LSTM) RNN** improves on this by making it easy to store information over long timespans.
- It contains both a hidden state $\mathbf{h}_t$ and a **cell state** $\mathbf{c}_t$, using the input $\mathbf{y}_t$.



LSTM RNNs

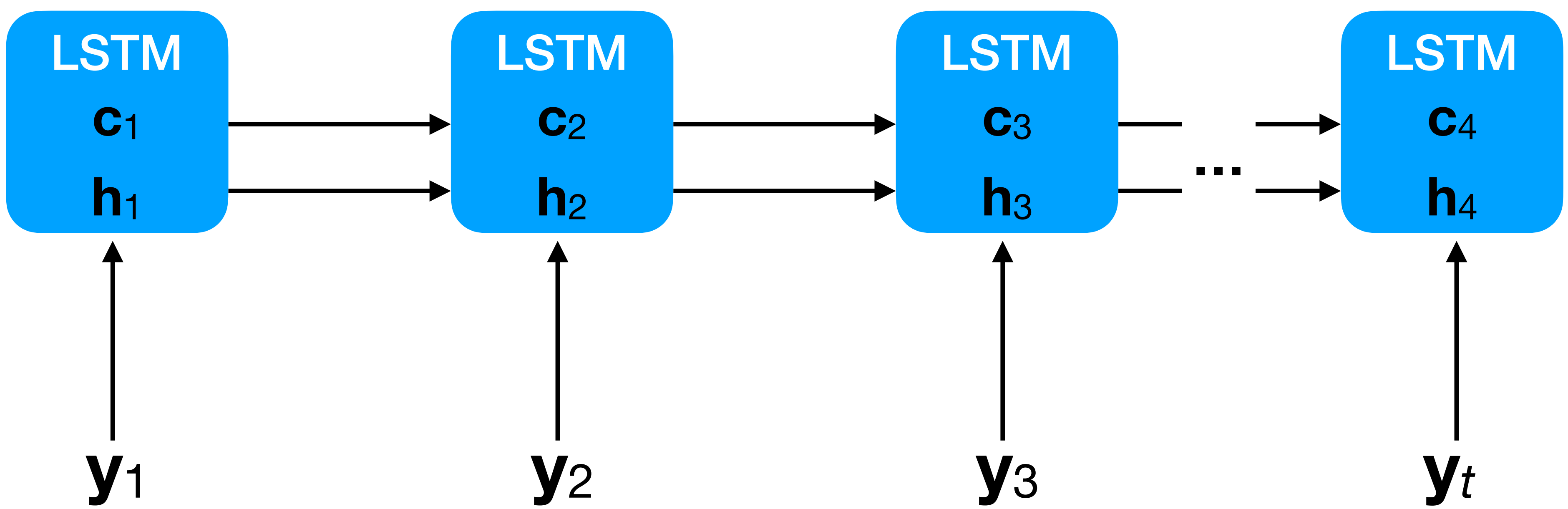
- **Input gates i**$_t$ control what parts of input **y**$_t$ are allowed into **c**$_t$.
- **Forgetting gates f**$_t$ control what parts of **c**$_{t\text{-}1}$ are allowed into **c**$_t$.
- **Output gates o**$_t$ control what parts of **c**$_t$ are allowed into **h**$_t$.



$$\mathbf{o}_t = \sigma(\mathbf{W}_o[\mathbf{h}_{t-1};\mathbf{y}_t] + \mathbf{b}_o)$$
$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{u}_t$$
$$\mathbf{f}_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1};\mathbf{y}_t] + \mathbf{b}_f)$$
$$\mathbf{u}_t = \tanh(\mathbf{W}_u[\mathbf{h}_{t-1};\mathbf{y}_t] + \mathbf{b}_u)$$
$$\mathbf{i}_t = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1};\mathbf{y}_t] + \mathbf{b}_i)$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

LSTM RNNs

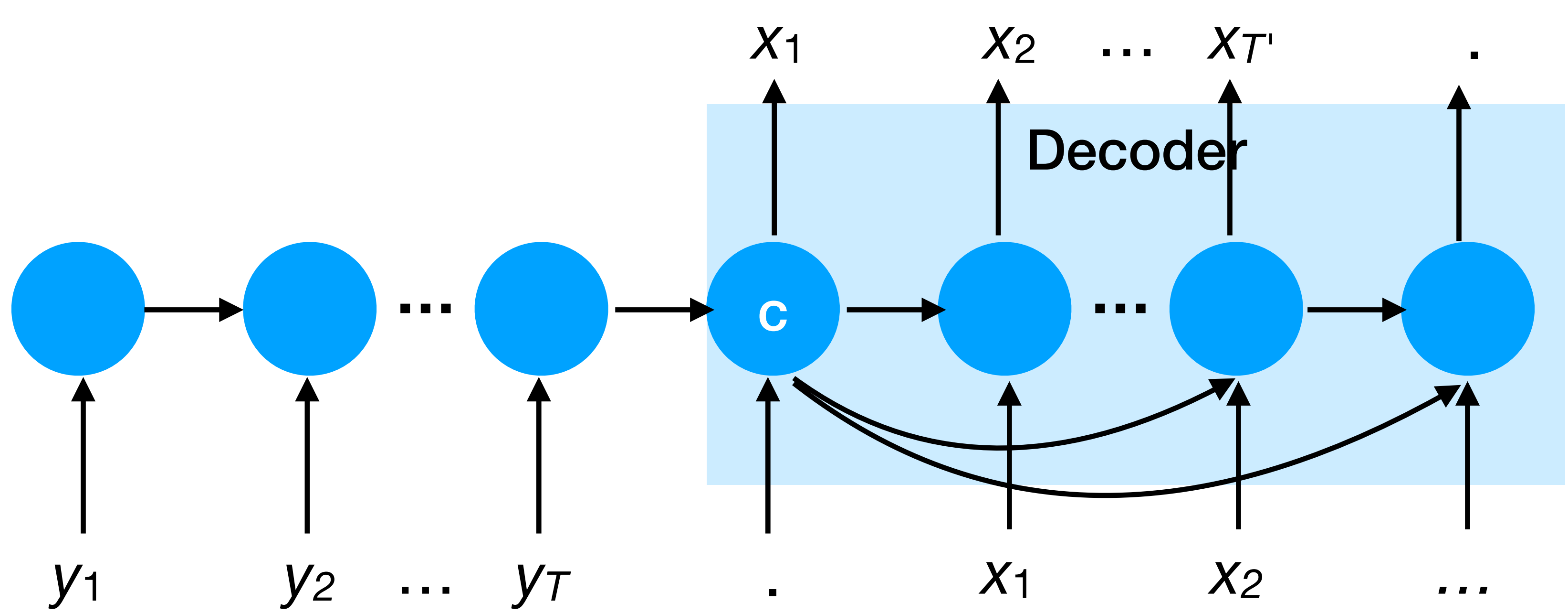
- In practice, LSTMs are much easier to train than basic RNNs.
- Memory cell $\mathbf{c}_t$ selectively stores & forgets information from the input.
- It is still limited since it tries to summarize an entire history in one vector.



Neural attention models

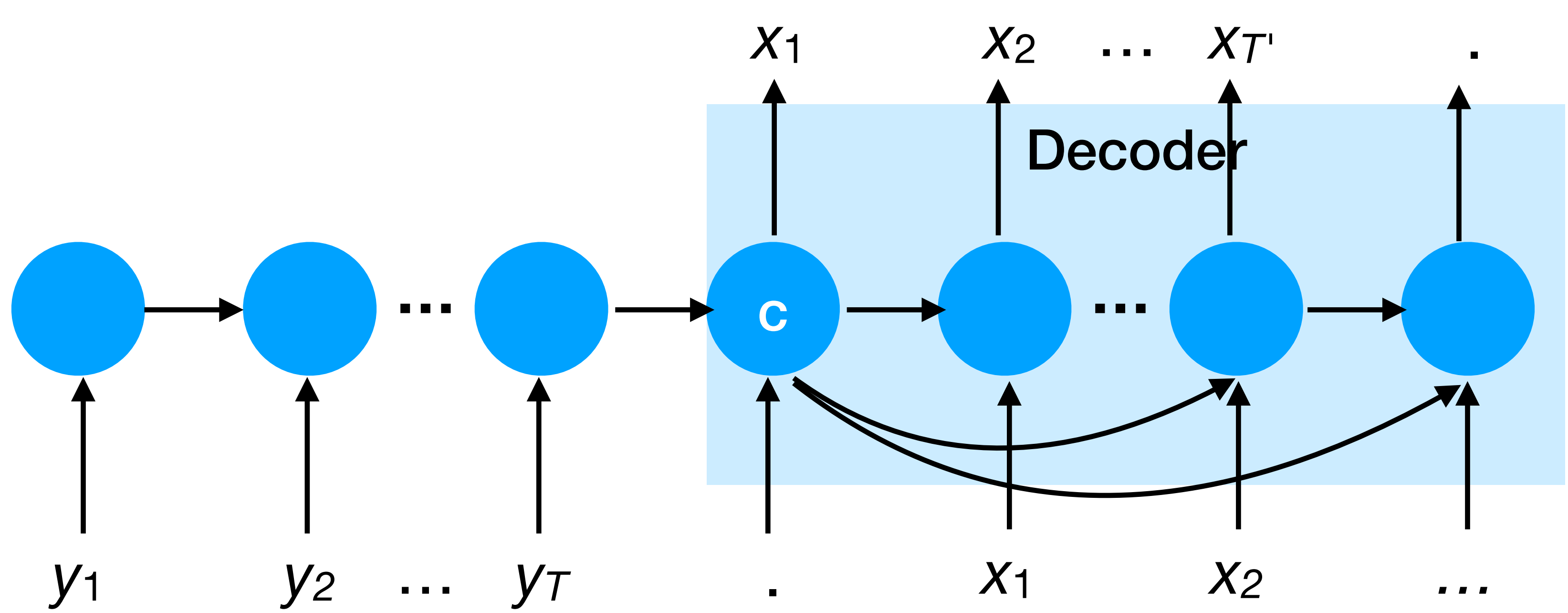
Encoder-Decoder models

- While elegantly simple, encoder-decoder RNNs may struggle to fit all information about **y** into a single vector **c**.



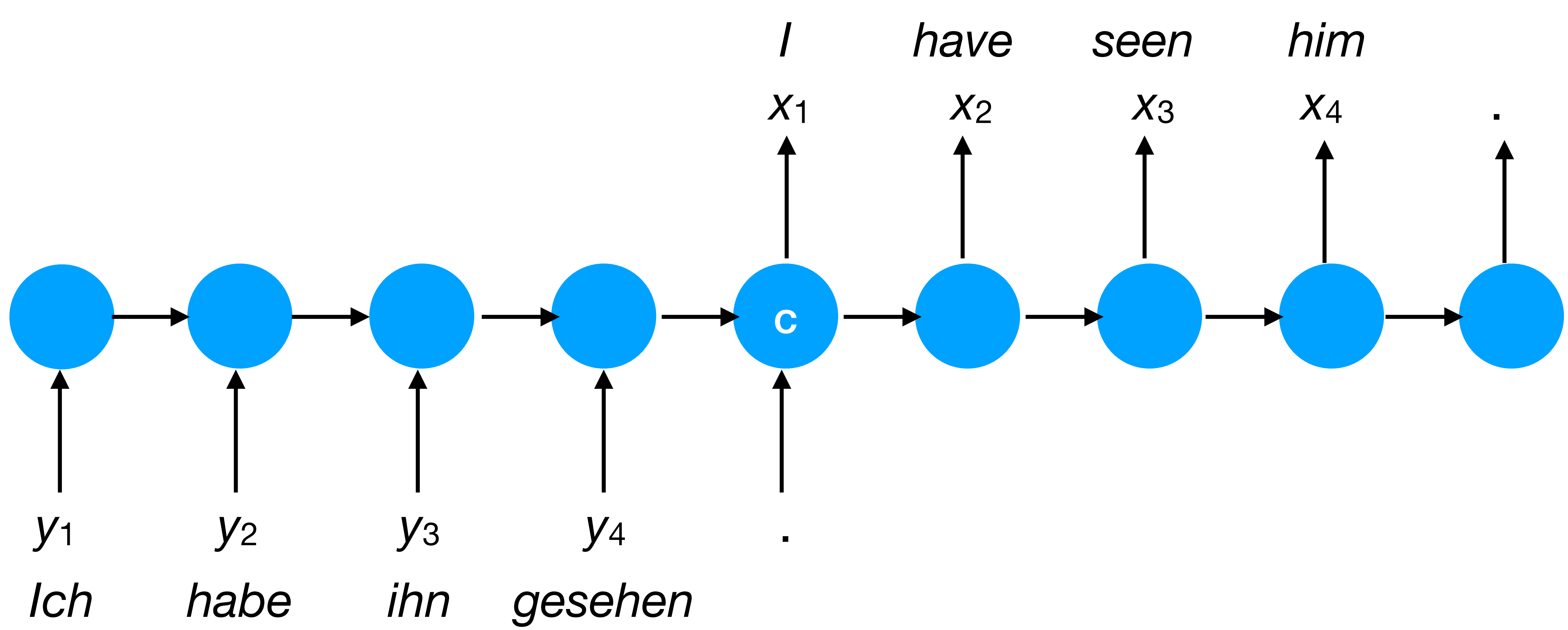
Encoder-Decoder models

- Intuitively, some parts of the input may be more relevant than others when producing a particular output symbol x_t.



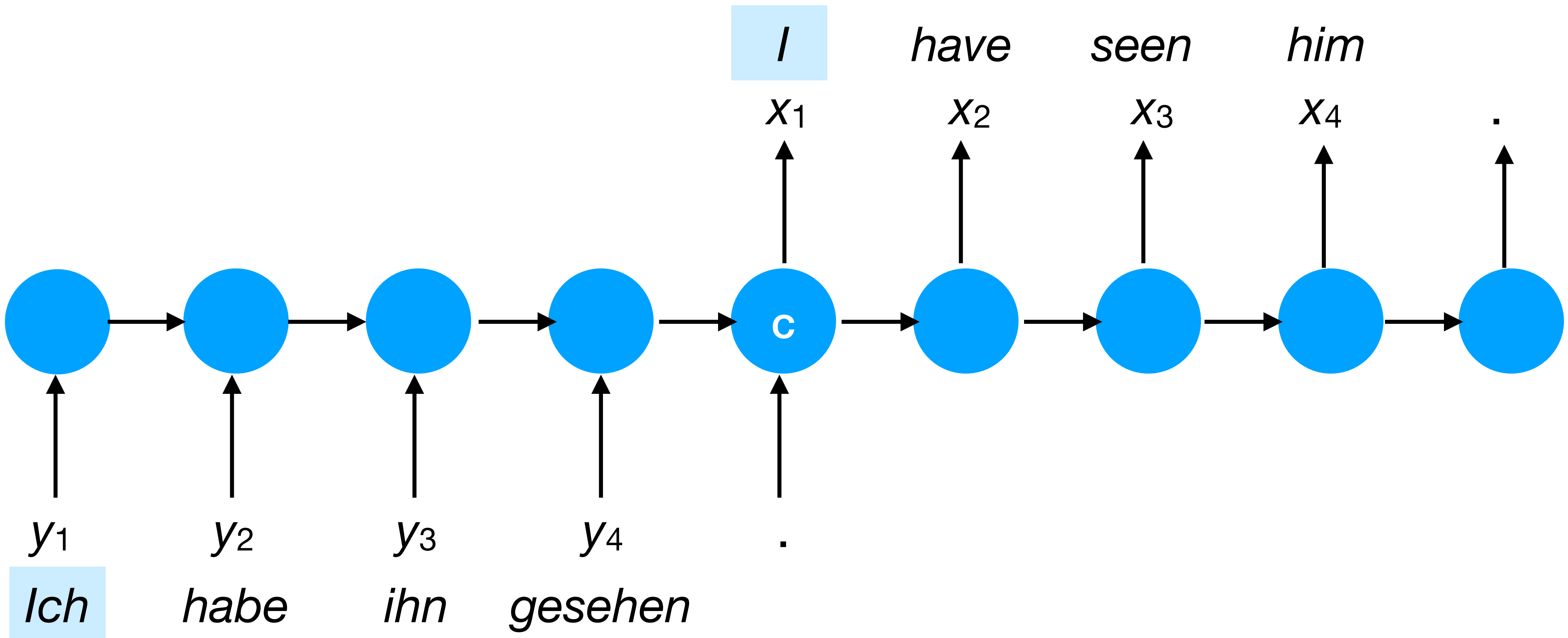
Encoder-Decoder models

- Intuitively, some parts of the input may be more relevant than others when producing a particular output symbol x_t.



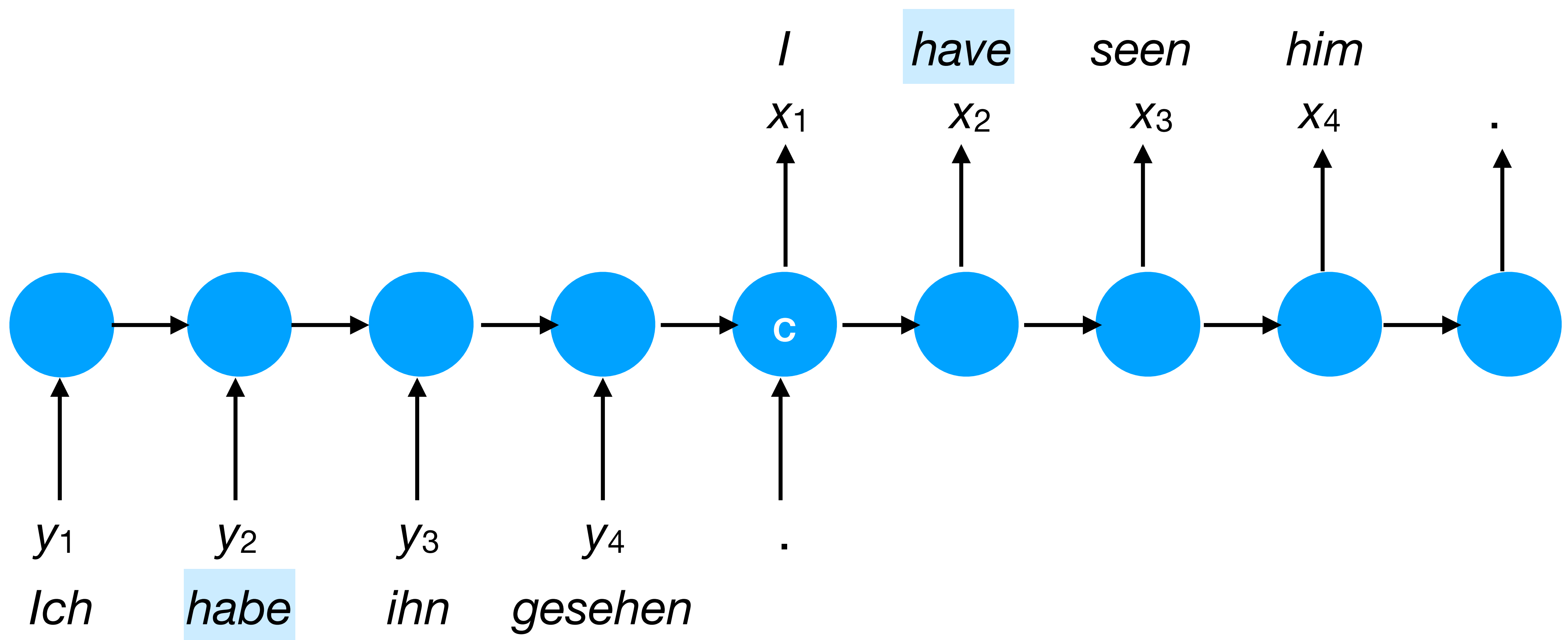
Encoder-Decoder models

- Intuitively, some parts of the input may be more relevant than others when producing a particular output symbol x_t.



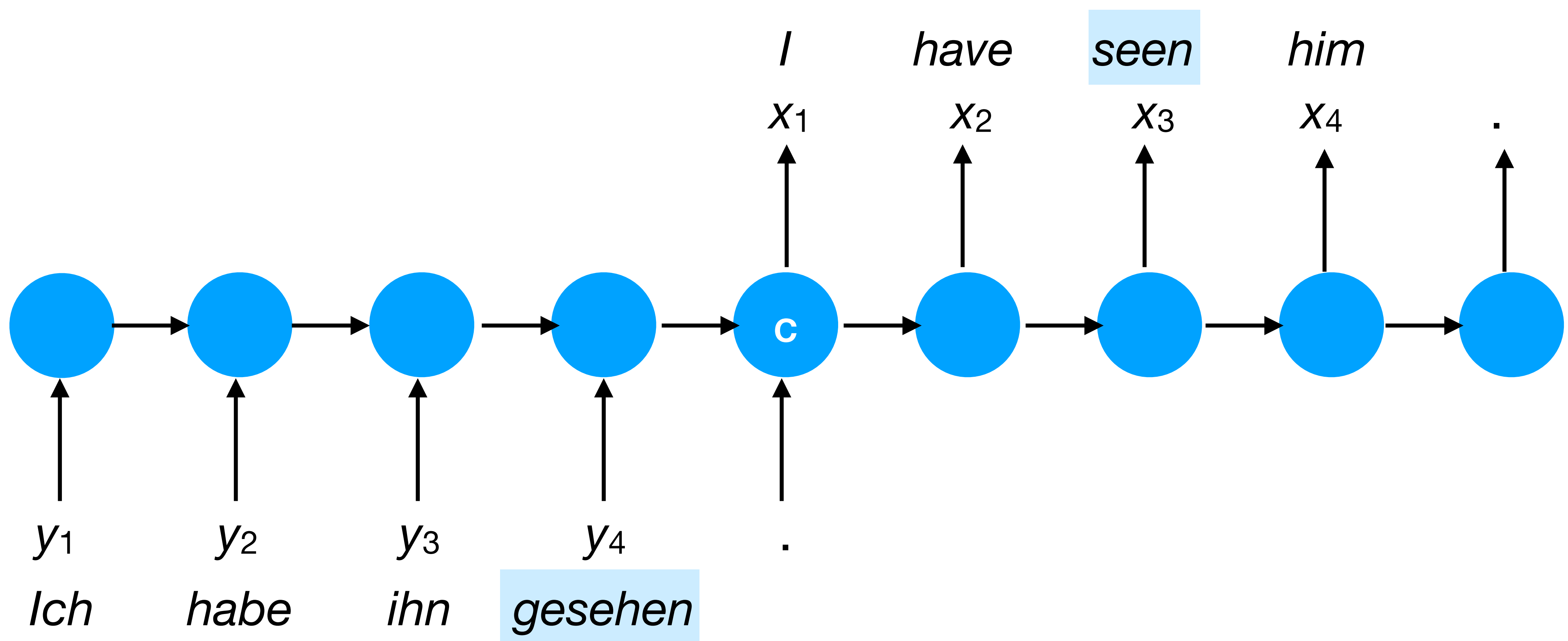
Encoder-Decoder models

- Intuitively, some parts of the input may be more relevant than others when producing a particular output symbol x_t.



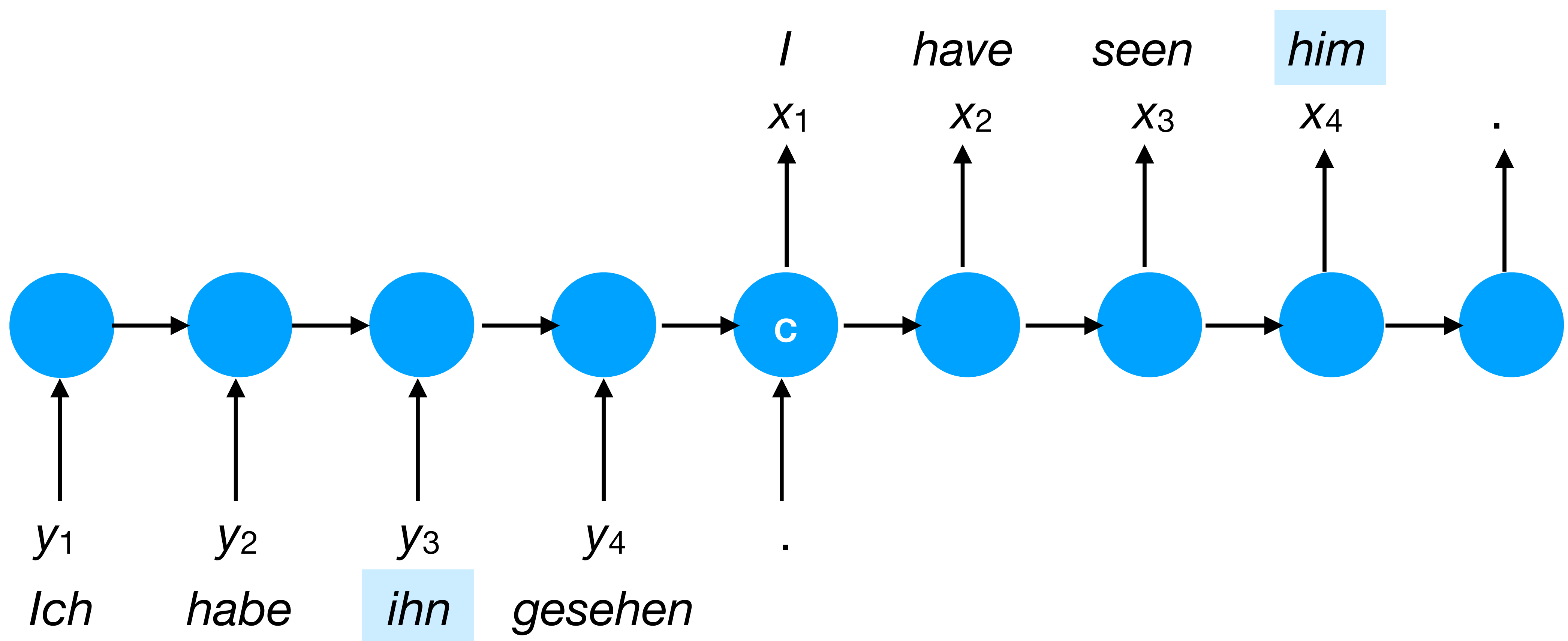
Encoder-Decoder models

- Intuitively, some parts of the input may be more relevant than others when producing a particular output symbol x_t.



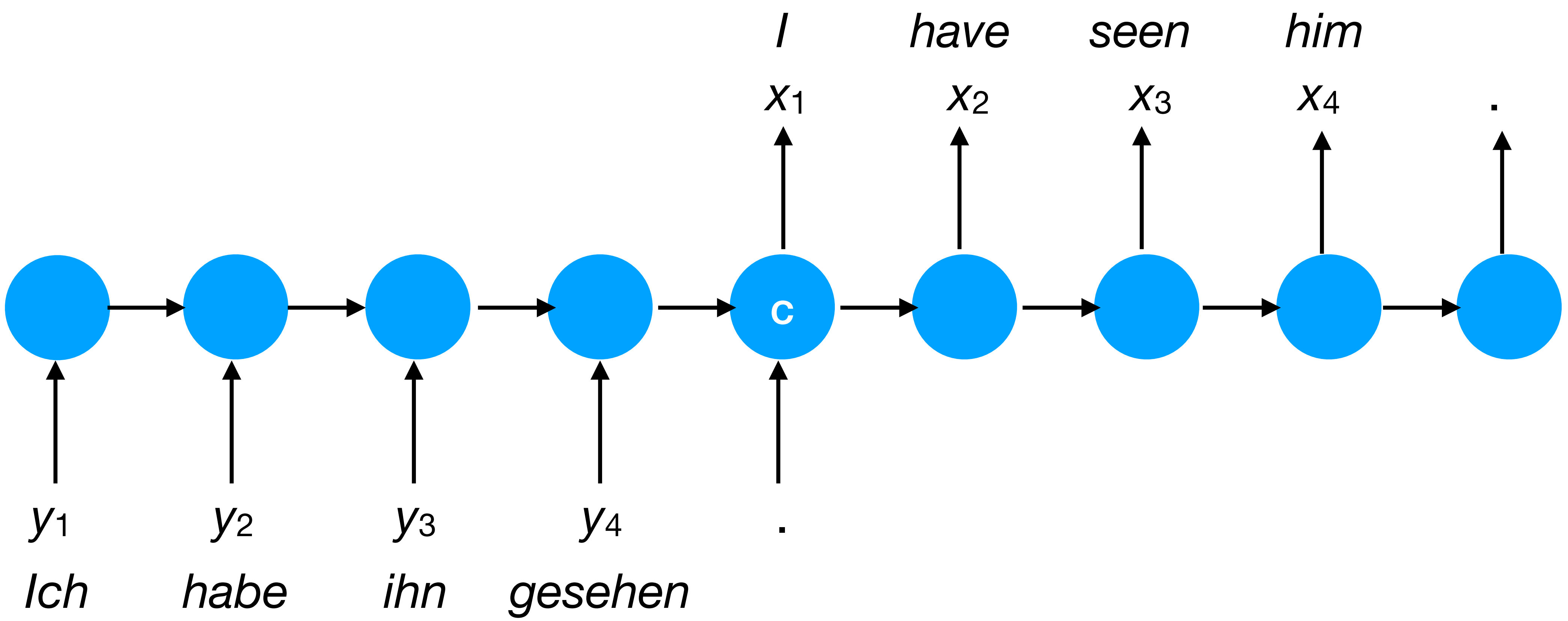
Encoder-Decoder models

- Intuitively, some parts of the input may be more relevant than others when producing a particular output symbol x_t.



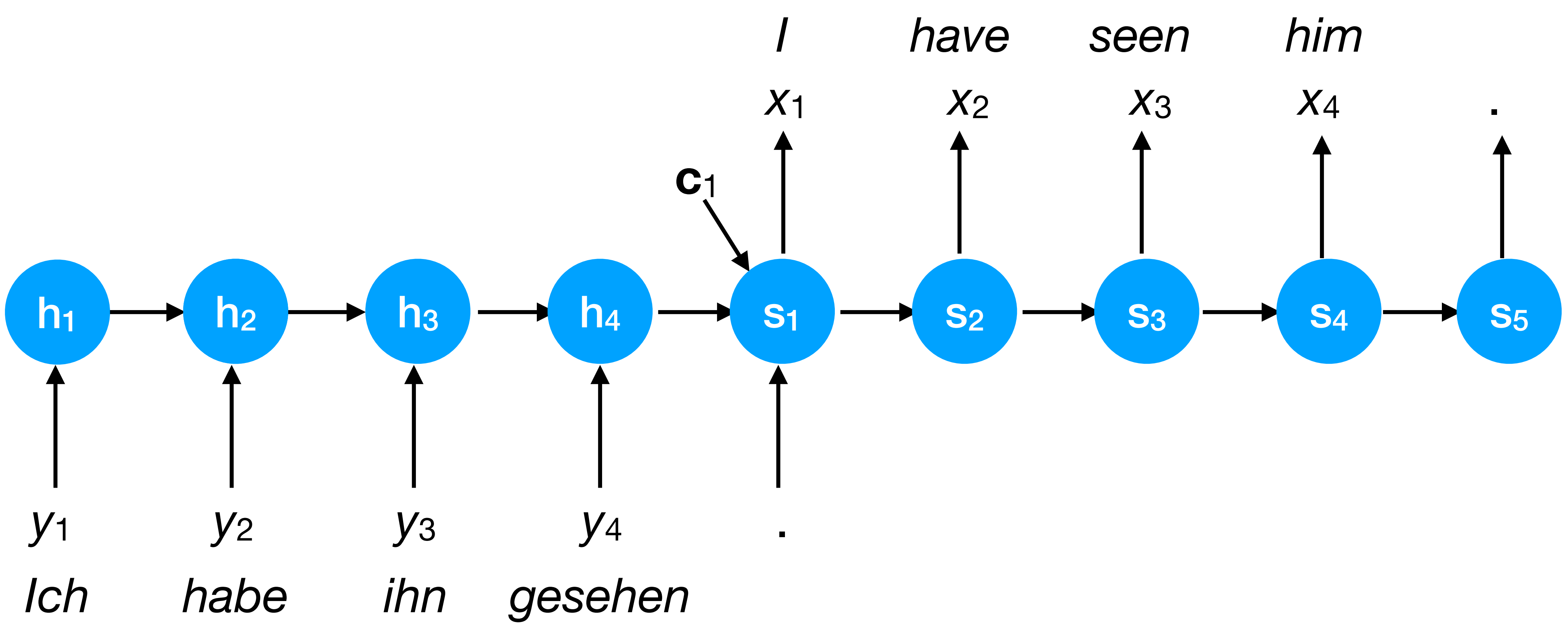
Encoder-Decoder models

- How can we focus the network's **attention** on the **most relevant inputs** when deciding each of the outputs?



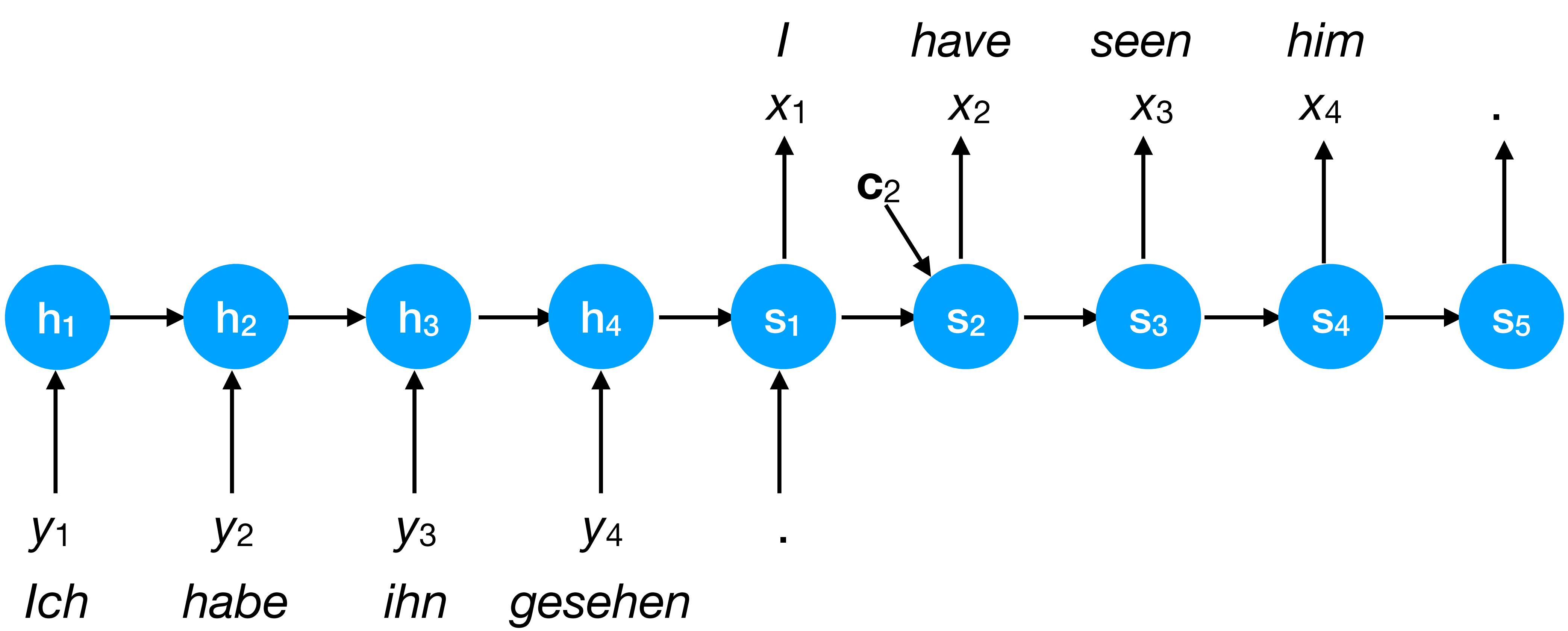
Varying context

- Instead of a fixed **c**, we compute a different $\mathbf{c}_t$ for each t.



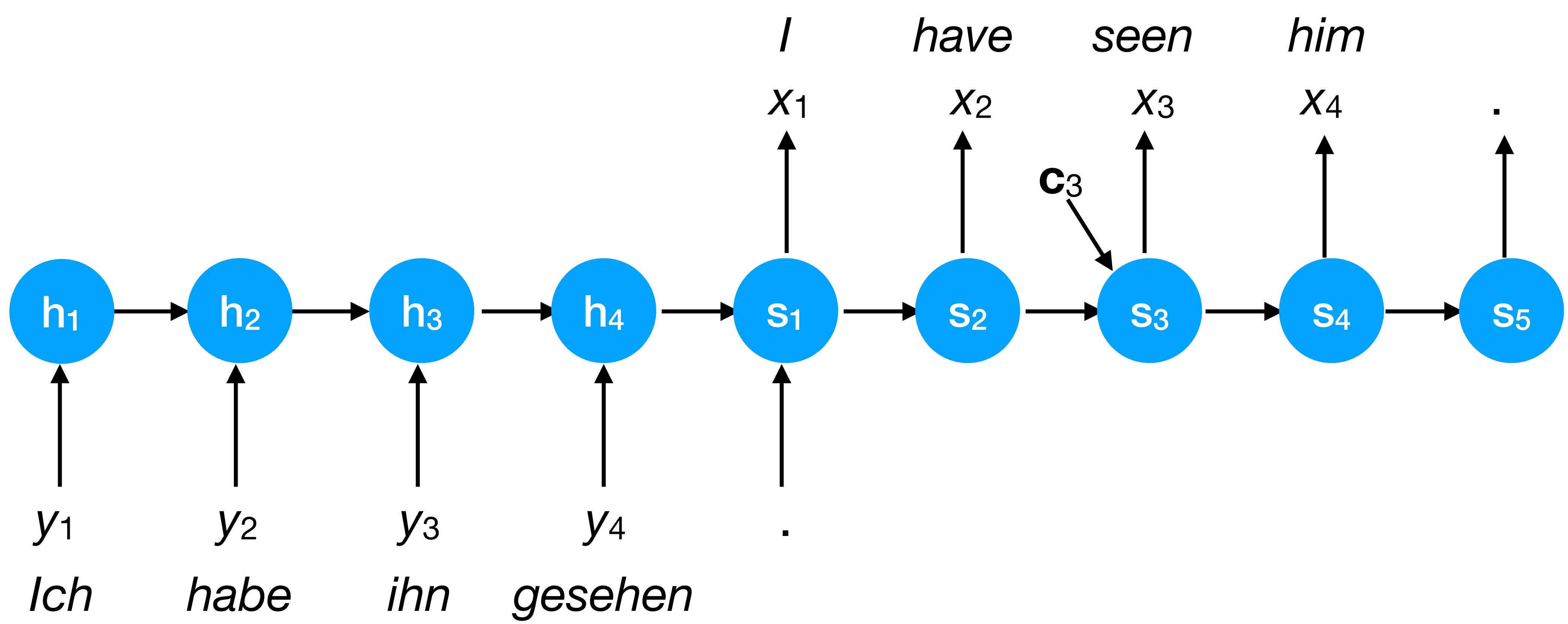
Varying context

- Instead of a fixed **c**, we compute a different $\mathbf{c}_t$ for each t.



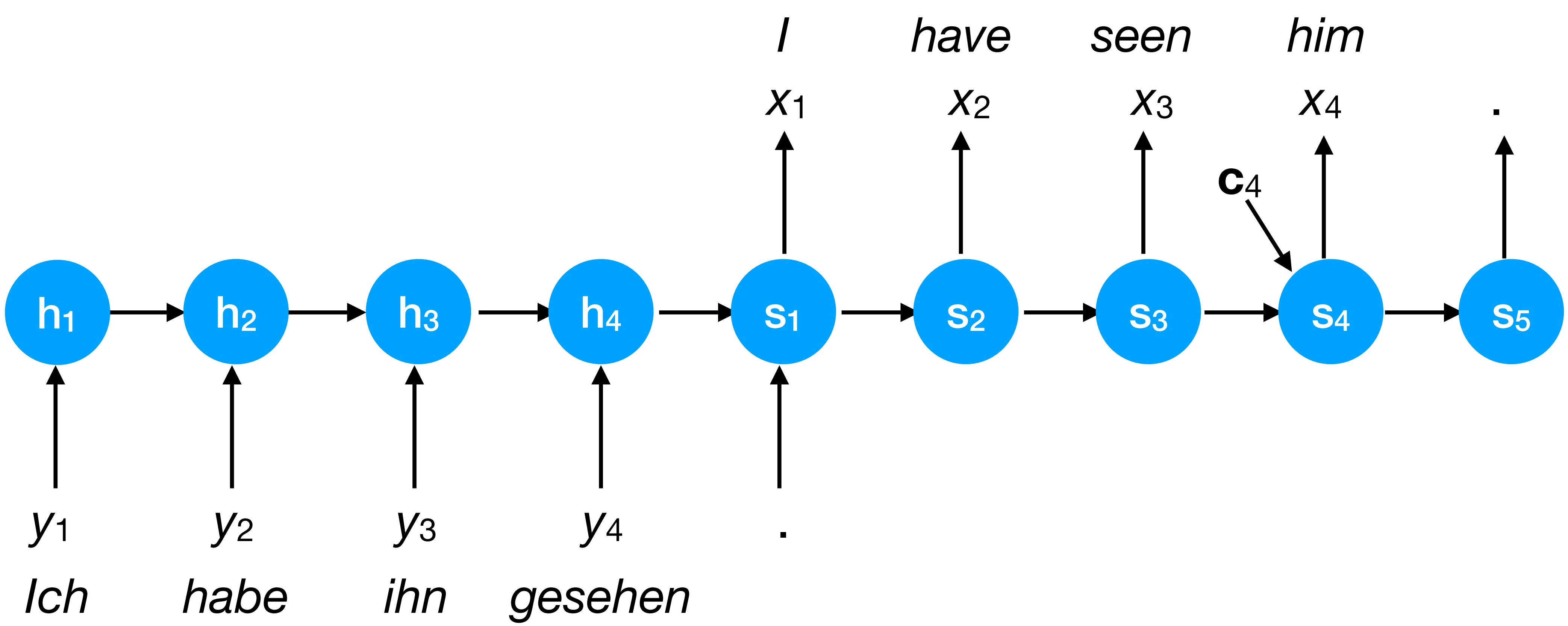
Varying context

- Instead of a fixed **c**, we compute a different $\mathbf{c}_t$ for each t.



Varying context

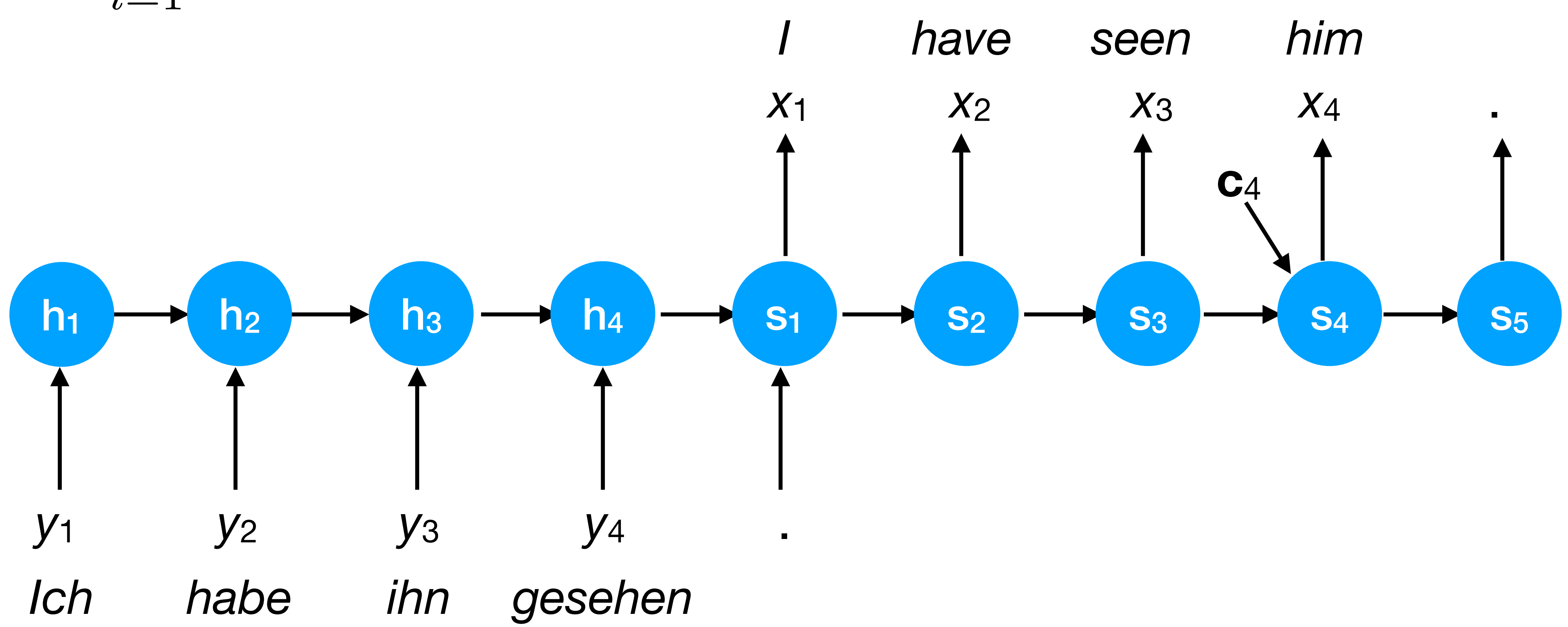
- Instead of a fixed **c**, we compute a different $\mathbf{c}_t$ for each t.



Attention weights

- Each $\mathbf{c}_t$ is a weighted sum (with attention weights $\boldsymbol{\alpha}_t$) of the hidden state sequence $\mathbf{h}_1, \ldots, \mathbf{h}_T$:

$$\mathbf{c}_t = \sum_{i=1}^{T} \alpha_{ti} \mathbf{h}_i$$

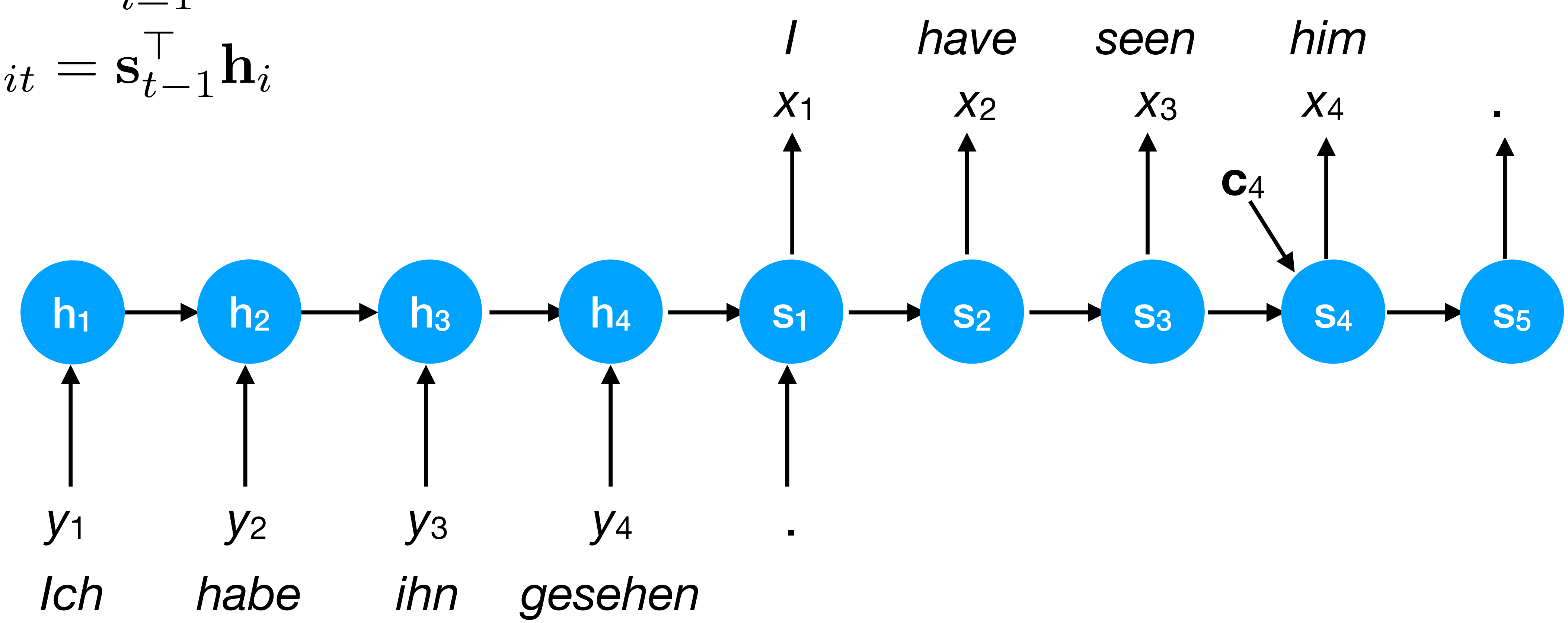


Attention weights

- The attention weights $\boldsymbol{\alpha}_t$ are computed based on multiplicative interactions between $\mathbf{s}_{t\text{-}1}$ and each $\mathbf{h}_i$.

$$\mathbf{c}_t = \sum_{i=1}^{T} \alpha_{ti}\mathbf{h}_i$$

$$\alpha_{it} = \mathbf{s}_{t-1}^{\top}\mathbf{h}_i$$



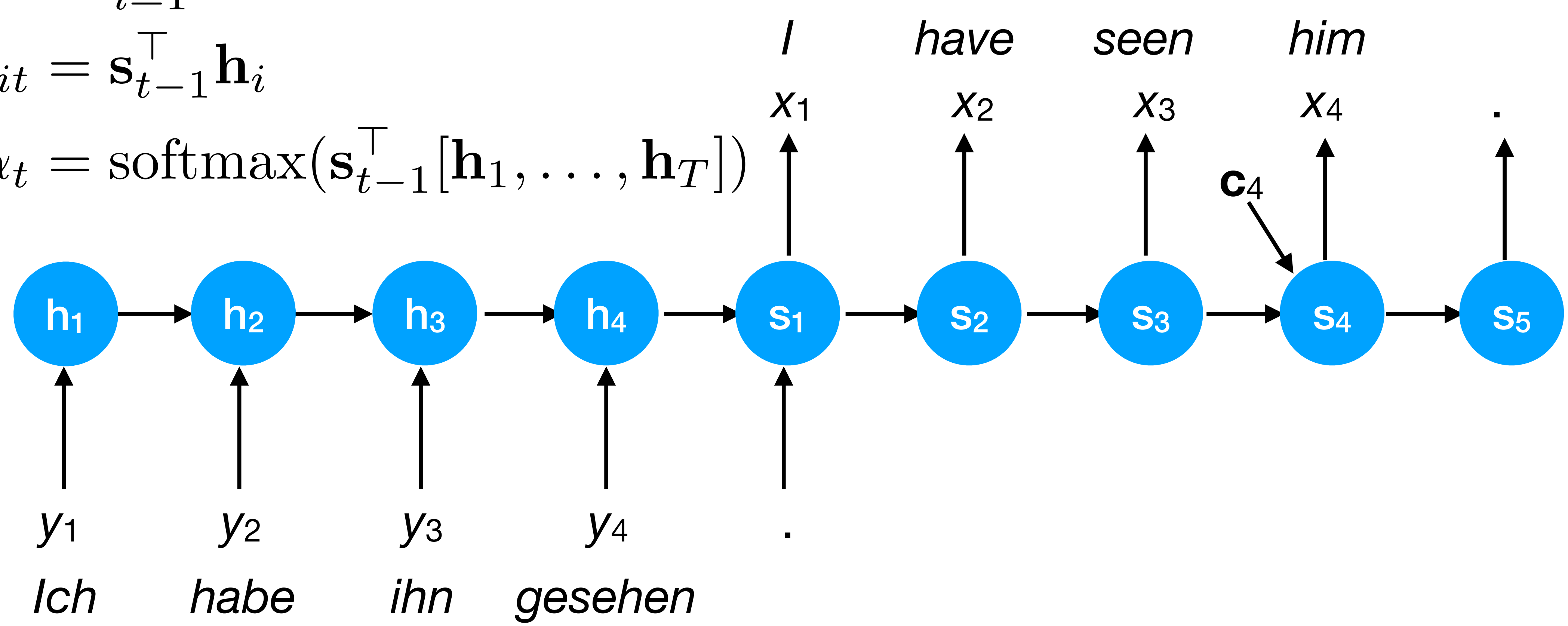
Attention weights

- With a softmax activation, they are forced to sum to 1.

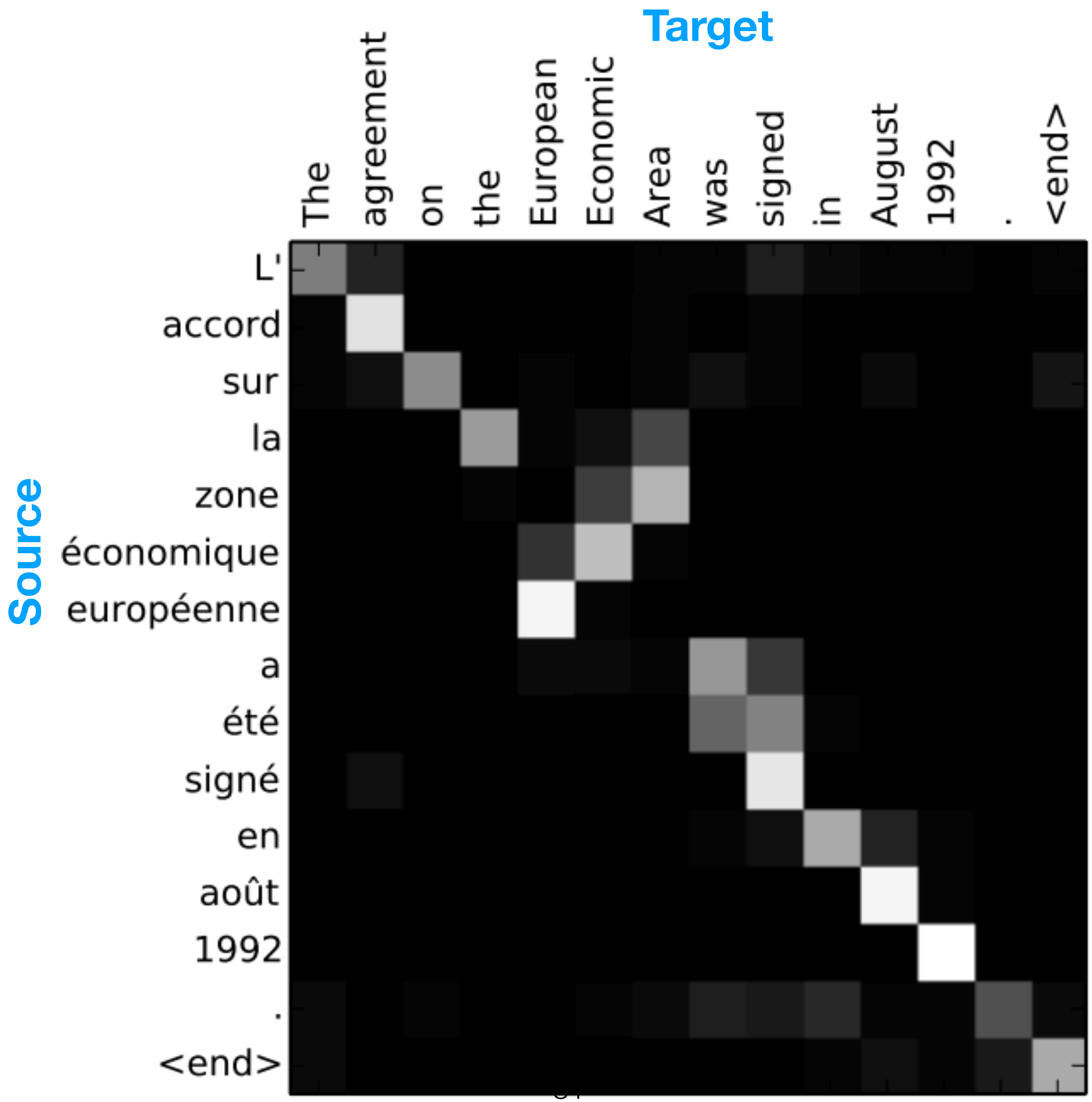
$$\mathbf{c}_t = \sum_{i=1}^{T} \alpha_{ti} \mathbf{h}_i$$

$$\alpha_{it} = \mathbf{s}_{t-1}^\top \mathbf{h}_i$$

$$\alpha_t = \text{softmax}(\mathbf{s}_{t-1}^\top [\mathbf{h}_1, \ldots, \mathbf{h}_T])$$



Example attention weights (Bahdanau et al. 2015)

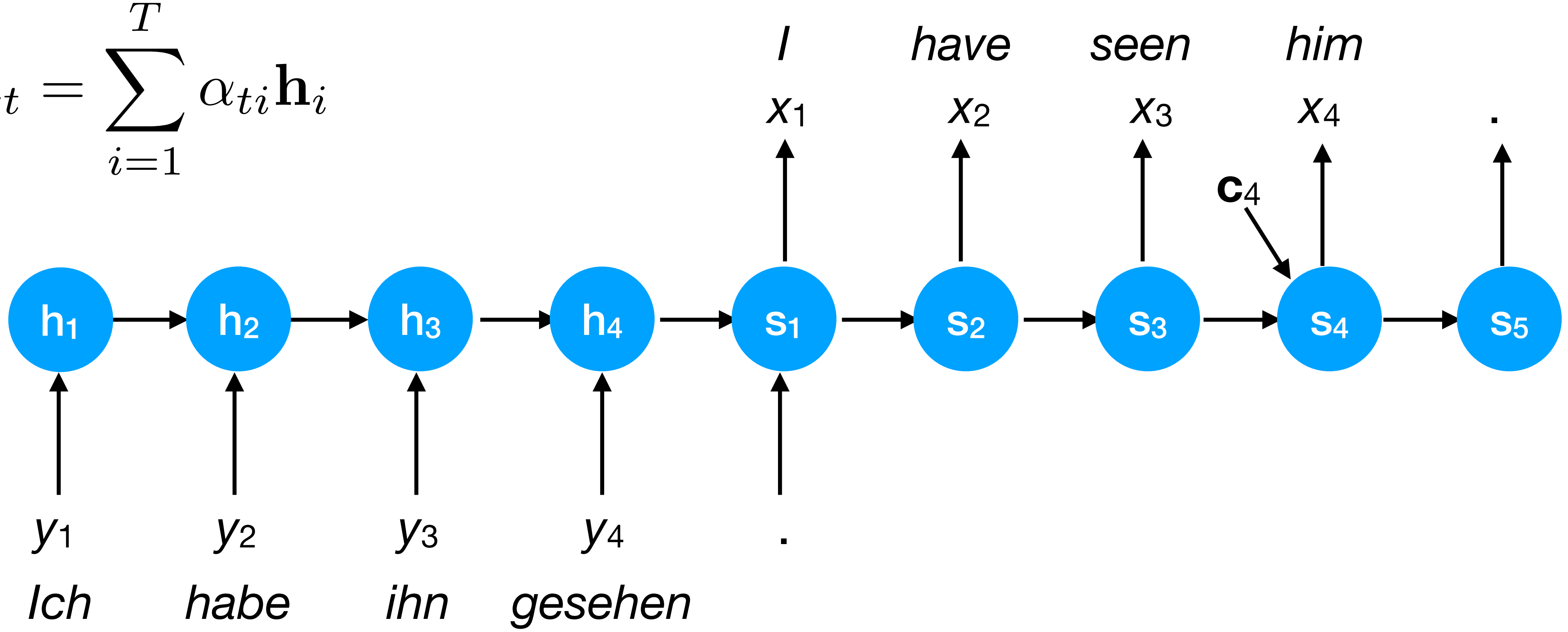


Attention weights

- With attention weights, we no longer compress the entire history into a single vector.

- Instead, we use $O(T)$ storage (for the $\{ \mathbf{h}_t \}$) to obtain better representational capacity.

$$\mathbf{c}_t = \sum_{i=1}^{T} \alpha_{ti} \mathbf{h}_i$$

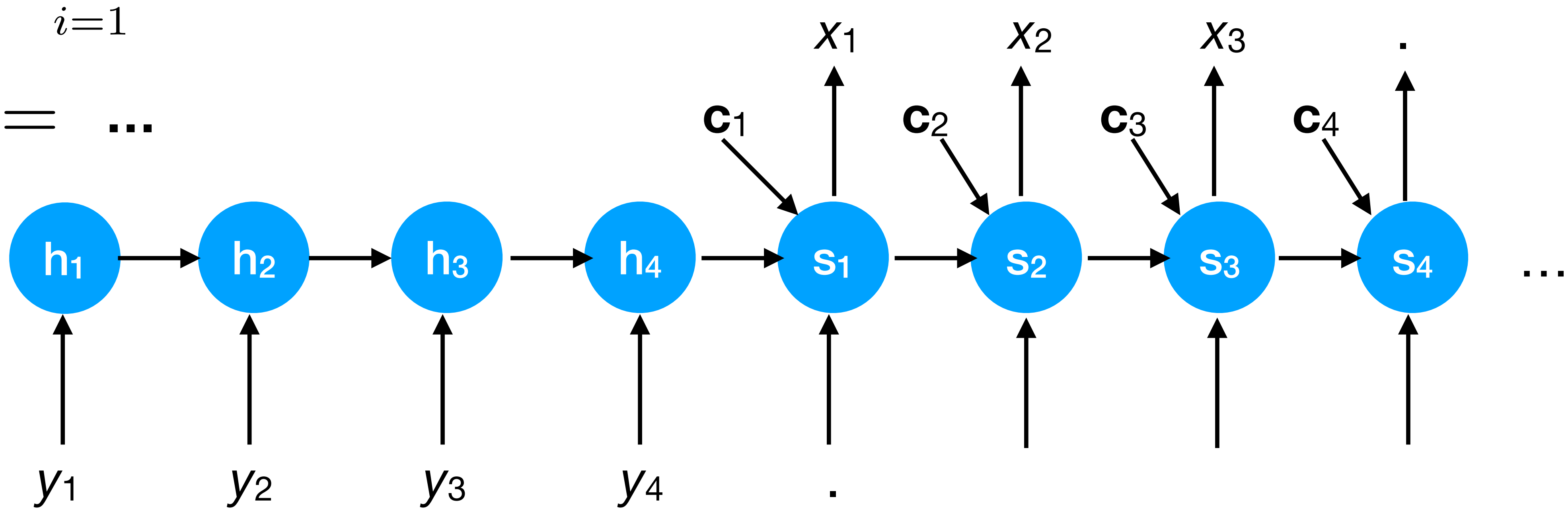


Attention: exercise

- Define matrices **S** and **H**, such that: `S[t,:]` = $\mathbf{s}_{t-1}^\top$, and `H[i,:]` = $\mathbf{h}_i^\top$.
- Suppose we ignore the temporal dependencies* and define `C[:,t]` = $\mathbf{c}_t^\top$.
- How can we compute **C** in one-fell-swoop using softmax() and matrix multiplication so that:

$$\mathbf{c}_t = \sum_{i=1}^{T} \alpha_{ti} \mathbf{h}_i \qquad \alpha_t = \mathrm{softmax}(\mathbf{s}_{t-1}^\top [\mathbf{h}_1, \ldots, \mathbf{h}_T])$$

$$\mathbf{C} = \ldots$$



* In a real RNN, each s_t depends on s_{t-1}; hence, while syntactically correct, this formula ignores the time dependence between states S. Hence, this algorithm to compute all contexts C could not actually be performed on real RNNs.

Queries, keys, & values

- Suppose you have the following question/query:
    - “How do I build a bridge from Unity Hall to the Eiffel Tower?”

Queries, keys, & values

- Suppose you have the following question/query:
  - “How do I build a bridge from Unity Hall to the Eiffel Tower?”
- We could consult different sources for advice.

Mechanical Engineer	**Biologist**	**Mathematician**	**Computer Scientist**

Queries, keys, & values

- Suppose you have the following question/query:
  - “How do I build a bridge from Unity Hall to the Eiffel Tower?”
- We could consult different sources for advice.

Mechanical Engineer	**Biologist**	**Mathematician**	**Computer Scientist**
“Construct a steel lattice such that…”	“Synthesize a DNA sequence with the base-pairs…”	“Consider the set *B* of all possible bridges…”	“While not built, add one stone…”

Queries, keys, & values

- How much should we trust each piece of advice?

Mechanical Engineer	**Biologist**	**Mathematician**	**Computer Scientist**
"Construct a steel lattice such that…"	"Synthesize a DNA sequence with the base-pairs…"	"Consider the set *B* of all possible bridges…"	"While not built, add one stone…"

Queries, keys, & values

- How much should we trust each piece of advice?
- We can weight each piece of advice **v** based on the similarity between our query **q** and the source **k**.

q	**“How do I build a bridge from Unity Hall to the Eiffel Tower?”**			
k	**Mechanical Engineer**	**Biologist**	**Mathematician**	**Computer Scientist**
v	“Construct a steel lattice such that…”	“Synthesize a DNA sequence with the base-pairs…”	“Consider the set *B* of all possible bridges…”	“While not built, add one stone…”

Queries, keys, & values

- How much should we trust each piece of advice?
- We can weight each piece of advice **v** based on the similarity between our query **q** and the source **k**.
- **q** is the **query**; **k** is the **key**; and **v** is the **value**.

q	**"How do I build a bridge from Unity Hall to the Eiffel Tower?"**			
k	**Mechanical Engineer**	**Biologist**	**Mathematician**	**Computer Scientist**
v	"Construct a steel lattice such that…"	"Synthesize a DNA sequence with the base-pairs…"	"Consider the set *B* of all possible bridges…"	"While not built, add one stone…"

Queries, keys, & values

- Suppose **q**, **k**, and **v** are all encoded as feature vectors of the same dimension.
- What is the simplest way to express the similarity between two vectors in the same feature space?

q	**"How do I build a bridge from Unity Hall to the Eiffel Tower?"**			
k	**Mechanical Engineer**	**Biologist**	**Mathematician**	**Computer Scientist**
v	"Construct a steel lattice such that…"	"Synthesize a DNA sequence with the base-pairs…"	"Consider the set *B* of all possible bridges…"	"While not built, add one stone…"

Queries, keys, & values

- Suppose **q**, **k**, and **v** are all encoded as feature vectors of the same dimension.
- What is the simplest way to express the similarity between two vectors in the same feature space?

Inner product

q	**"How do I build a bridge from Unity Hall to the Eiffel Tower?"**			
k	**Mechanical Engineer**	**Biologist**	**Mathematician**	**Computer Scientist**
v	"Construct a steel lattice such that…"	"Synthesize a DNA sequence with the base-pairs…"	"Consider the set *B* of all possible bridges…"	"While not built, add one stone…"

Attention: queries, keys, and values

- For multiple queries, keys, and values, we compute the "answers" to our queries as:

$$\mathbf{C} = \underbrace{\mathrm{softmax}\left(\mathbf{Q}\mathbf{K}^{\top}\right)}_{\text{Attention weights}} \mathbf{V}$$

Attention: queries, keys, and values

- For multiple queries, keys, and values, we compute the "answers" to our queries as:

$$\mathbf{C} = \mathrm{softmax}\left(\mathbf{Q}\mathbf{K}^\top\right)\mathbf{V}$$

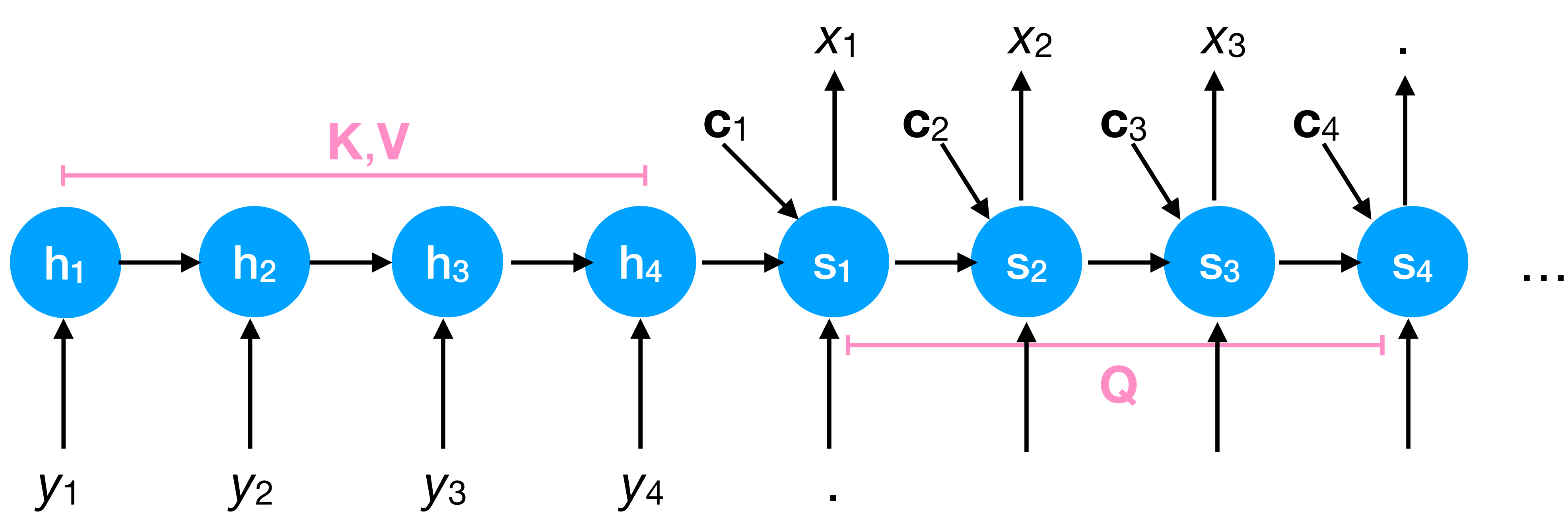
Conceptually:

- **Queries Q** come from the nodes (in the computational graph) that should *receive* the attention information.
- **Values V** come from the nodes that *provide* the attention information; they are multiplied by attention weights to produce the final attention information
- **Keys K** are *multiplied* with **Q** to yield attention weights.

Attention: queries, keys, and values

- In the encoder-decoder RNN attention model, the keys and values just happen to be the same:

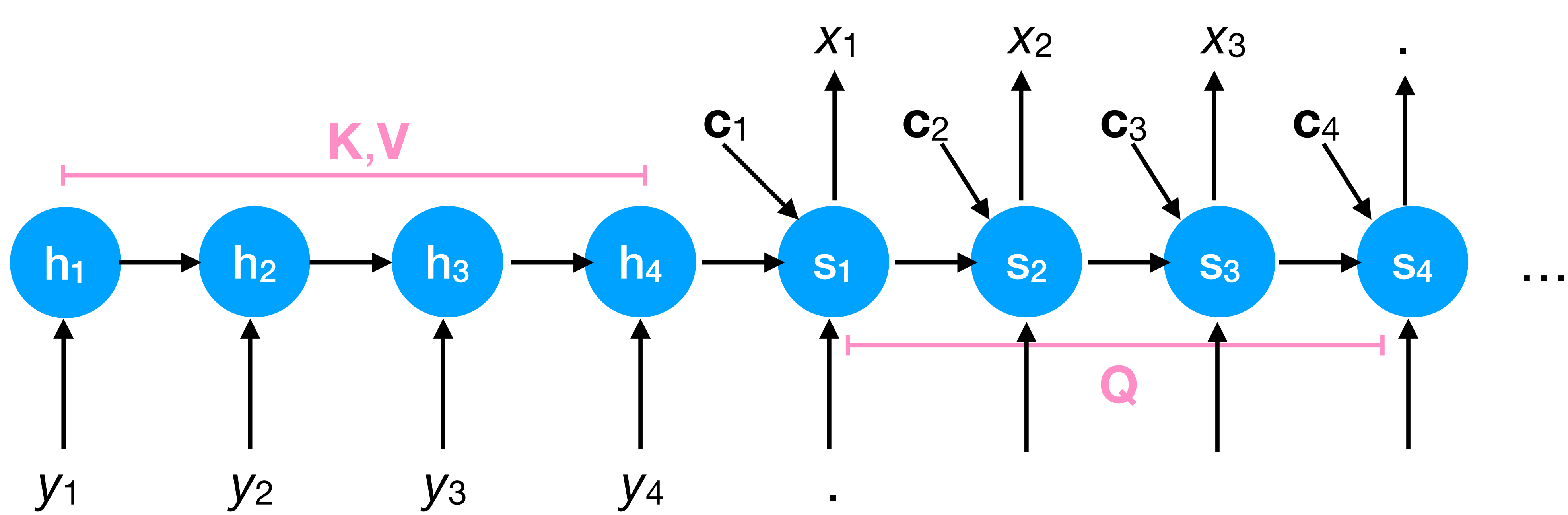
$$\mathbf{C} = \mathrm{softmax}\left(\mathbf{S}\mathbf{H}^\top\right)\mathbf{H}$$



Attention: queries, keys, and values

- However, instead of the "raw" encoder/decoder hidden states **S** and **H,**

$$\mathbf{C} = \mathrm{softmax}\left(\mathbf{S}\mathbf{H}^\top\right)\mathbf{H}$$



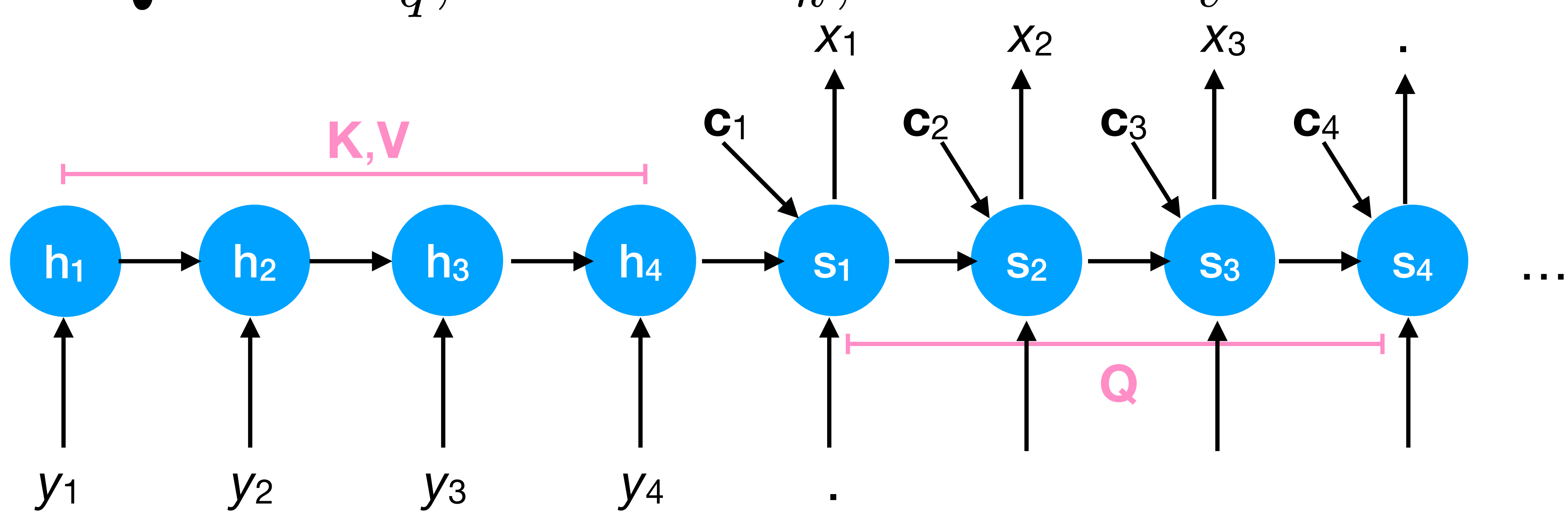
Attention: queries, keys, and values

- However, instead of the "raw" encoder/decoder hidden states **S** and **H,**

$$\mathbf{C} = \mathrm{softmax}\left(\mathbf{Q}\mathbf{K}^\top\right)\mathbf{V}$$

we can first project them with learned weight matrices $\mathbf{W}_q$, $\mathbf{W}_k$, and $\mathbf{W}_v$:

$$\mathbf{Q} = \mathbf{S}\mathbf{W}_q, \mathbf{K} = \mathbf{H}\mathbf{W}_k, \mathbf{V} = \mathbf{H}\mathbf{W}_v$$

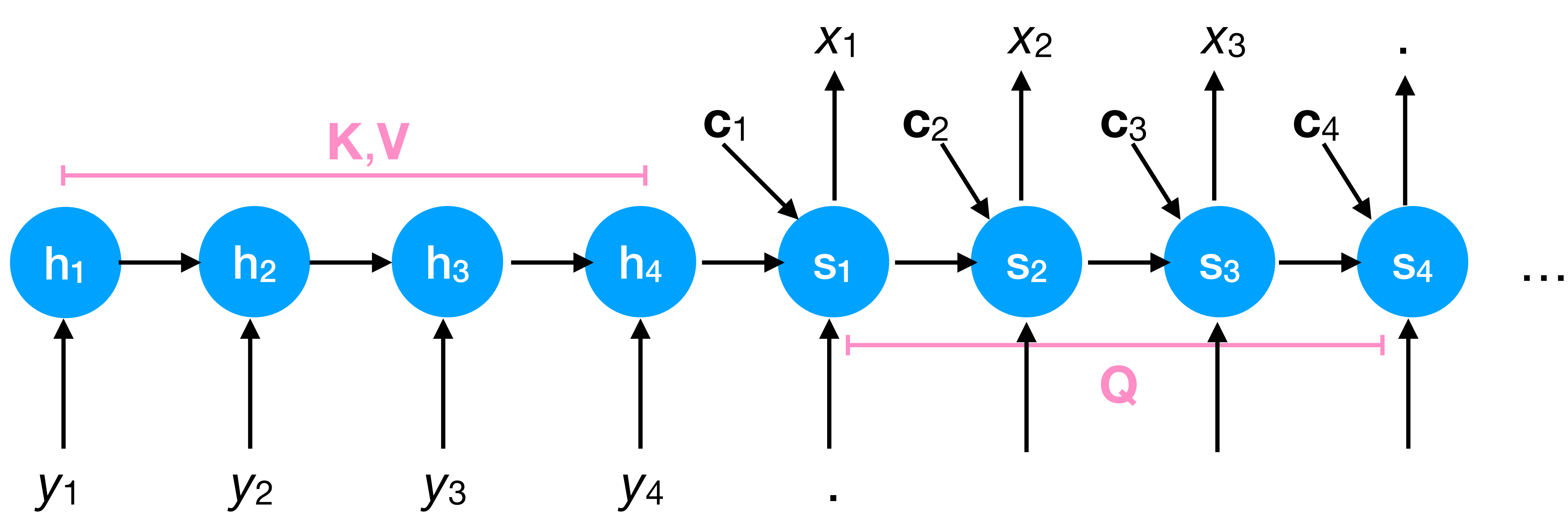


Attention: queries, keys, and values

- We typically normalize the preactivations by $\sqrt{d}$, where d is the key dimension.

$$\mathbf{C} = \mathrm{softmax}\left(\mathbf{Q}\mathbf{K}^\top / \sqrt{d}\right) \mathbf{V}$$

- This helps to prevent the weights from becoming too peaked with large d.



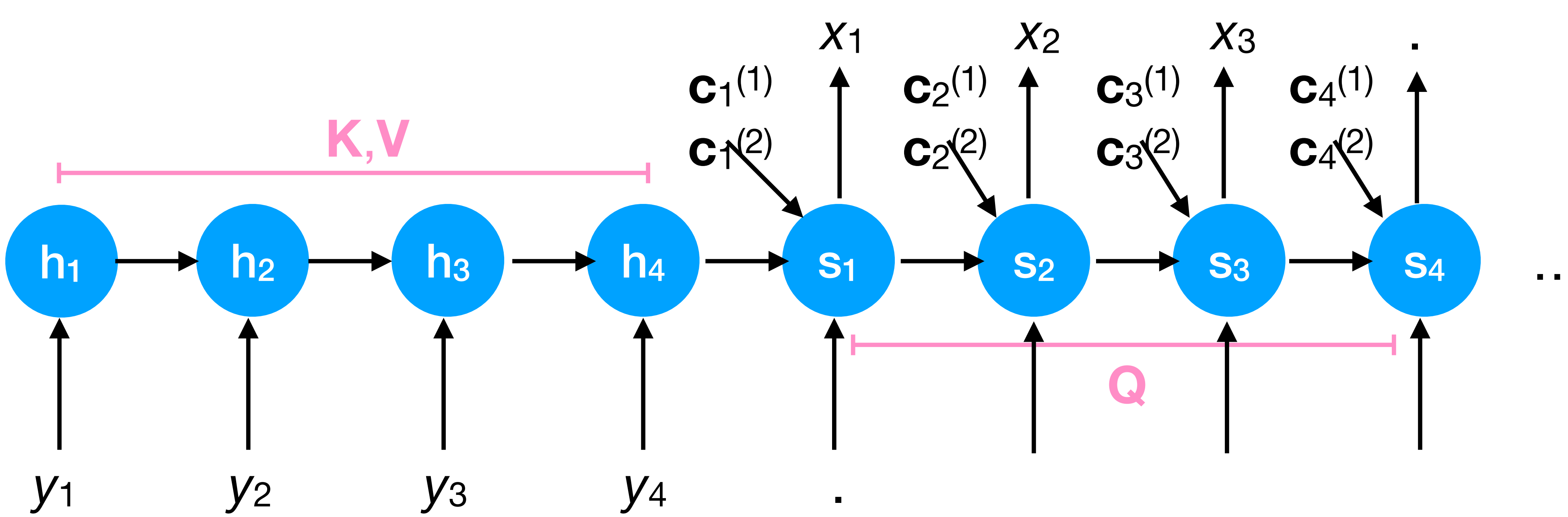
Multi-head attention

- Finally, we can generalize this further to allow *multiple* context vectors for each decoder timestep *t*:

$$\mathbf{C}^{(j)} = \mathrm{softmax}\left(\mathbf{Q}^{(j)}\mathbf{K}^{(j)\top}\right)\mathbf{V}^{(j)}$$

$$\mathbf{Q}^{(j)} = \mathbf{SW}_q^{(j)}, \mathbf{K}^{(j)} = \mathbf{HW}_k^{(j)}, \mathbf{V}^{(j)} = \mathbf{HW}_v^{(j)}$$

- This is called **multi-head attention.**

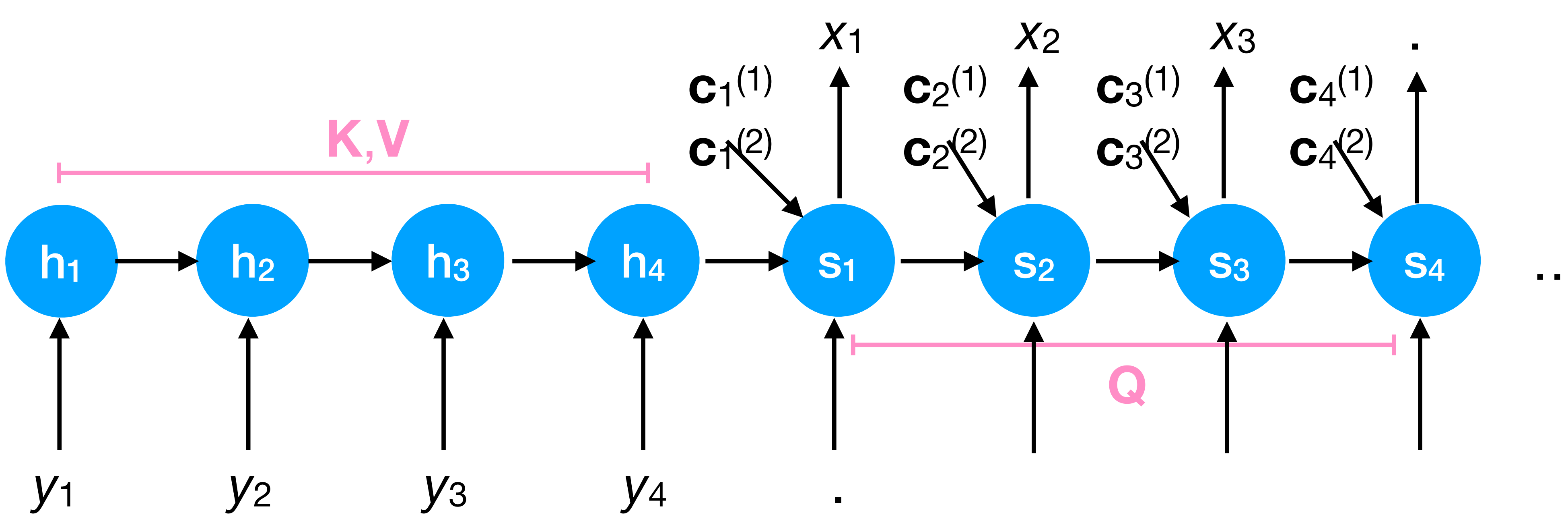


Multi-head attention

- Importantly, the shapes of these weight matrices do *not* depend on the length of the time series *T*.

$$\mathbf{C}^{(j)} = \text{softmax}\left(\mathbf{Q}^{(j)}\mathbf{K}^{(j)\top}\right)\mathbf{V}^{(j)}$$

$$\mathbf{Q}^{(j)} = \mathbf{S}\mathbf{W}_q^{(j)}, \mathbf{K}^{(j)} = \mathbf{H}\mathbf{W}_k^{(j)}, \mathbf{V}^{(j)} = \mathbf{H}\mathbf{W}_v^{(j)}$$

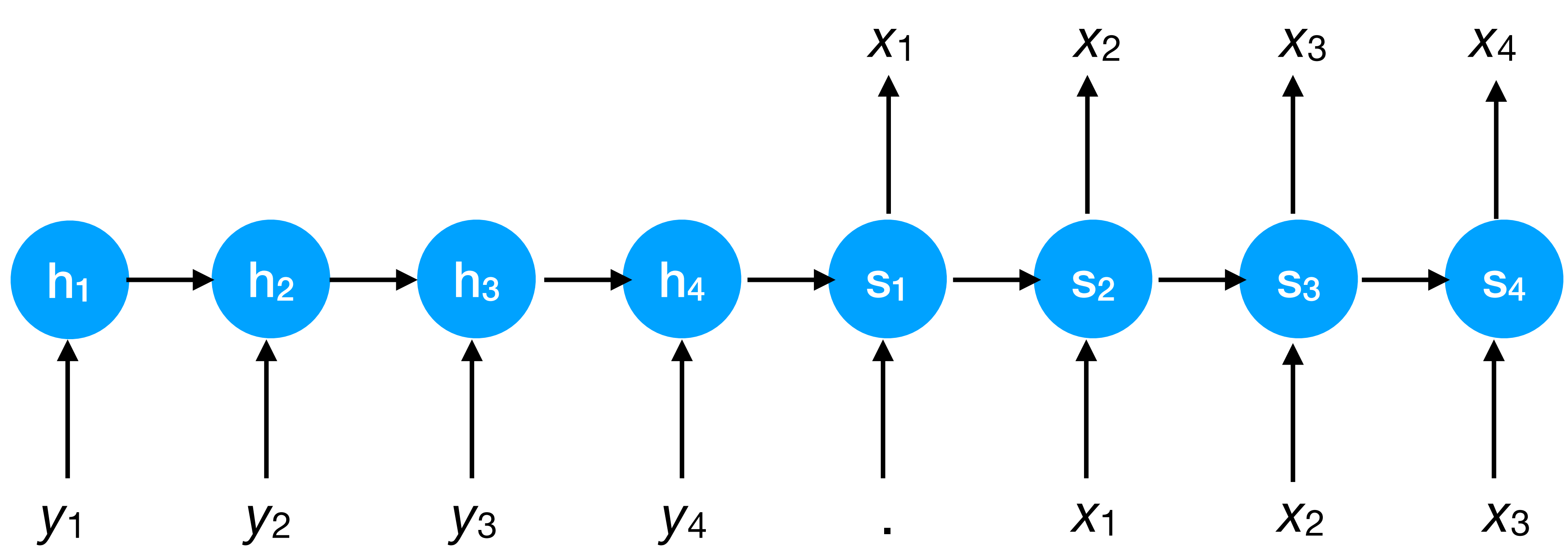


Limitations of encoder-decoder RNNs

Parallelizability

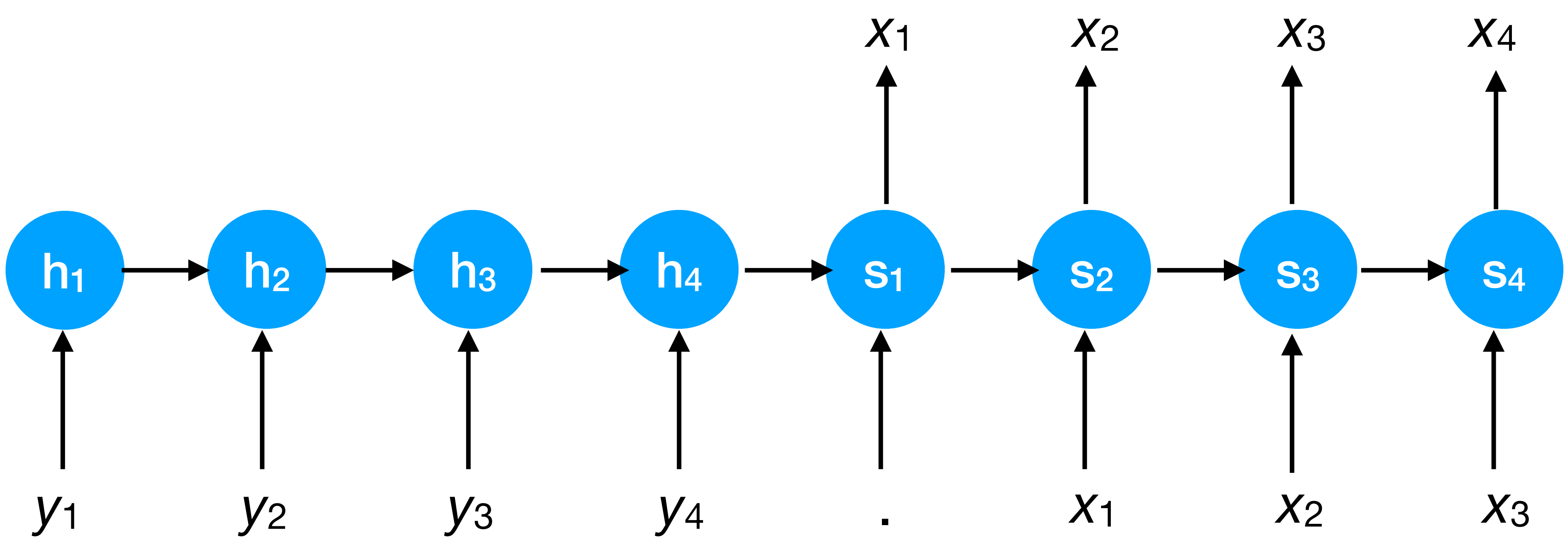
- Suppose we have *T* CPU/GPU cores available for training, and that (for simplicity) each sentence in the training set contains *T* words.

- How can we parallelize the computation of the *T* hidden states across the *T* cores?



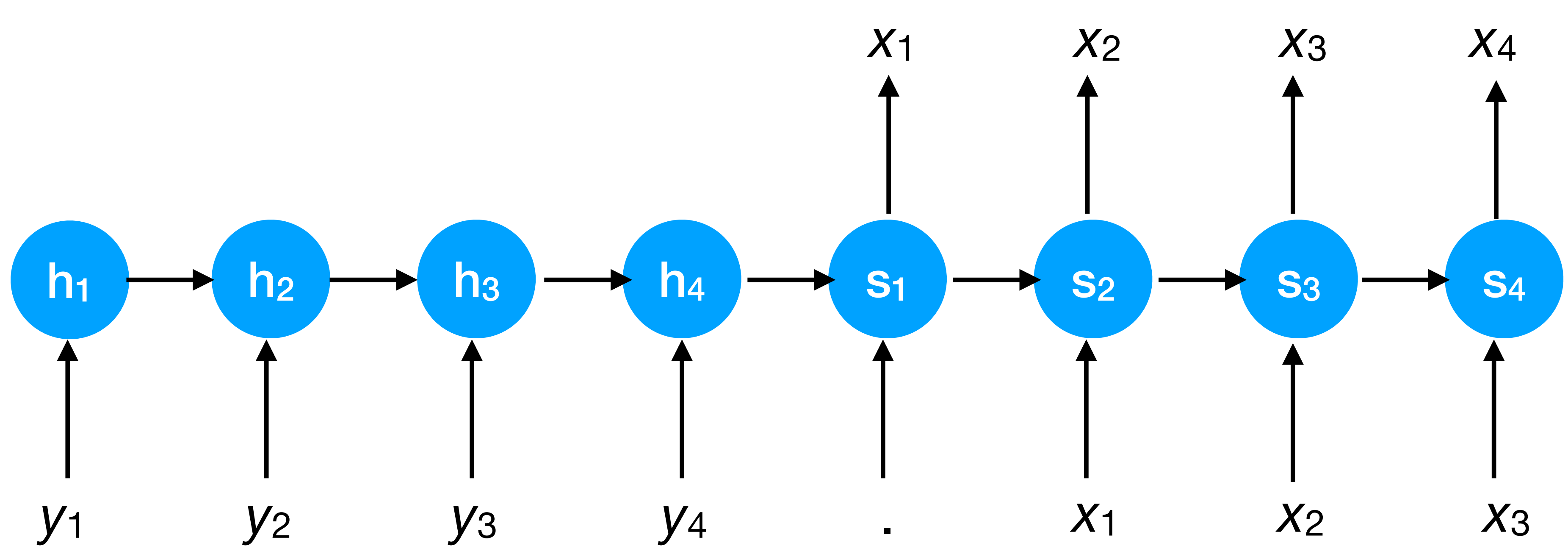
Parallelizability

- We can't. There is a performance bottleneck because the hidden state $\mathbf{h}_t$ can be computed only after computing the previous hidden state $\mathbf{h}_{t\text{-}1}$.



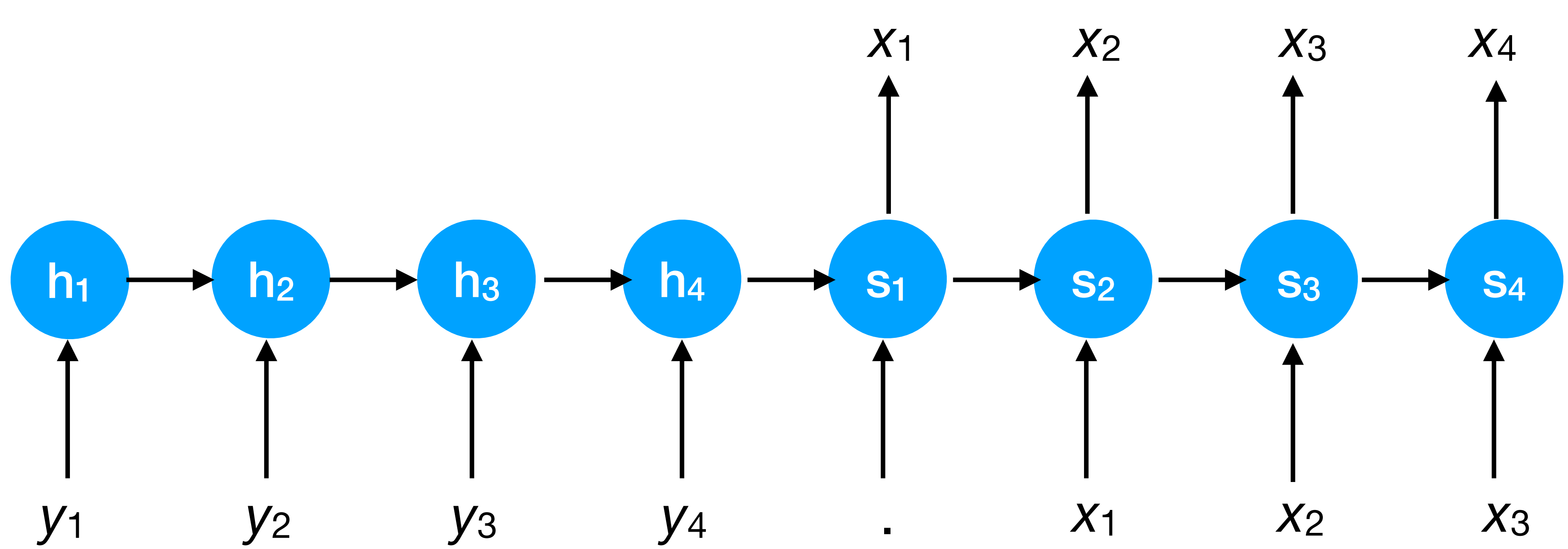
Parallelizability with teacher forcing

- What if we use teacher forcing to train this RNN, i.e., feed the ground-truth y_t instead of prediction $\hat{y}_t$?



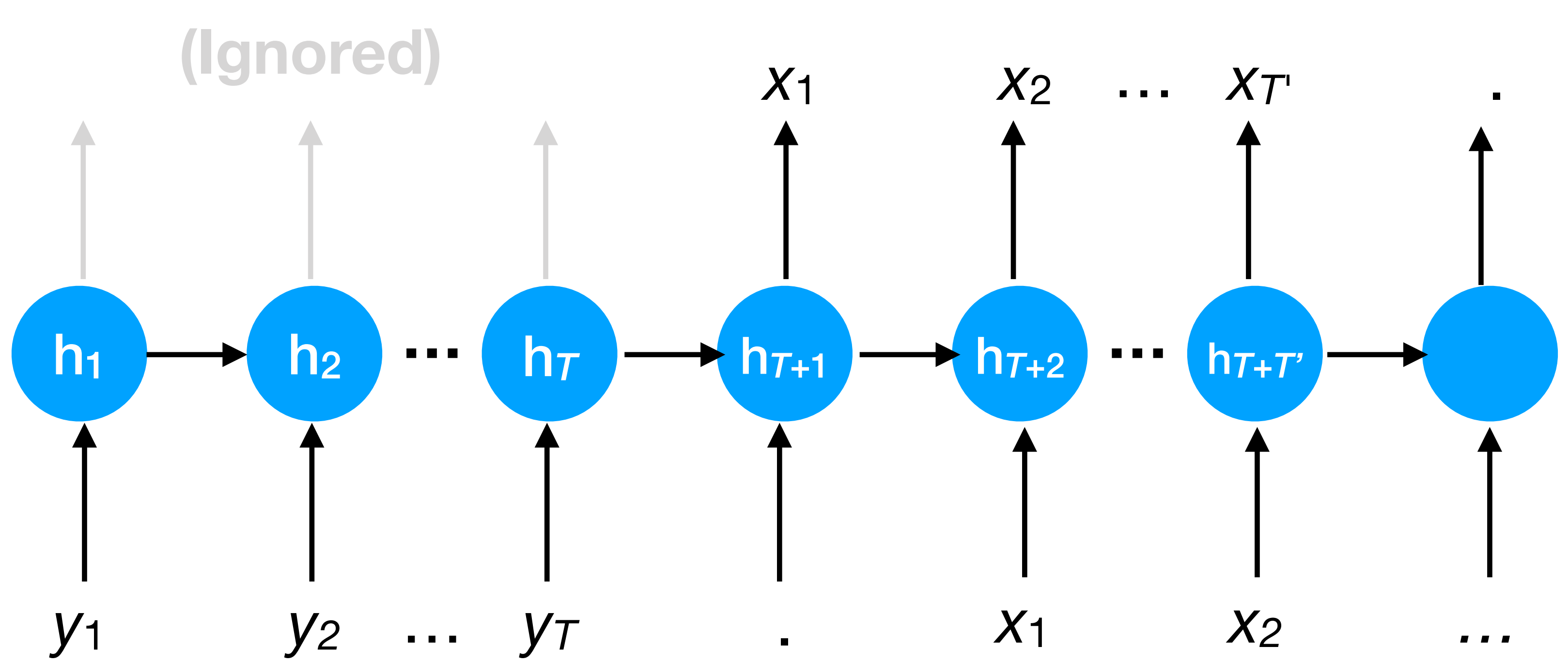
Parallelizability with teacher forcing

- Still can't — we still have the temporal dependency.



Limited memory

- $\mathbf{h}_t$ tries to "compress" the variable-length history into a fixed-length representation (i.e., $O(1)$ space).



Limitations of encoder-decoder RNNs

- Might there be a better encoder-decoder architecture for machine translation?
  - “Attention is All You Need”: Transformer (Vaswani et al. 2017).

Transformer

Transformer

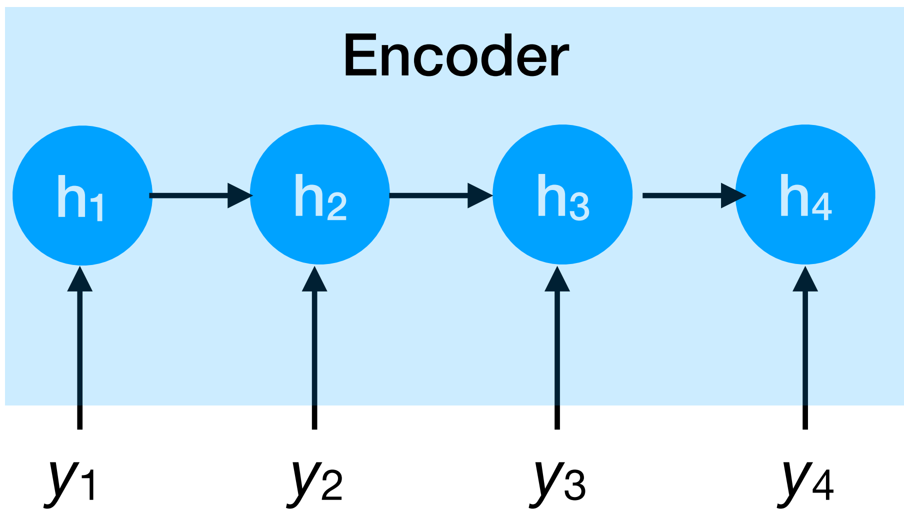
- Transformer (or one of its variants) is the state-of-the-art computational “backbone” of modern machine translation models (e.g., ChatGPT).
- It has also been adopted for other application domains (e.g., computer vision) and often outperforms CNNs.
- As suggested by the paper title, the key insight is that neural attention models are powerful.

Transformer

- Key elements of Transformer encoder-decoder model:
    1. Multi-head attention
    2. Point-wise fully-connected layers
    3. Positional encodings
    4. Masked inputs to the decoder

Transformer: encoder

- Instead of an RNN…

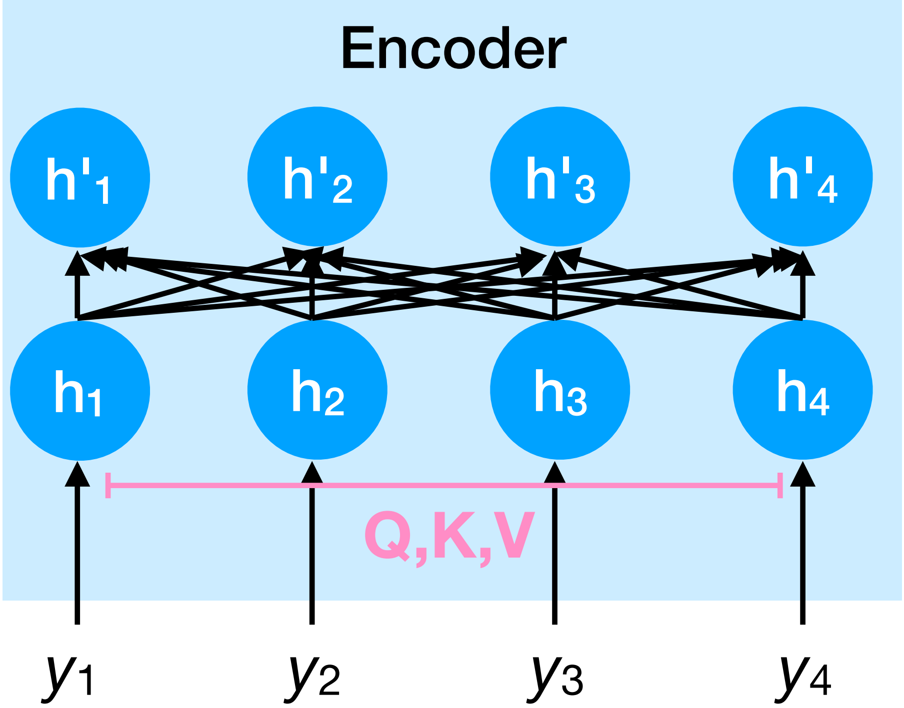


Transformer: encoder

- Instead of an RNN…we use (multi-head) **self-attention**:

$$\mathbf{H}' = \mathrm{softmax}\left(\mathbf{Q}\mathbf{K}^\top\right)\mathbf{V}$$

$$\mathbf{Q} = \mathbf{H}\mathbf{W}_q, \mathbf{K} = \mathbf{H}\mathbf{W}_k, \mathbf{V} = \mathbf{H}\mathbf{W}_v$$

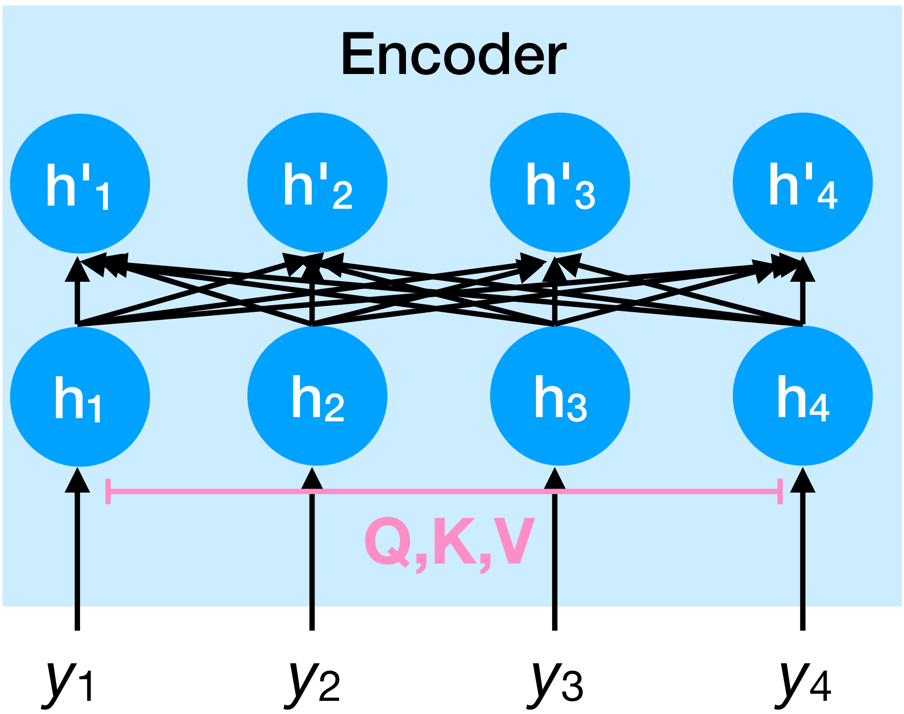


Transformer: encoder

- This is similar to a FC layer in the sense that each neuron h'_i depends on each neuron h_j.

$$\mathbf{H}' = \mathrm{softmax}\left(\mathbf{Q}\mathbf{K}^\top\right)\mathbf{V}$$

$$\mathbf{Q} = \mathbf{H}\mathbf{W}_q, \mathbf{K} = \mathbf{H}\mathbf{W}_k, \mathbf{V} = \mathbf{H}\mathbf{W}_v$$

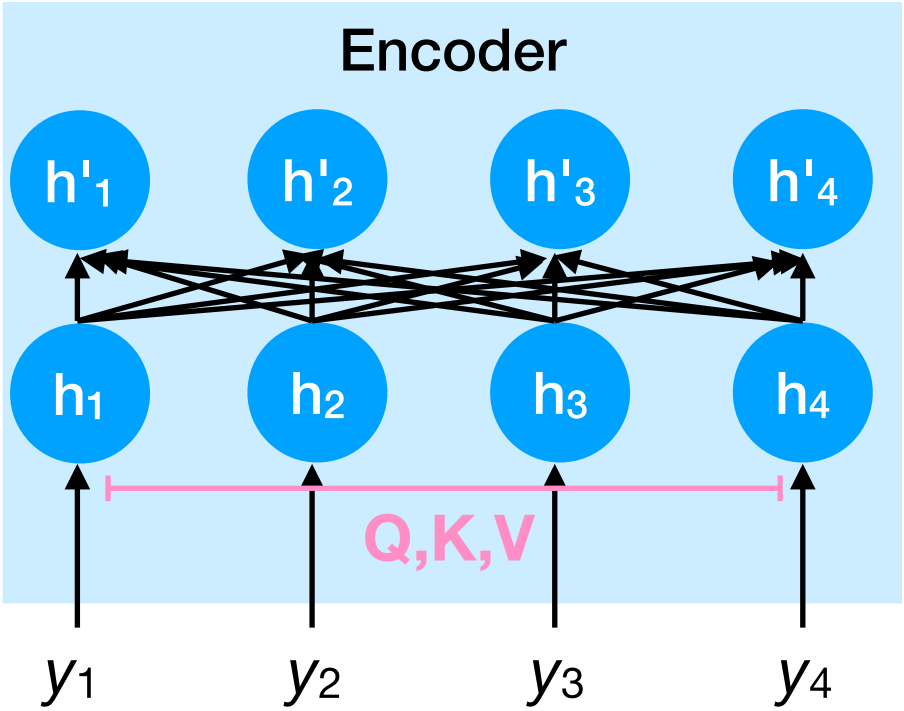


Transformer: encoder

- However, there is a crucial difference due to the **multiplicative interactions** between h_i and h_j.

$$\mathbf{H}' = \mathrm{softmax}\left(\mathbf{Q}\mathbf{K}^\top\right)\mathbf{V}$$

$$\mathbf{Q} = \mathbf{H}\mathbf{W}_q, \mathbf{K} = \mathbf{H}\mathbf{W}_k, \mathbf{V} = \mathbf{H}\mathbf{W}_v$$



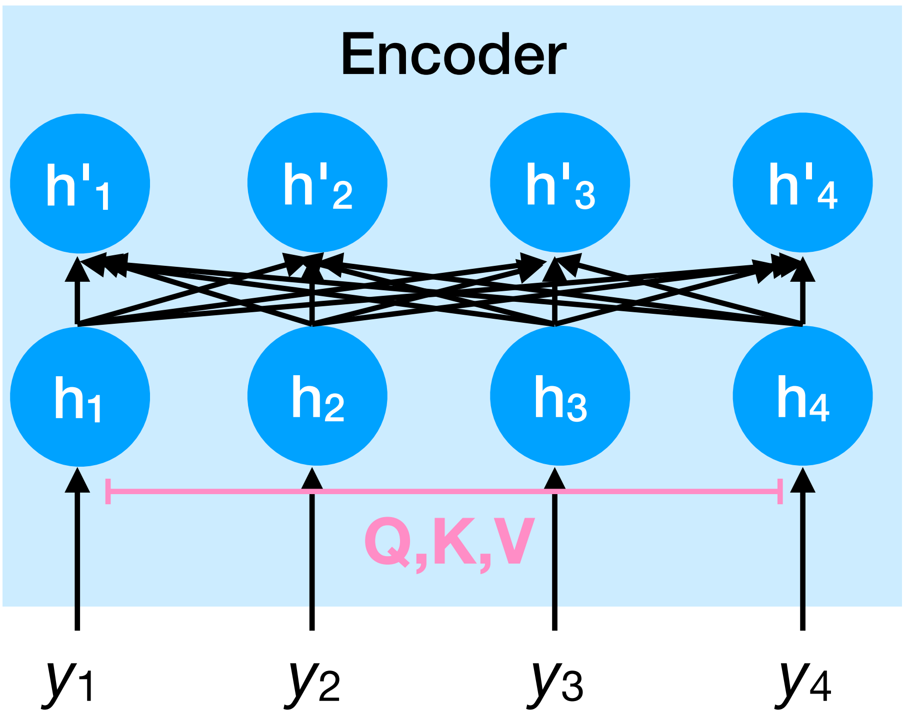
- We can conceptualize this in two alternative ways:
  1. The weights $\mathbf{W}_q$, $\mathbf{W}_k$, $\mathbf{W}_v$ are static but modulate the **multiplicative interactions** between h_i and h_j.
  2. The (attention) weights between h_i' and h_j are determined **dynamically**.

Transformer: encoder

- Conveniently, the number of attention parameters does *not* depend on the input length T (similarly to an RNN).

$$\mathbf{H}' = \mathrm{softmax}\left(\mathbf{Q}\mathbf{K}^\top\right)\mathbf{V}$$

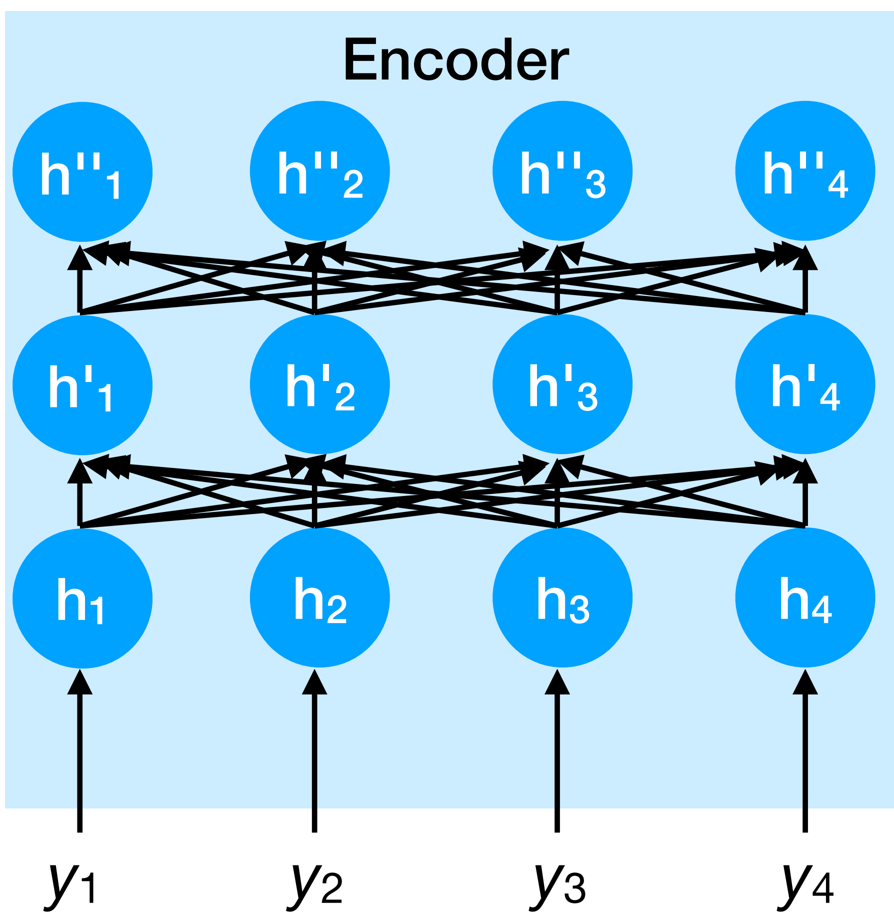
$$\mathbf{Q} = \mathbf{H}\mathbf{W}_q, \mathbf{K} = \mathbf{H}\mathbf{W}_k, \mathbf{V} = \mathbf{H}\mathbf{W}_v$$



Transformer: encoder

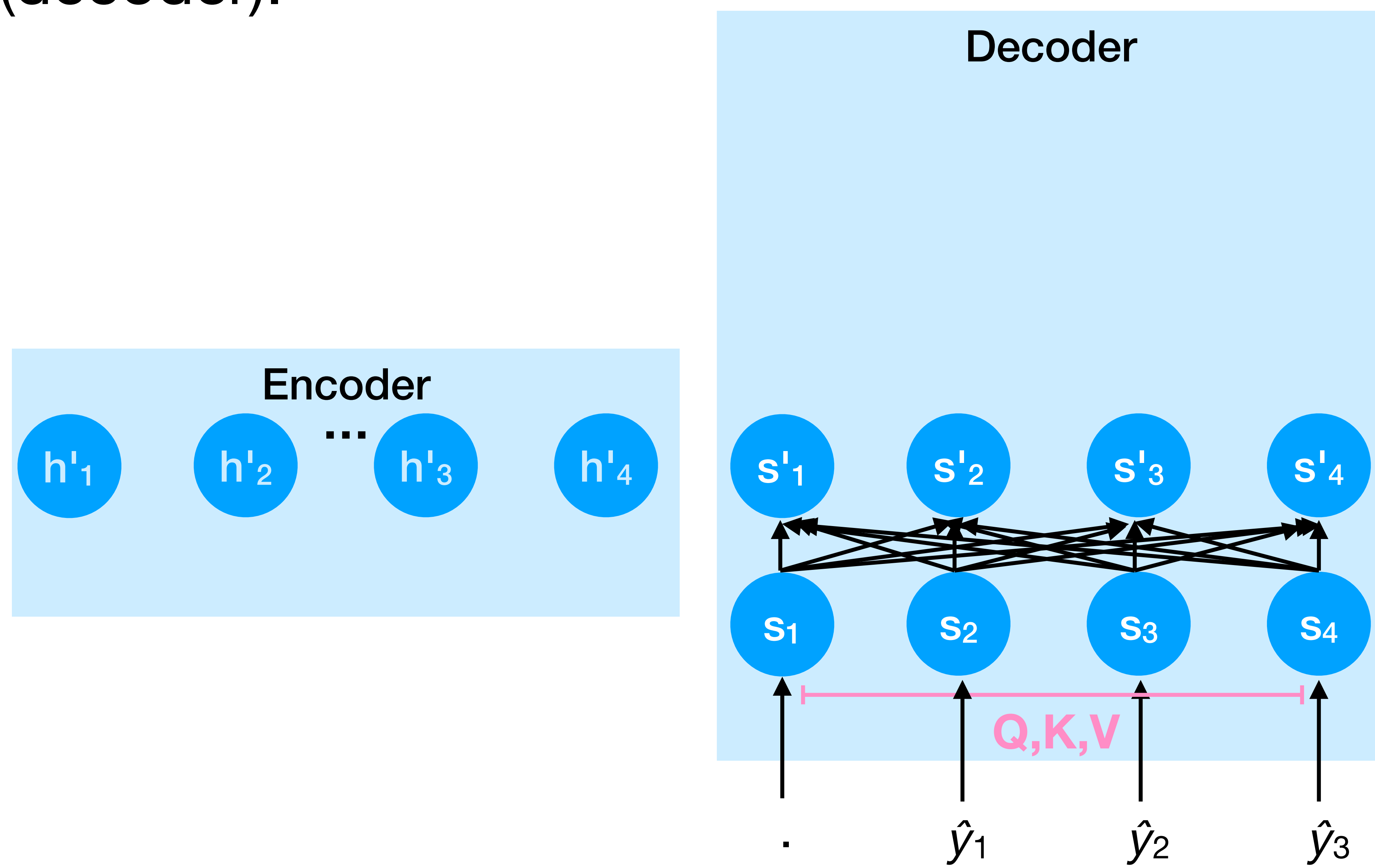
- We can apply this within multiple “transformer layers” of the encoder.

- Intuitively, this enables the encoder to compute successively more abstract representations of the entire input.



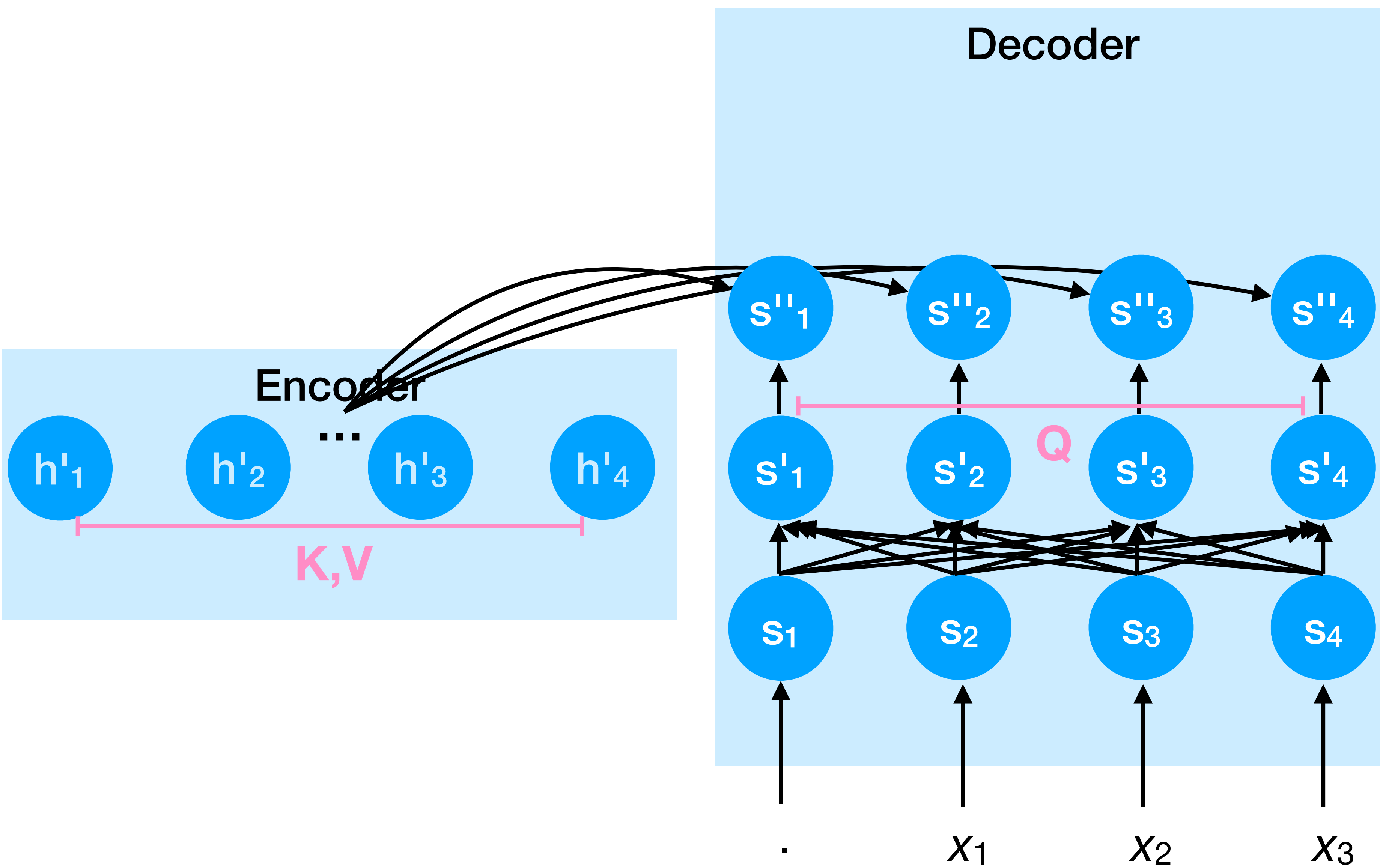
Transformer: decoder

- The decoder also uses multi-head **self-attention** to extract information across the output time series (decoder).



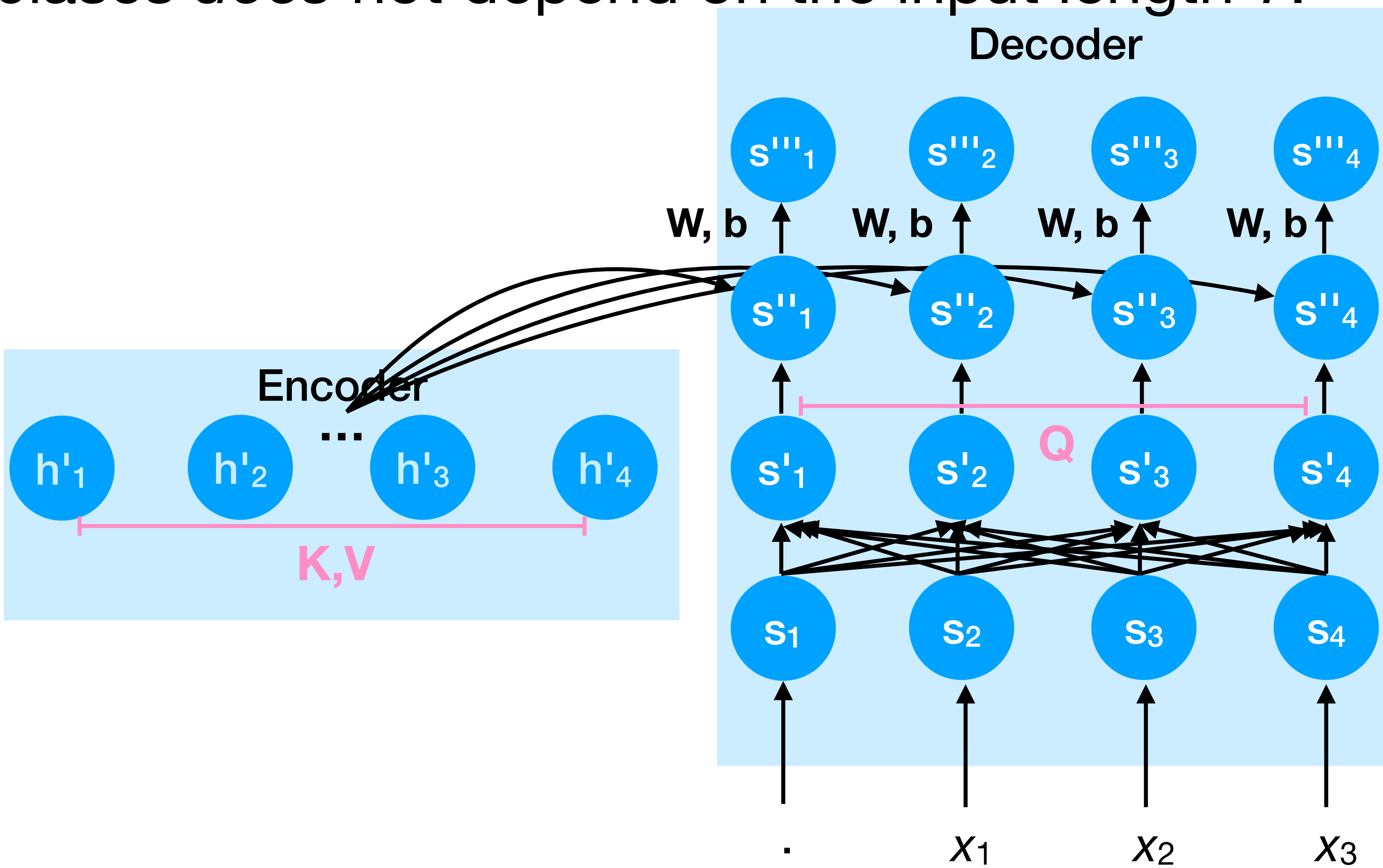
Transformer: decoder

- It also uses multi-head **cross-attention** to extract information from the input time series (encoder).



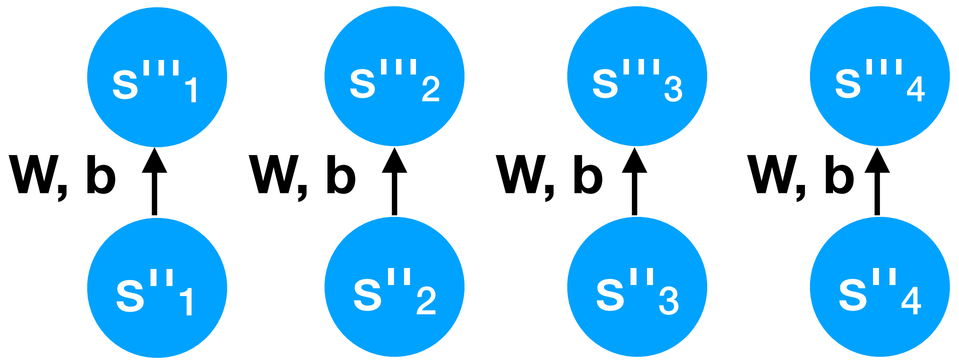
Transformer: decoder

- Next, it uses a **point-wise fully-connected** layer to transform each state vector. The number of weights/biases *does* not depend on the input length *T*.



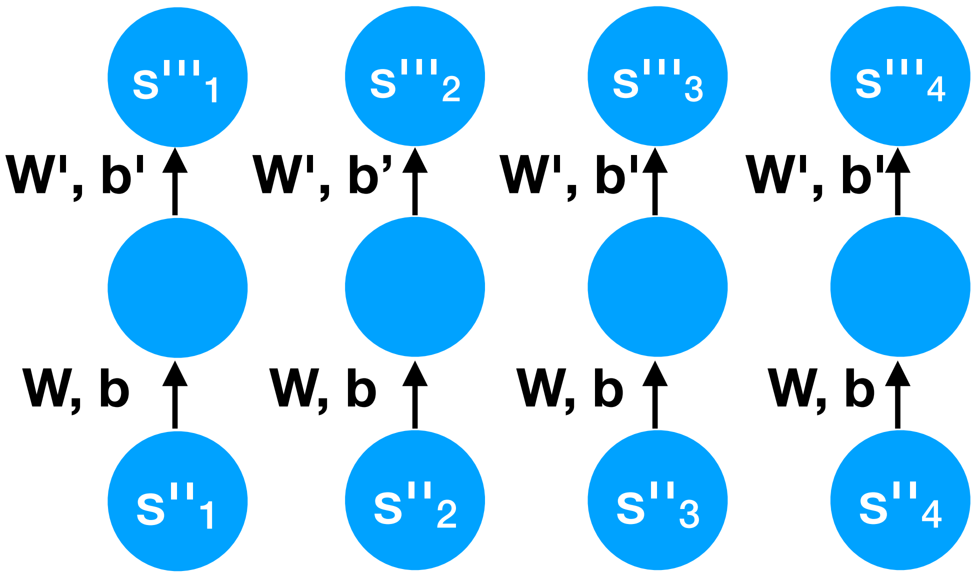
Point-wise FC layer

- The point-wise FC layer actually consists of two distinct operations:



Point-wise FC layer

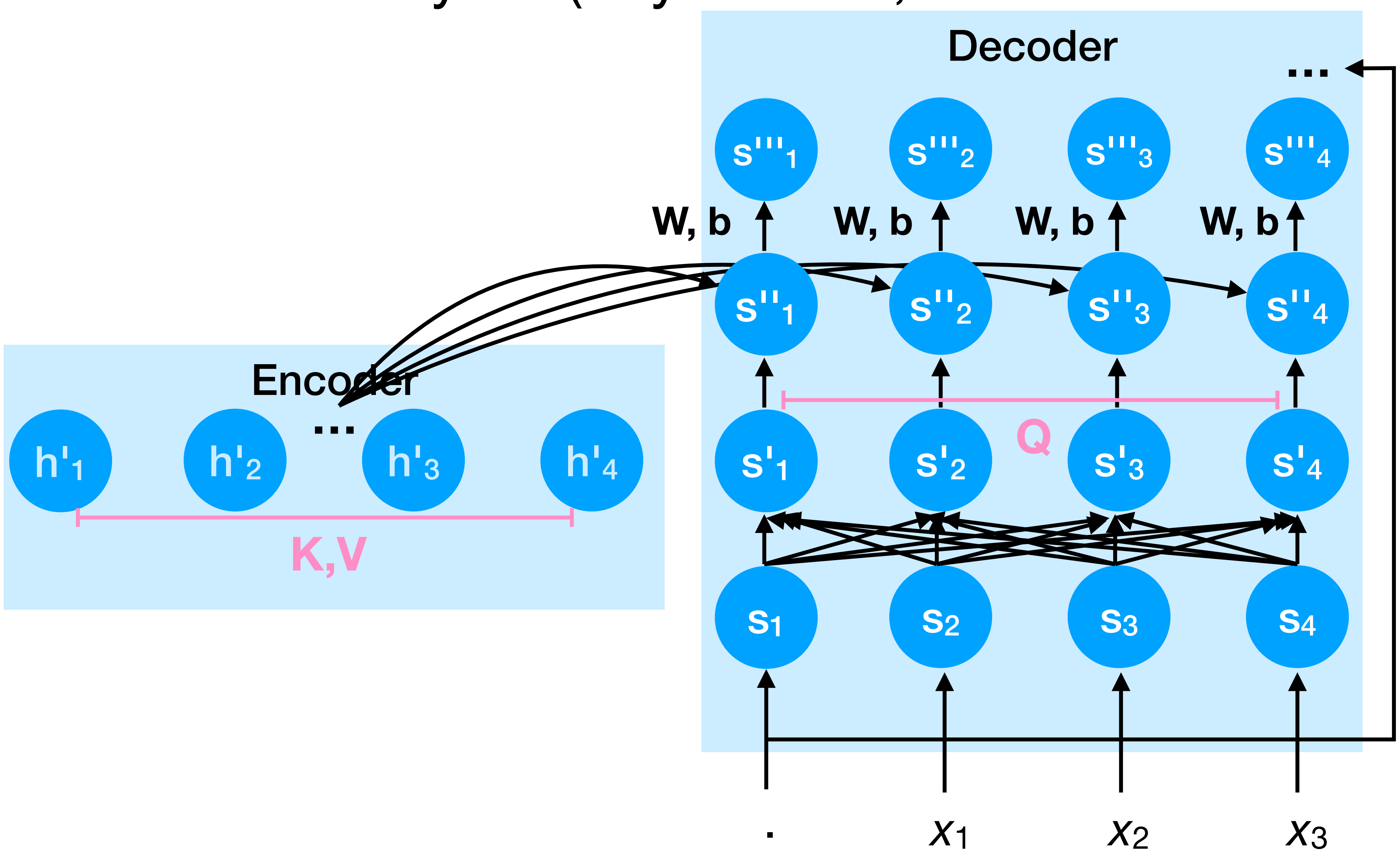
- The point-wise FC layer actually consists of two distinct operations:
  - Project from d_{model} to d_{FC}.
  - Project from d_{FC} to d_{model}.



- In this way, the input & output dimensions of the point-wise FC “layer” is preserved.

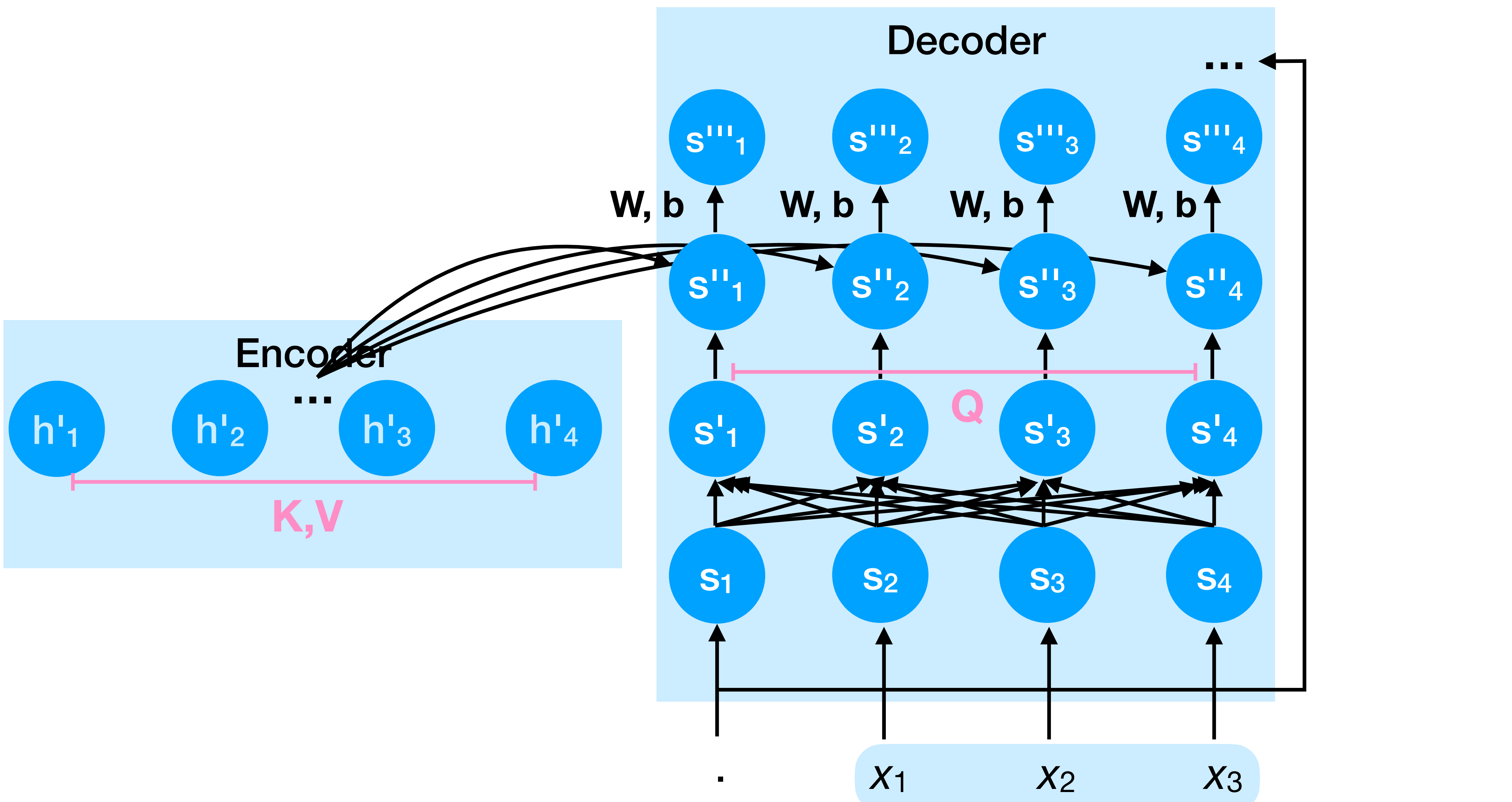
Transformer

- Finally, to improve training & testing performance, Transformers use skip connections (like in Resnet) and normalization layers (LayerNorm, similar to BatchNorm).

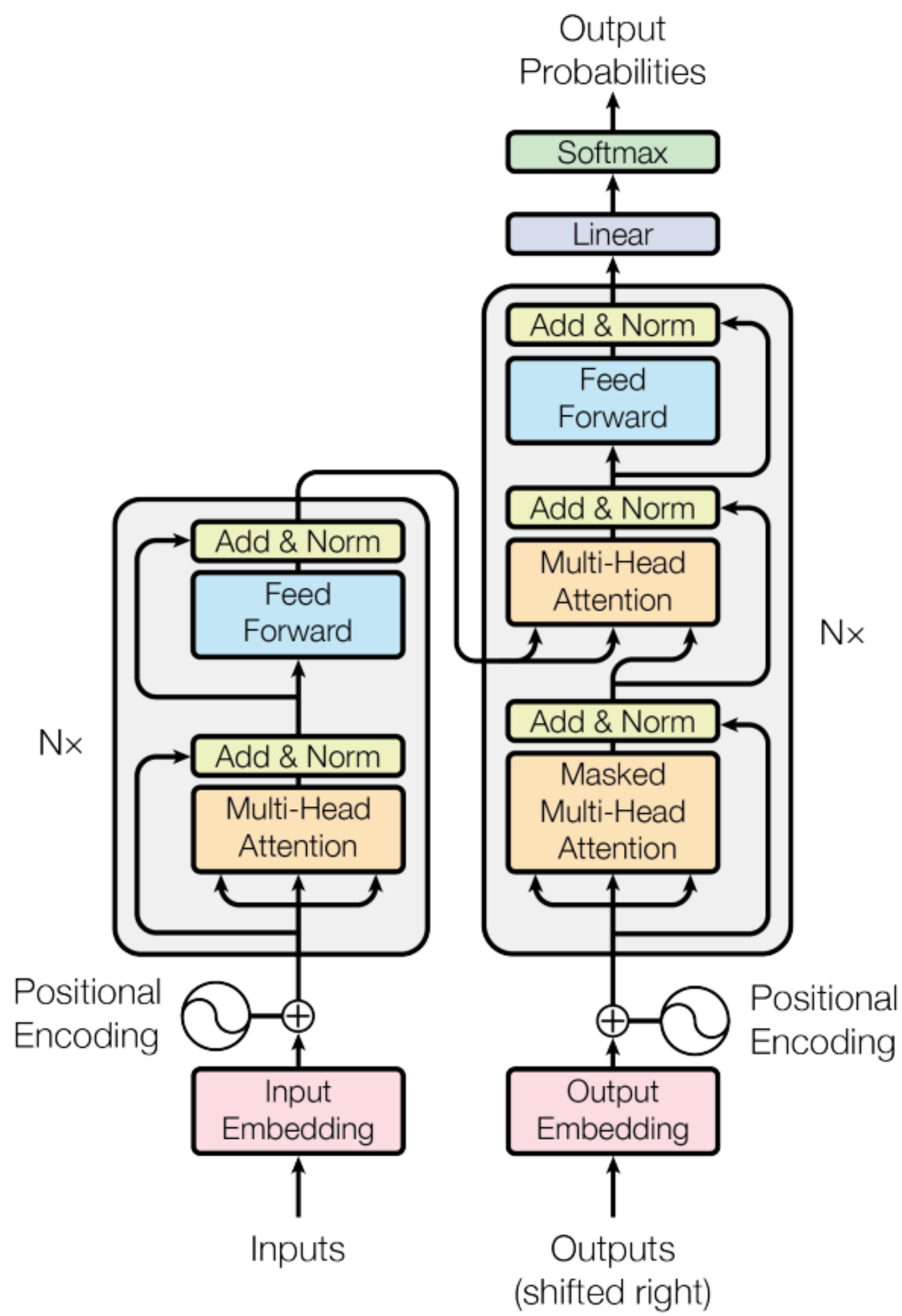


Transformer

- With teacher forcing, Transformers have **no temporal dependencies** at training time! We can thus parallelize over t.



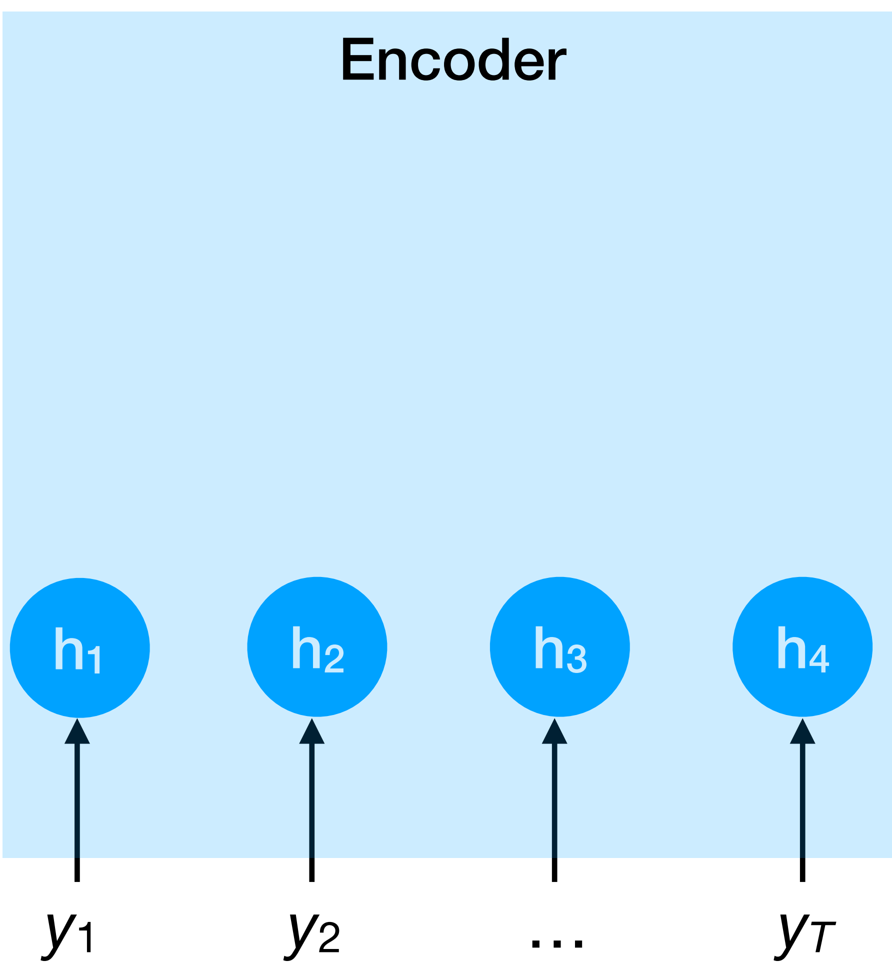
Transformer



Transformers: inference

Tranformers: inference

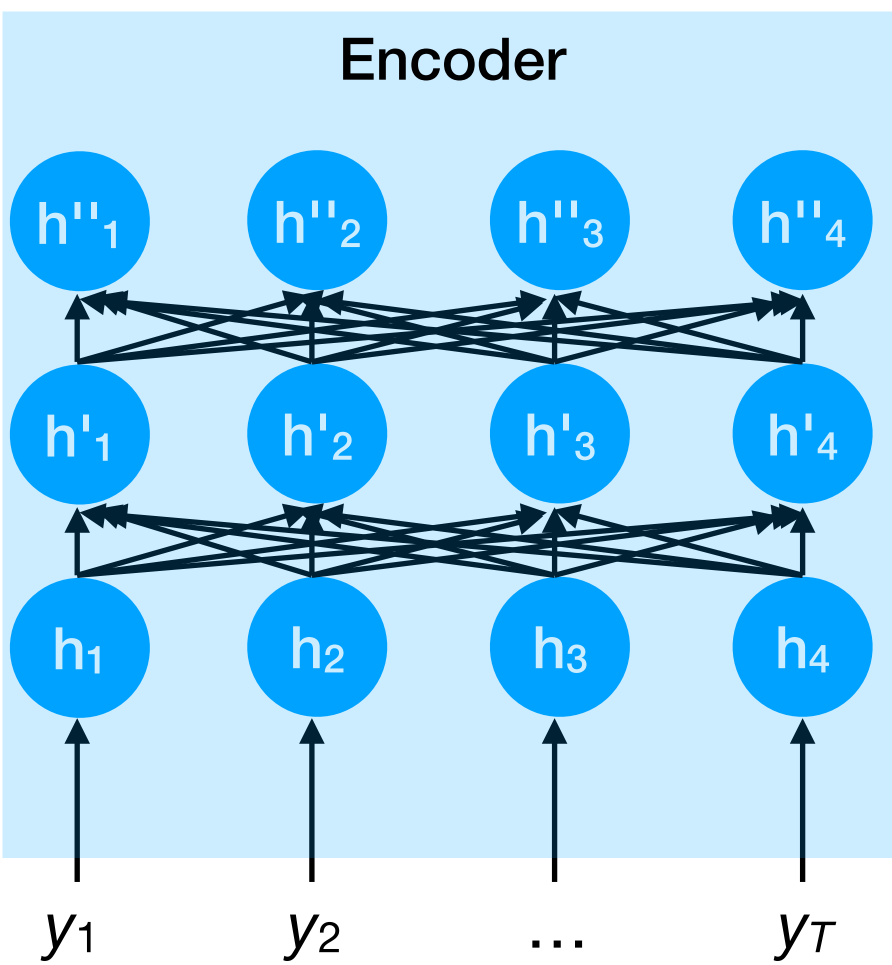
1. Input the input sequence $y_1, \ldots, y_T$ into the encoder.



75

Tranformers: inference

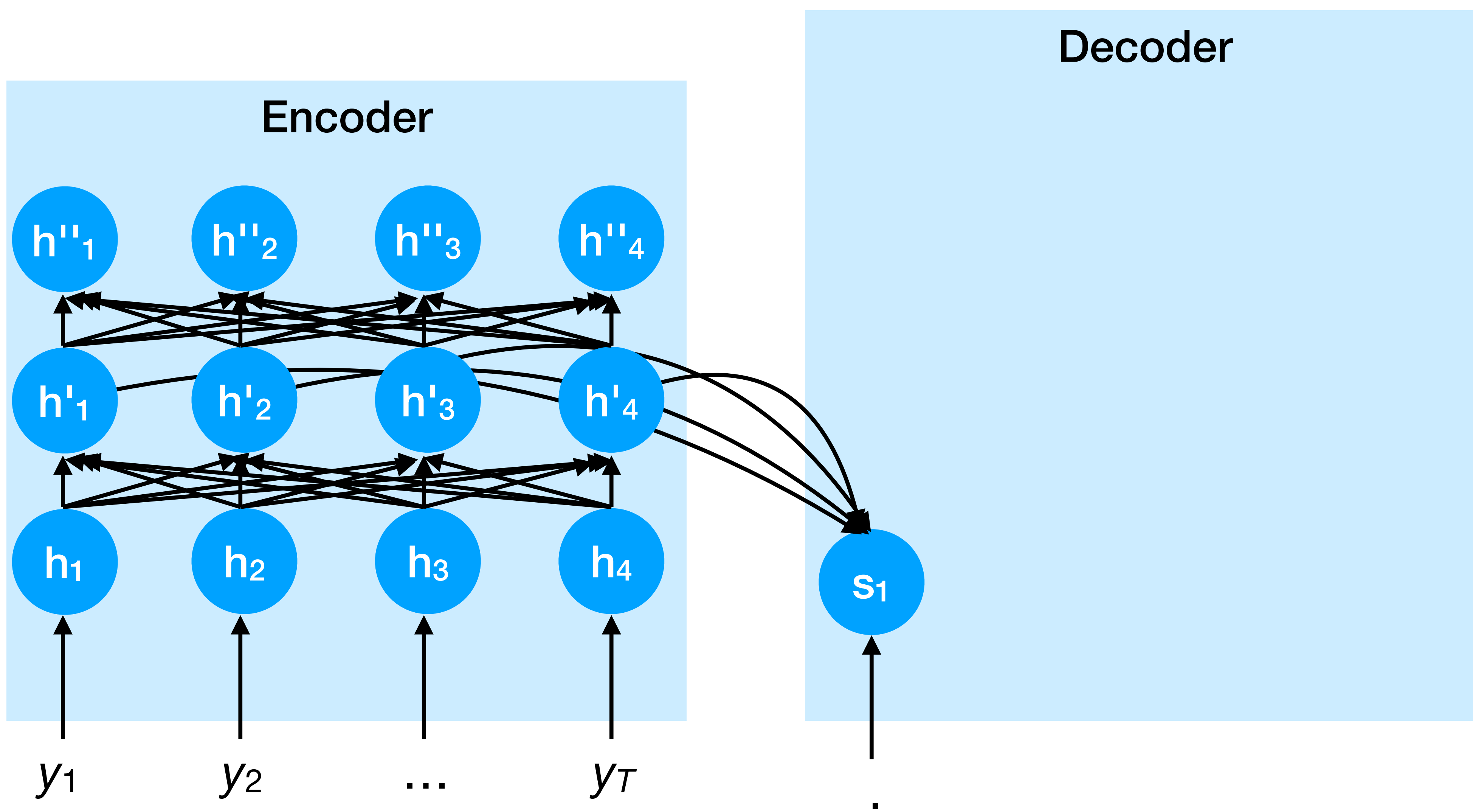
2. Compute hidden encoder representations $\mathbf{h}_1, \ldots, \mathbf{h}_T$ layer-by-layer using self-attention.



76

Tranformers: inference

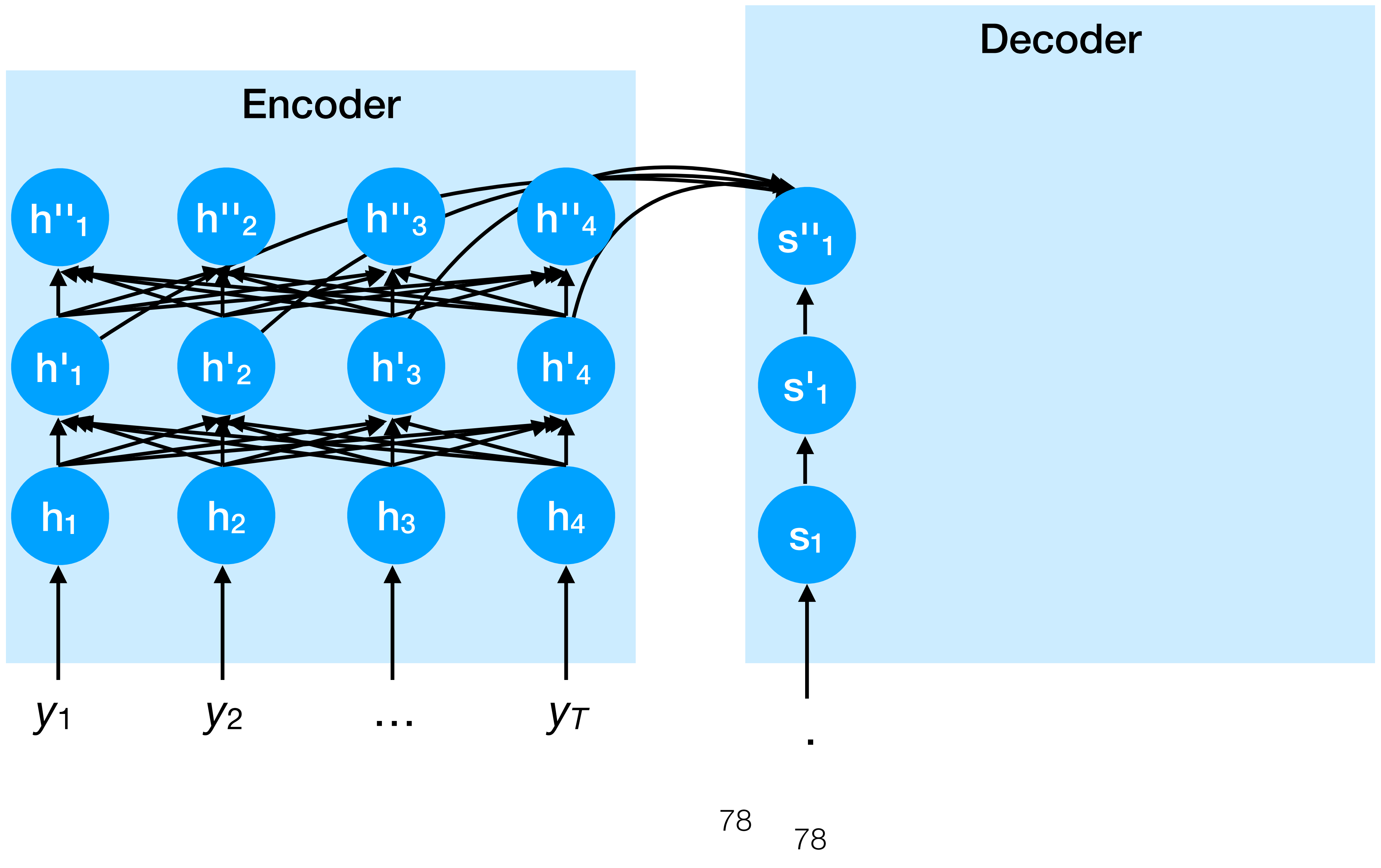
3. Given hidden codes from encoder, compute hidden decoder representations $\mathbf{s}_1, \ldots, \mathbf{s}_{T'}$ layer-by-layer using self-attention and cross-attention.



77

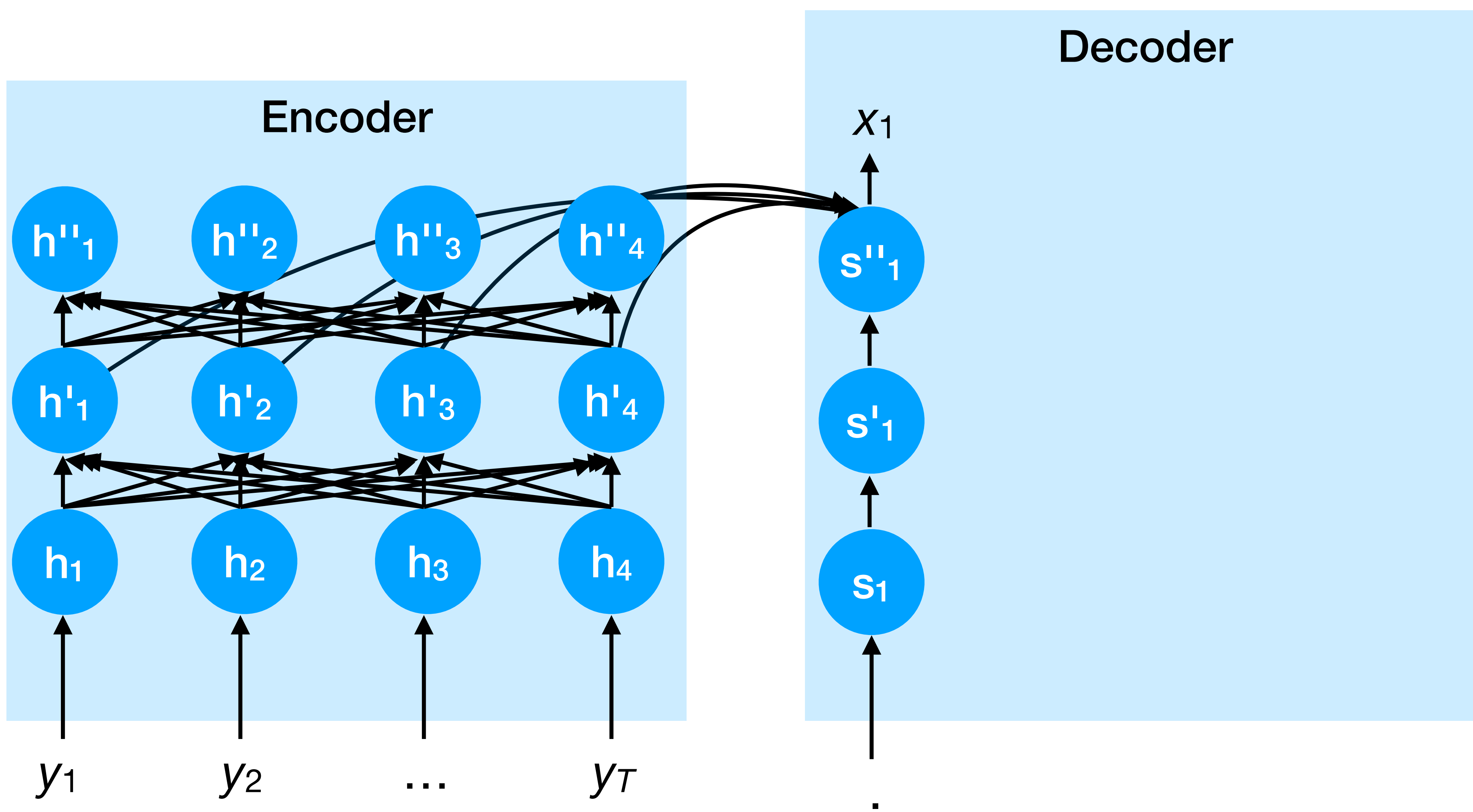
Tranformers: inference

3. Given hidden codes from encoder, compute hidden decoder representations $\mathbf{s}_1, \ldots, \mathbf{s}_{T'}$ layer-by-layer using self-attention and cross-attention.



Tranformers: inference

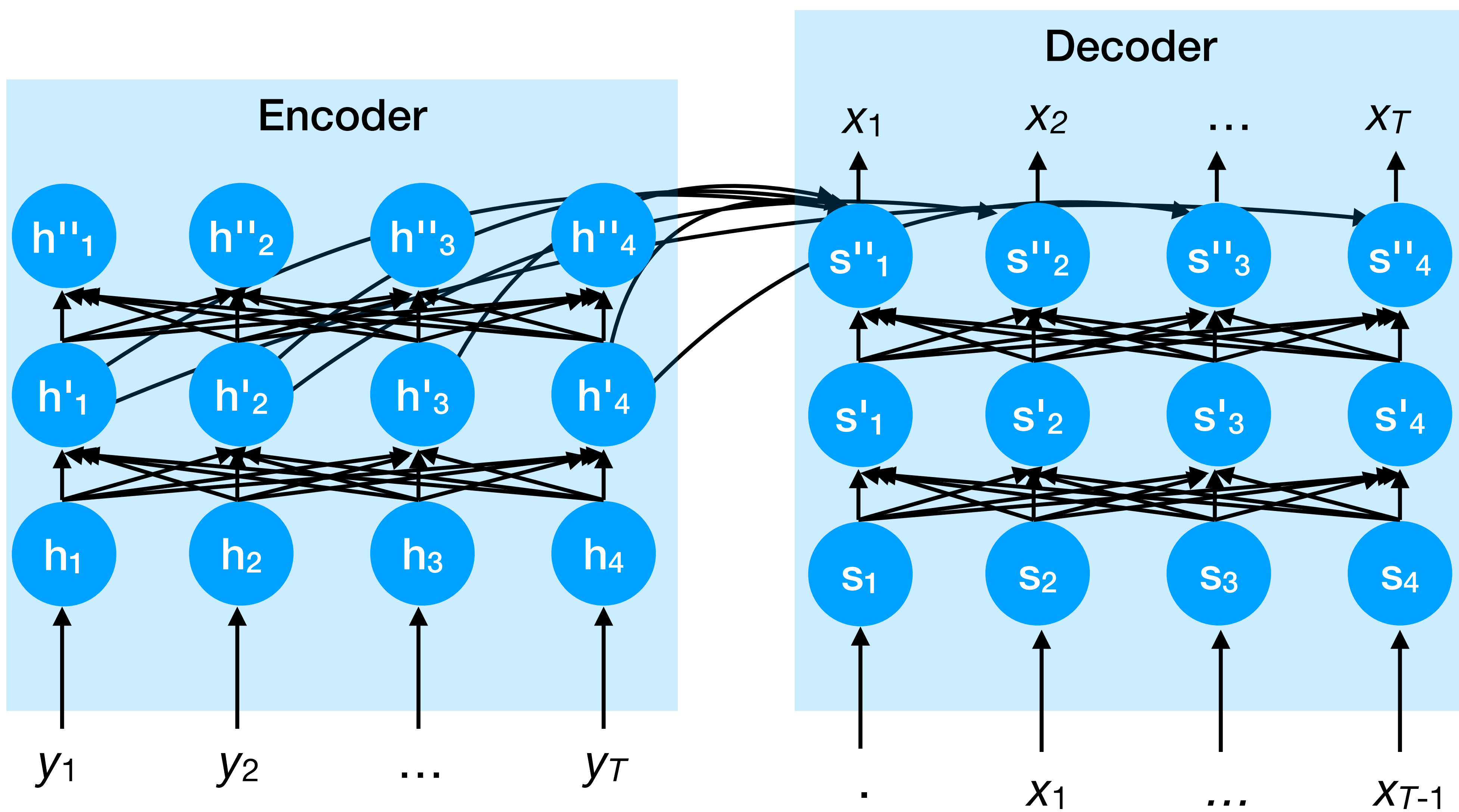
4. Given final hidden codes from encoder, predict x_1.



79

Tranformers: inference

5. Autoregressively predict x_t from $x_{t\text{-}1}$ until "." symbol.



80

Exercise

- Suppose the input embedding dimension is 10.
- Suppose the input **x** has length T=4.
- How many trainable parameters are contained (in total) in the $\mathbf{W}_q$, $\mathbf{W}_k$, and $\mathbf{W}_v$ matrices of each Transformer encoder layer?
  a. 40
  b. 30
  c. 100
  d. 300
  e. 1200