Neural Networks for Sequence Data

CMPUT 261: Introduction to Artificial Intelligence

GBC §10.0-10.2
P §12.1-12.2, §12.4, §12.6

Lecture Outline

1. Recap & Logistics
2. Unfolding Computations
3. Recurrent Neural Networks
4. Attention & Transformers

After this lecture, you should be able to:

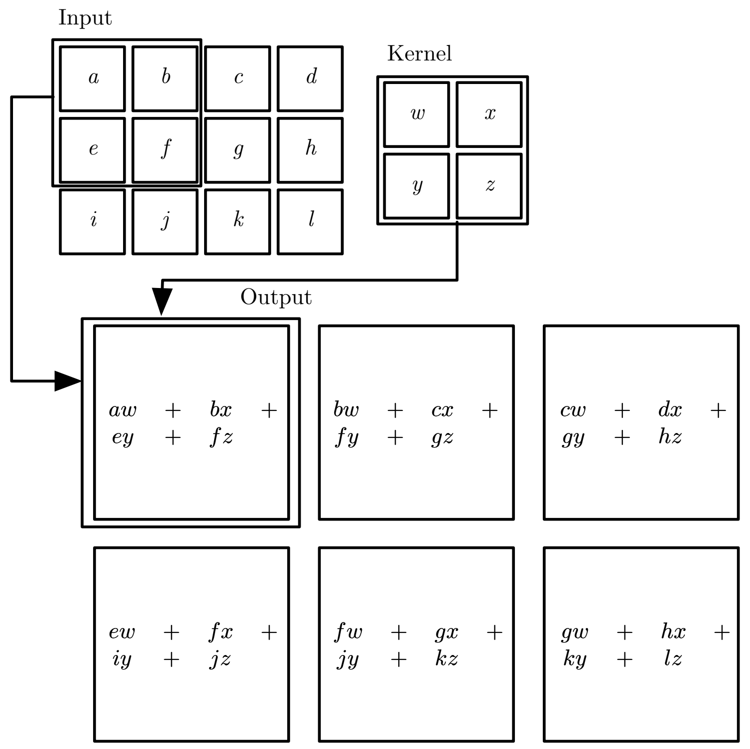
- demonstrate unfolding a recurrent expression
- explain the problems with handling sequence input using dense or convolutional neural networks
- explain the high-level idea behind neural networks and transformers
- describe how self-attention combines inputs to generate its outputs
- describe the architecture of a transformer layer
- explain the high-level idea behind encoder-decoder architectures

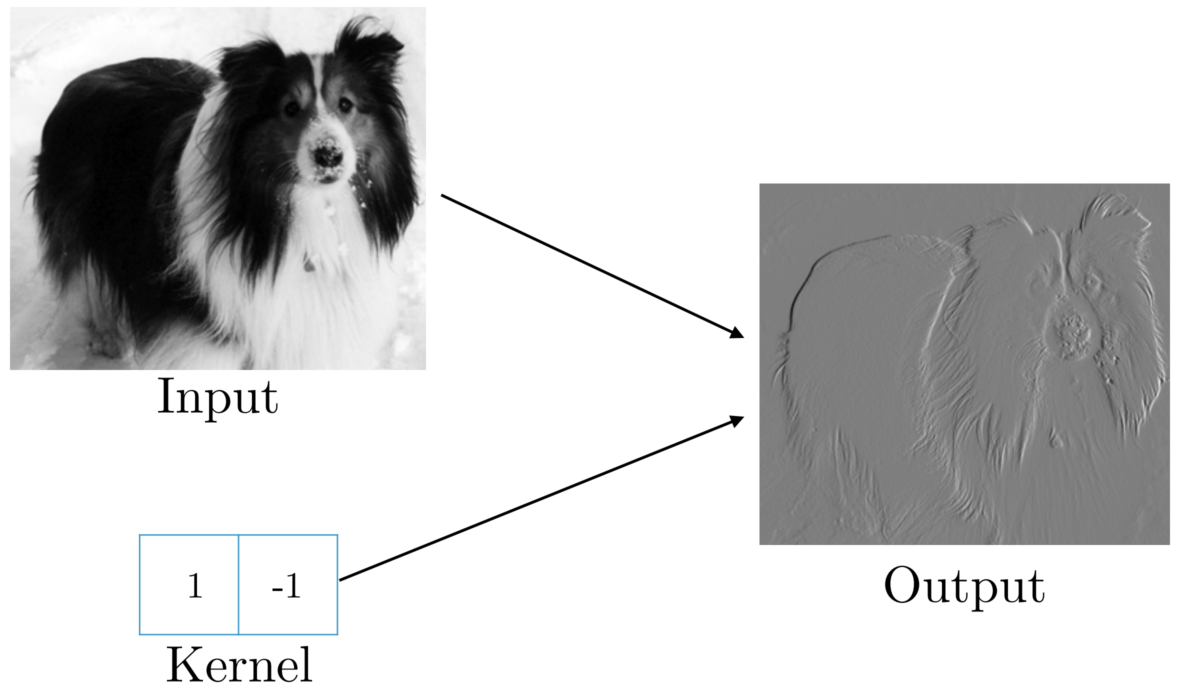
Logistics

- **Assignment #3** is available
  - Due ~~Tuesday, March 26~~ **Wednesday, March 27**
  - Submit via eClass
  - Please **submit the correct files**
- **Assignment #2 and midterm** marks are released

Recap:
Convolutional Neural Networks

- Convolutional networks: Specialized architecture for **images**
- Number of **parameters** controlled by using **convolutions** and **pooling** operations instead of **dense connections**
- Fewer parameters means more **efficient to train**





(Images: Goodfellow 2016)

Sequence Modelling

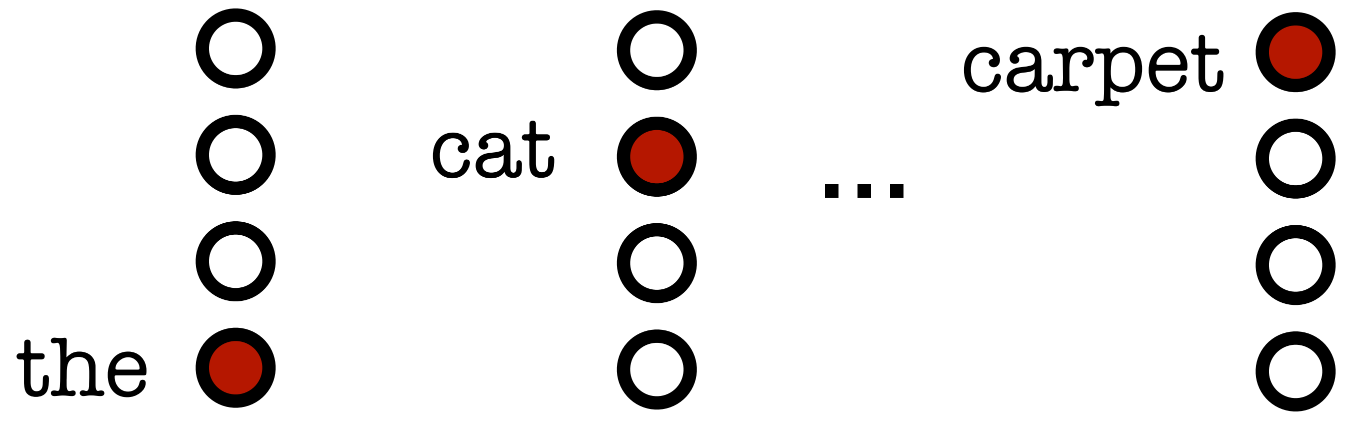
- For many tasks, especially involving language, we want to model the behaviour of **sequences**
- **Example:** Translation
  - The cat is on the carpet $\implies$ Le chat est sur le tapis
- **Example:** Sentiment analysis
  - This pie is great $\implies$ POSITIVE
  - This pie is okay, not great $\implies$ NEUTRAL
  - This pie is not okay $\implies$ NEGATIVE

Sequential Inputs

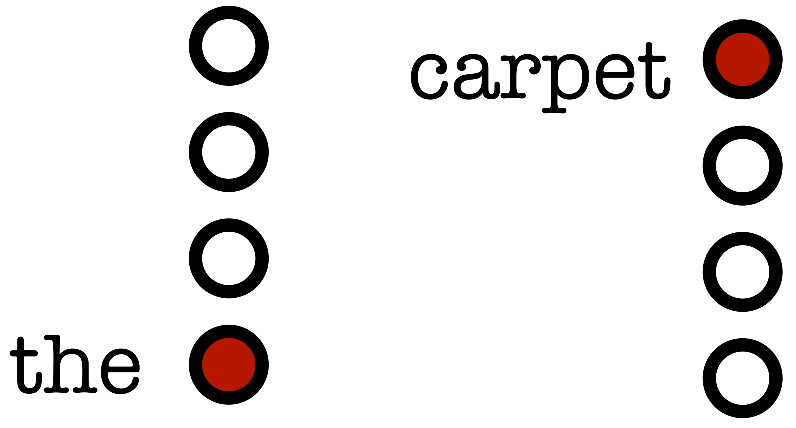
The cat is on the carpet

Question: How should we **represent** sequential input to a neural network?

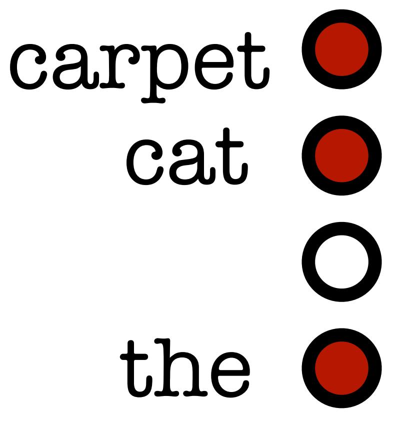
1. 1-hot vector for **each word**
   (Sequence must be a specific length?)



2. 1-hot vector for **last few words**
   (n-gram)



3. **Single vector** indicating each word that is present
   (bag of words)



One-Hot Representations

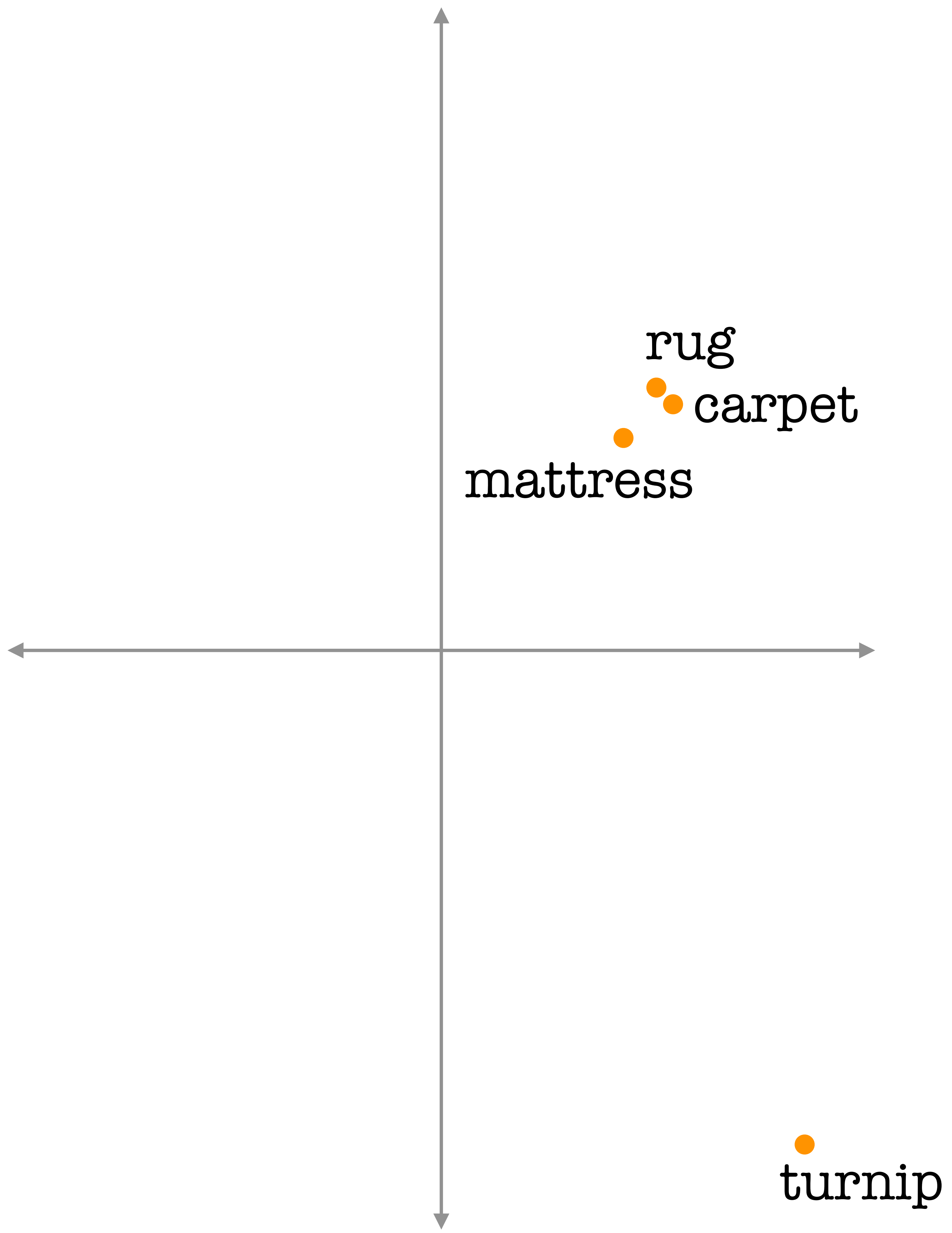
$$\begin{matrix} \\ \\ \\ \\ \\ \\ \\ \texttt{carpet} \\ \end{matrix}\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ \mathbf{1} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

One-hot representations of words have some problems:

1. **Wasteful:** Each input vector must have a dimension equal to the size of the vocabulary (possible words)
    - If vocabulary has 30,000 words, then each vector has 29,999 zeros
2. **Poor generalization:** Ideally, similar words would be treated similarly
    - Exploiting meaningful similarity between images was an important feature of convolutional neural networks

Semantic Embeddings

- The usual approach is to first learn a **semantic embedding** for one-hot vectors
- Every word gets represented as a **dense vector** with smaller dimension than the vocabulary (typical size: 1,024)
- **Goal:** Words with similar **meanings** will have small **distance** between embedded vectors; words with different meanings will have large **distance** between embedded vectors



(Pre-)Training Semantic Embeddings

Question: How many **parameters** are required to convert a one-hot encoding for vocabulary of V words into a D-dimensional embedding?

- Embeddings require the training of ***many*** parameters
- Fortunately, this can be done with **unlabeled** data
- **Trick:** "Pre-train" neural network for a task that we don't care about
  - But which can be evaluated using unlabeled data
  - Predicting words from k nearby words
  - Predicting "masked" words
- Keep the weights that convert the one-hot layer into a dense embedding layer
- Throw away the weights that convert the embedding layer into output

Processing Variable-Length Sequences

- Image inputs can be restricted to a **standard size** (20x20, 1024x768, etc.)
- Sequence inputs (e.g., text) are **variable-length**
  - And often **very long**
- **Solution:** Apply the **same operations** to each position in the sequence
- Two such approaches:

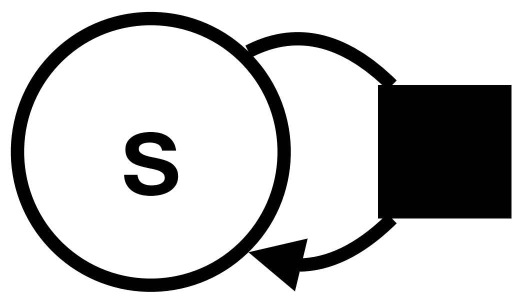
1. **Recurrent neural networks:**
   input is current token + fixed-dimension "state" from previous operation
2. **Transformers / self-attention:** Size of state varies with size of sequence

Dynamical Systems

- A **dynamical system** is a system whose state at time $t + 1$ depends on its state at time t:

$$\mathbf{s}^{(t)} = f(\mathbf{s}^{(t-1)}; \theta)$$

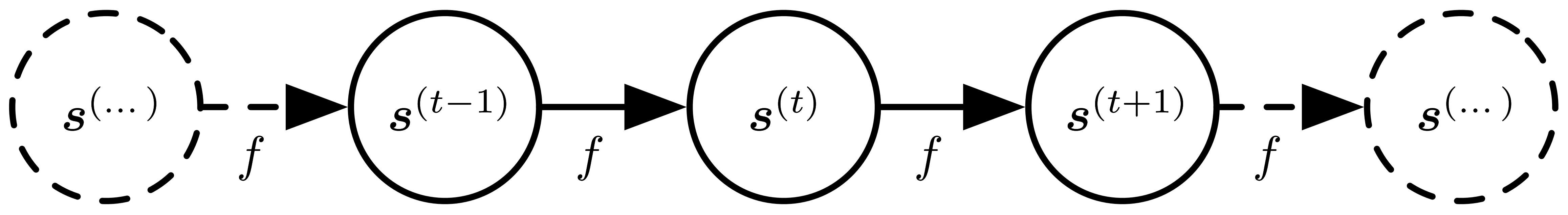
- An expression that depends on the same expression at an earlier time is **recurrent**.



Unfolding Computations

- A recurrent expression can be converted to a non-recurrent expression by **unfolding**:

$$\begin{aligned}\mathbf{s}^{(3)} &= f(\mathbf{s}^{(2)}; \theta) \\ &= f(f(\mathbf{s}^{(1)}; \theta); \theta)\end{aligned}$$



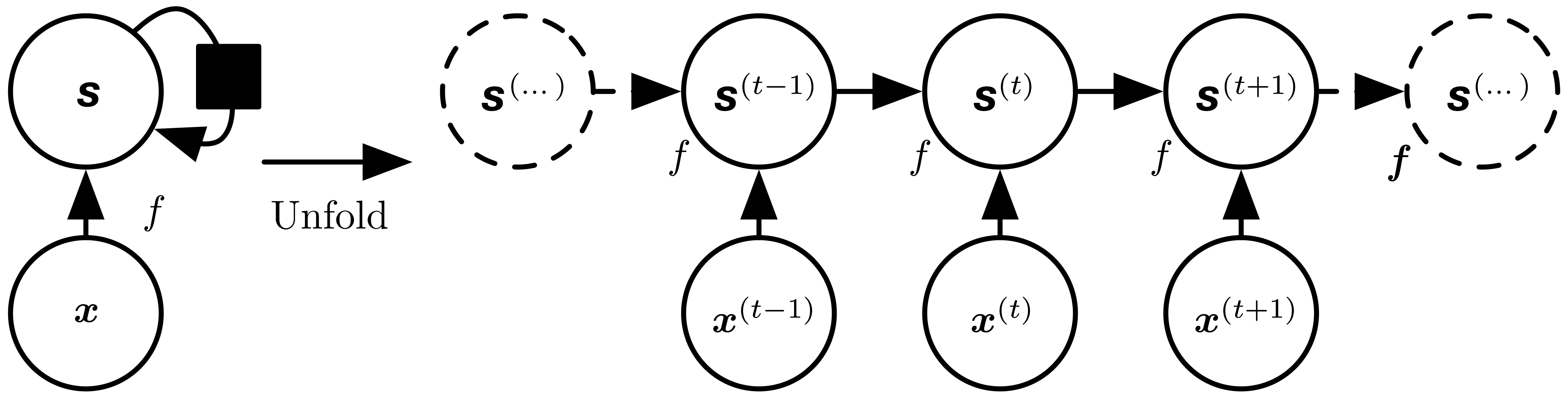
(Image: Goodfellow 2016)

External Signals

- Dynamical systems can also be driven by **external signals**:

$$\mathbf{s}^{(t)} = f(\mathbf{s}^{(t-1)}, \mathbf{x}^{(t)}; \theta)$$

- These systems can also be represented by non-recurrent, unfolded computations:

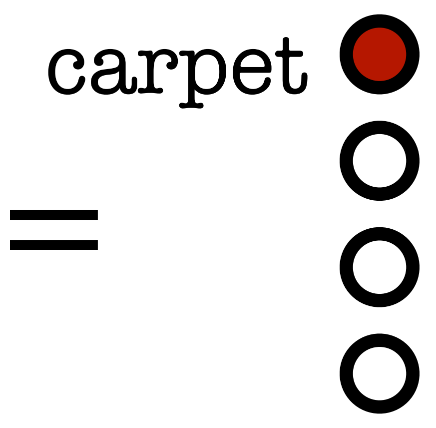


(Image: Goodfellow 2016)

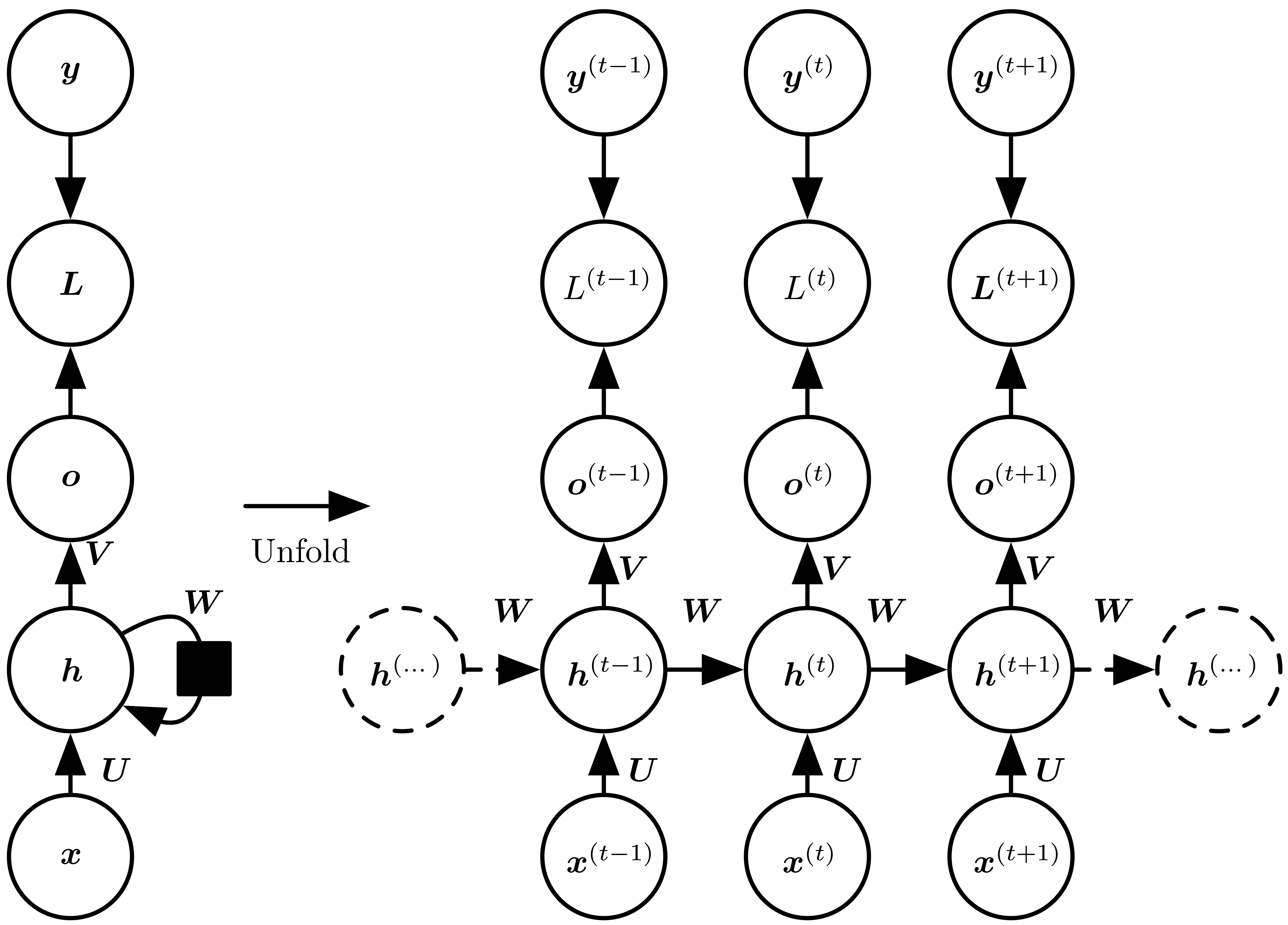
Recurrent Neural Networks

- Recurrent neural network: a specialized architecture for modelling **sequential data**
- Input presented **one element at a time**
- Parameter sharing by:
  - Treating the sequence as a system with **state**
  - Introducing hidden layers that **represent** state
  - Computing **state transitions** and **output** using **same functions** at each stage
- The **same computation** is applied to each pair of state and input
  - But the state is different after each application

$\mathbf{x}^{(6)} =$



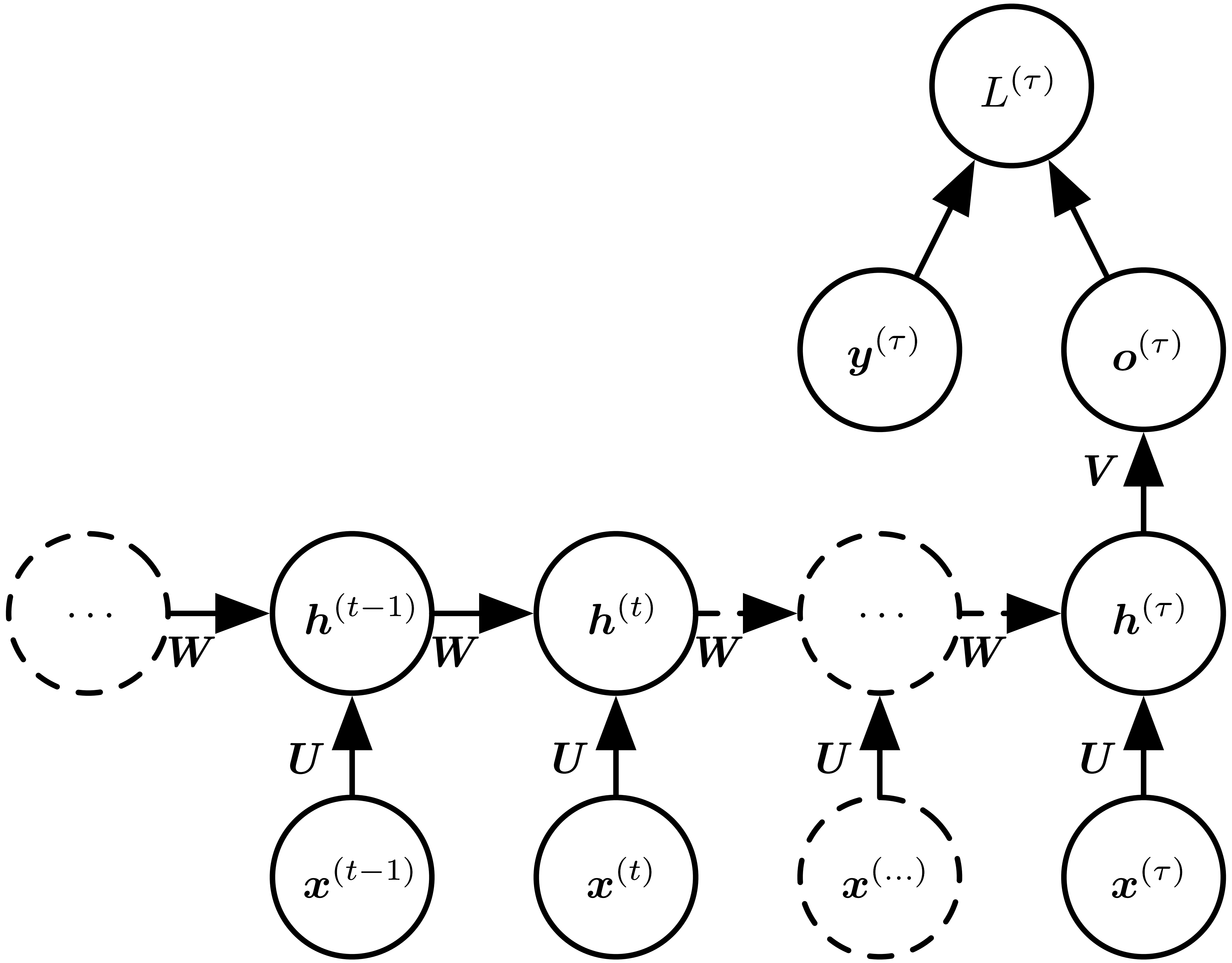
Recurrent Hidden Units: Sequence to Sequence



- **Input values** $\mathbf{x}$ connected to **hidden state** $\mathbf{h}$ by weights $\mathbf{U}$
- Hidden state $\mathbf{h}$ mapped to **output** $\mathbf{o}$ by weights $\mathbf{V}$
- Hidden state $\mathbf{h}^{(t-1)}$ connected to hidden state $\mathbf{h}^{(t)}$ by weights $\mathbf{W}$
- Gradients computed by **back propagation through time**: from final loss all the way back to initial input.
- All hidden states computed must be **stored** for computing gradients

(Image: Goodfellow 2016)

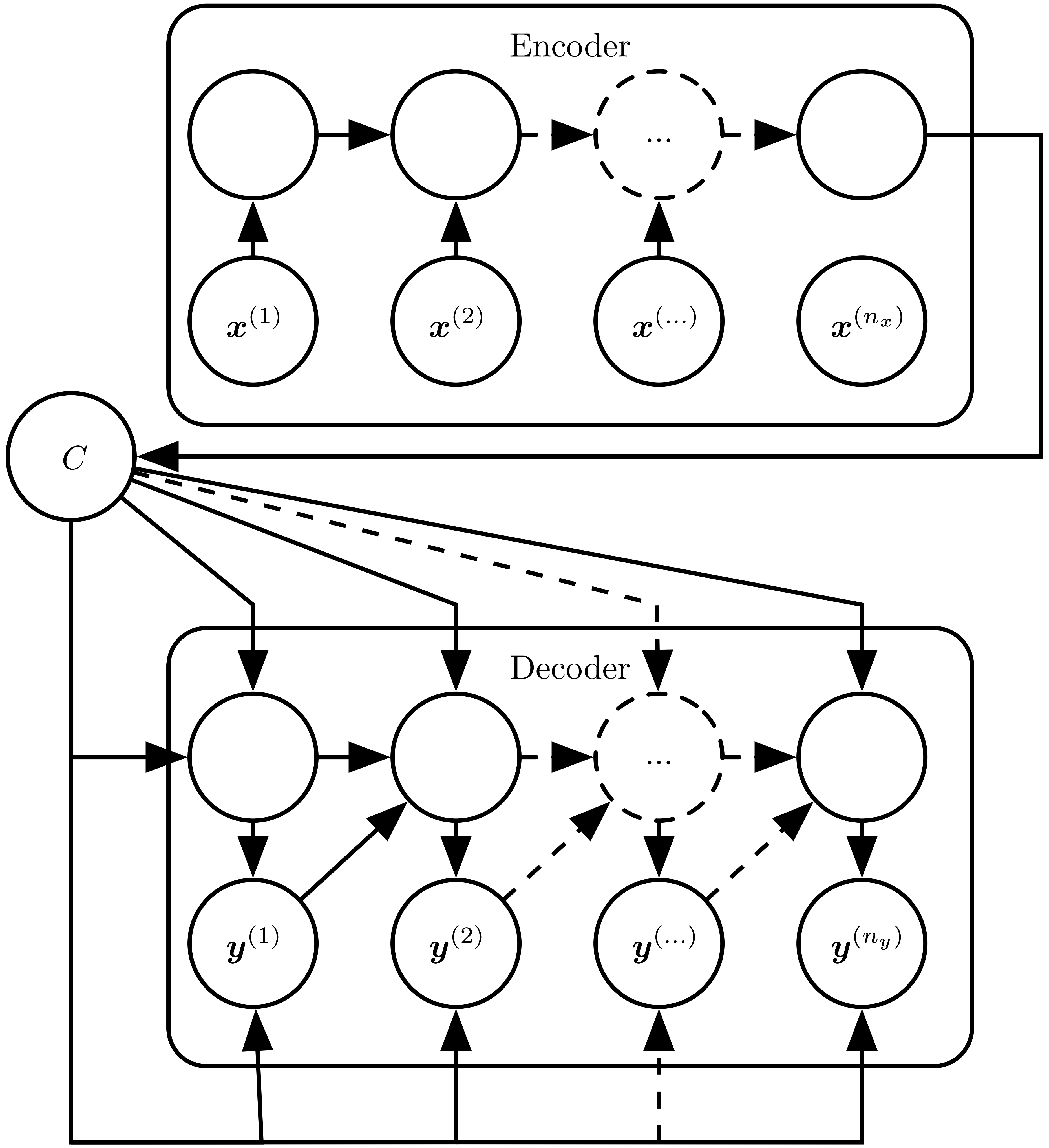
Recurrent Hidden Units: Sequence to Single Output



- Update state as inputs are provided
- Only compute a **single** output at the **end**
- $\mathbf{W}, \mathbf{U}$ still shared at every stage
- Back propagation through time still requires **evaluating every state** in gradient computation

(Image: Goodfellow 2016)

Encoder/Decoder Architecture for Sequence to Sequence



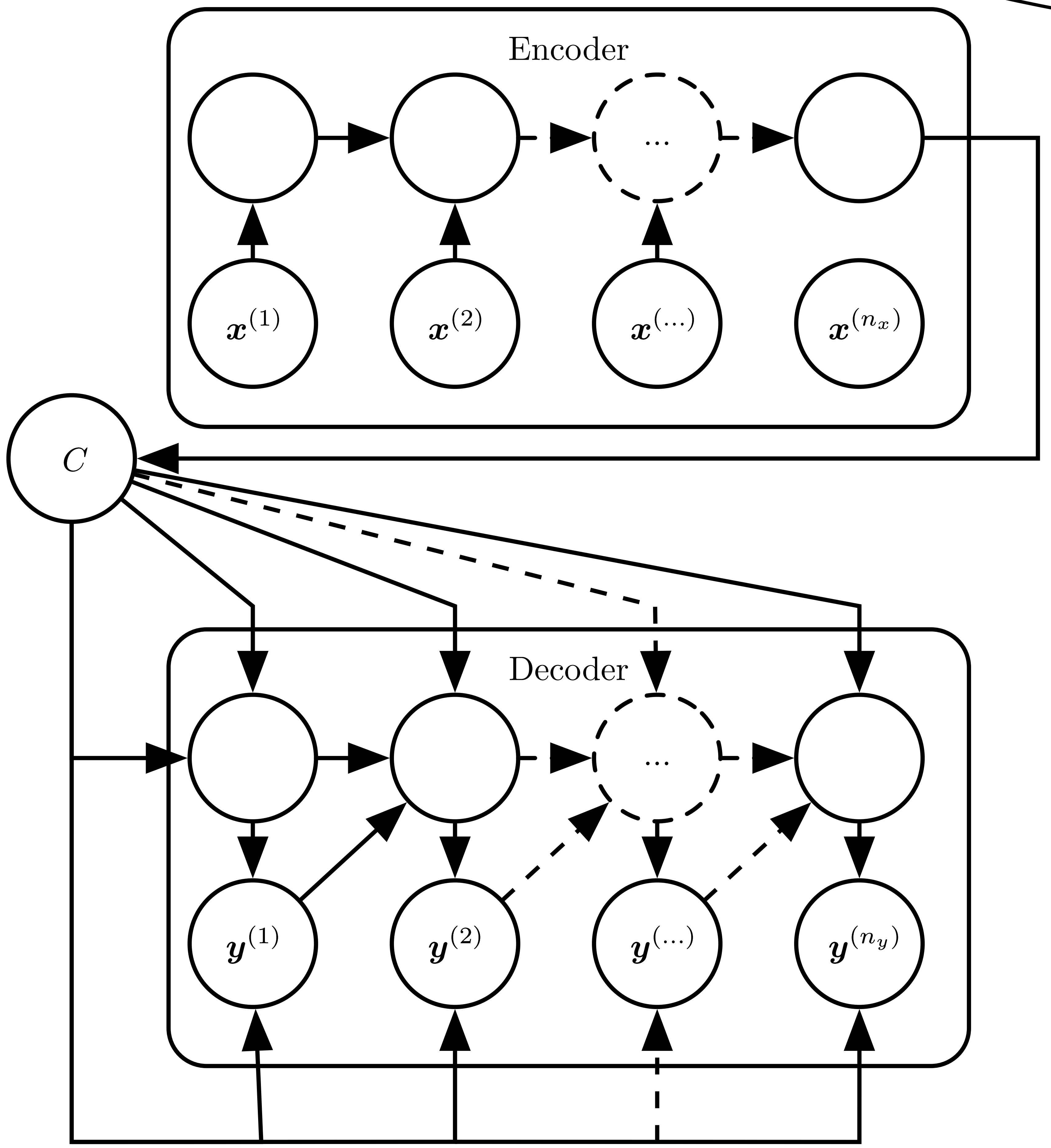
Can **combine approaches** for sequence-to-sequence:

1. Accept entire input to construct a single "**context**" output $\mathbf{C}$
2. Construct new sequence using context $\mathbf{C}$ as **only input**

(Image: Goodfellow 2016)

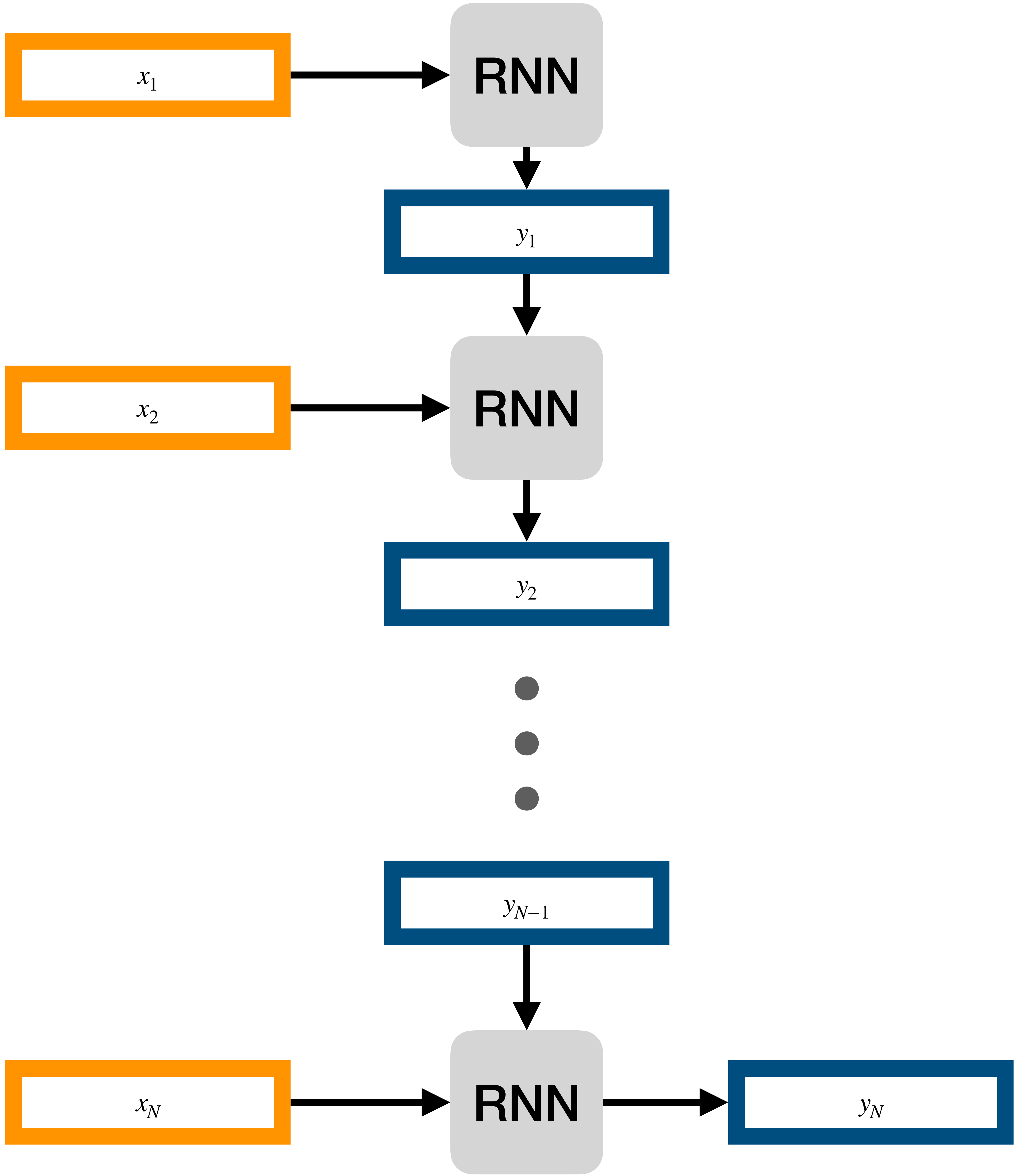
Long-Range Dependence

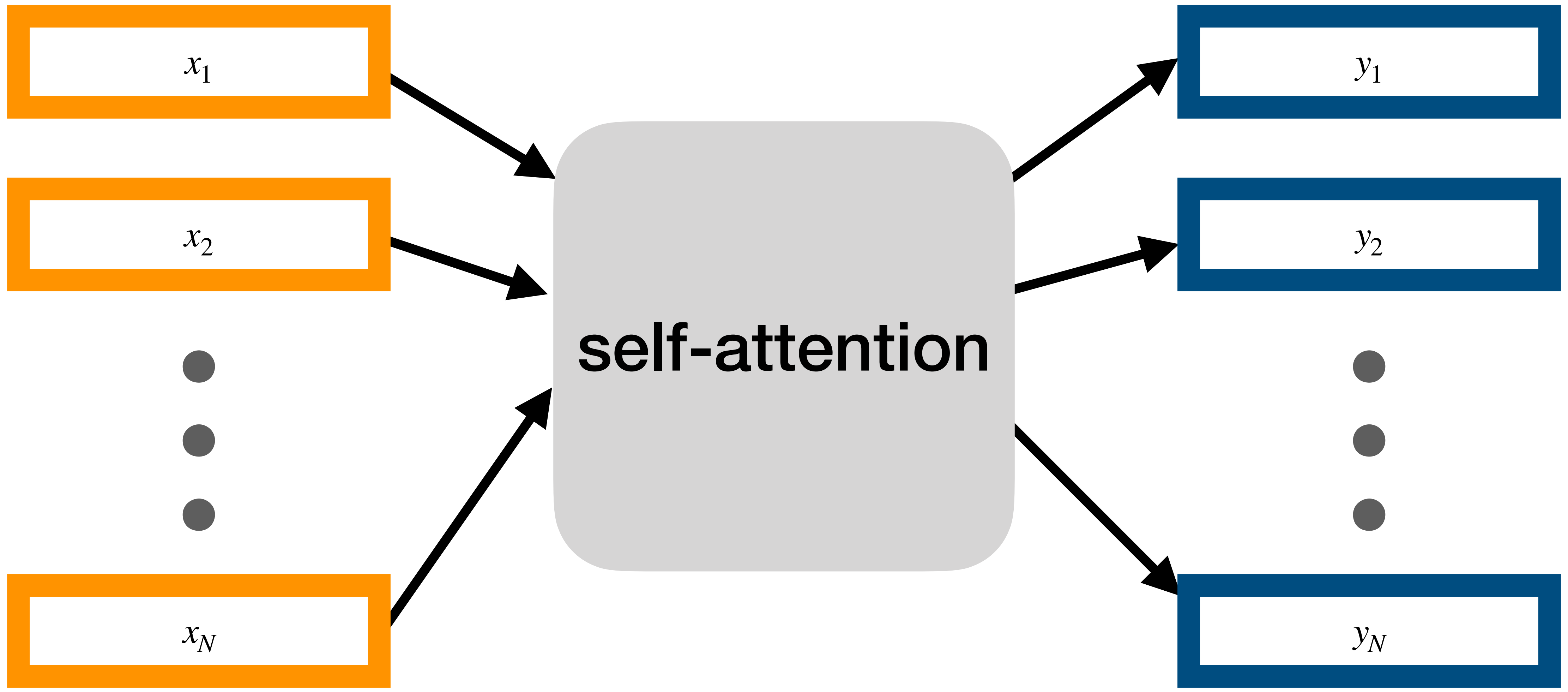
The submarine, which was the subject of a well known song by the Beatles, was yellow.



- Information sometimes needs to be **accumulated** for a long part of the sequence
- But **how long** an individual piece of information should be accumulated is **context-dependent**
- Long-range dependence can be difficult for a recurrent network
  - Often need to **accumulate** information in the state, and then **forget** it later

Self-Attention vs. RNN





- **RNN:** accept "previous" state and current input; output "next" state
  - Final output is last state
- **Self-attention:** Accept ALL inputs
  - Final output is ALL states

Self-Attention

- Each input is transformed into a single output
  - N inputs means N outputs
- An output is computed by:

1. Each **input** x_i transformed into **value** v_i by a **linear** operation

$$v_i = \beta_v + \Omega x_i$$

2. Each **output** y_j is a **weighted combination** of the **values**:

$$y_j = \sum_i \boxed{a[x_i, x_j]} v_i$$

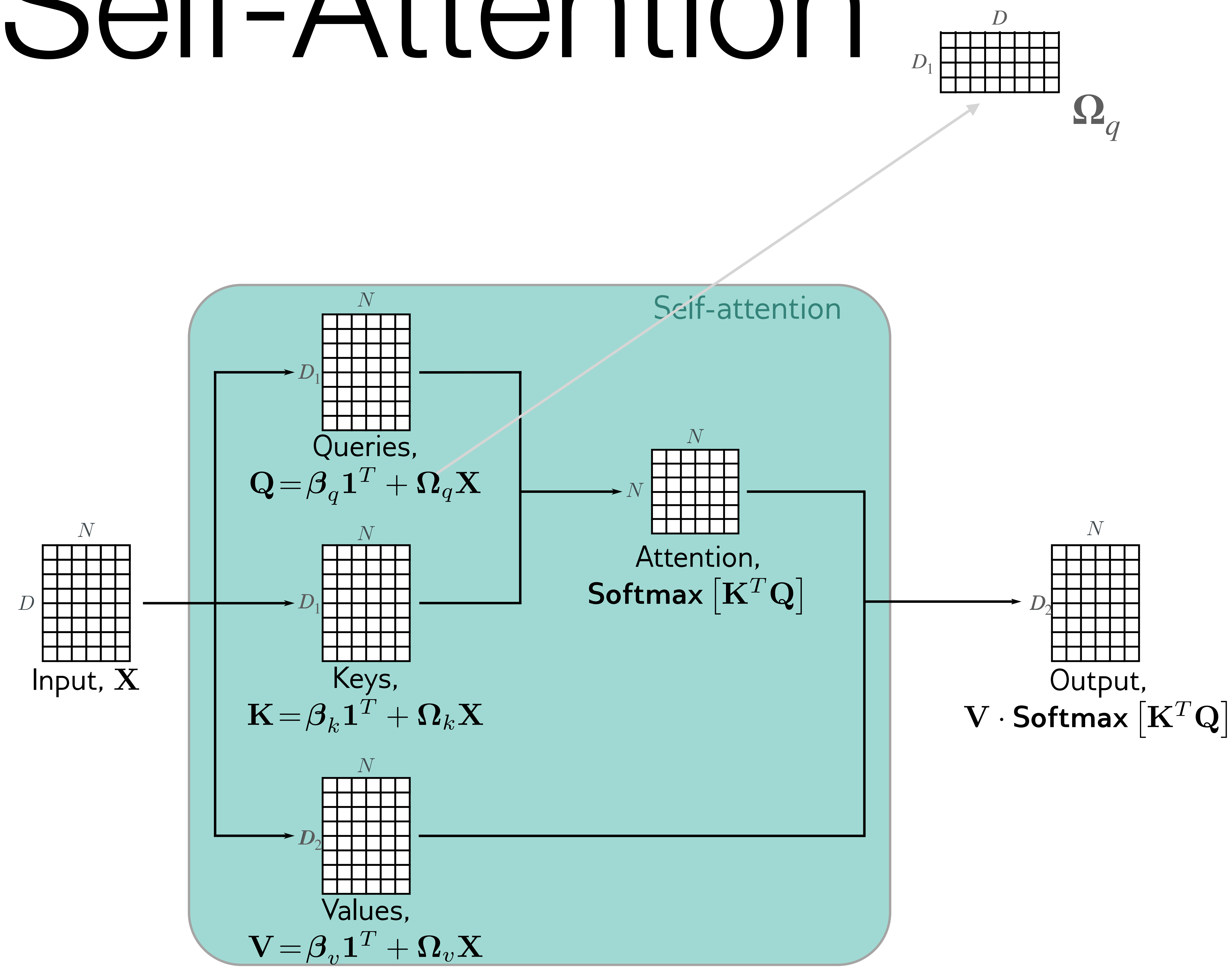
Dot-Product Self-Attention

- A **self-attention unit** computes **three values** for each input x_i:
    - **Query** q_i, **Key** k_i, and **Value** v_i
    - These values are computed in the **same way** for each input
- Each output is a **weighted** combination of the **values** of **all** inputs:

$$y_j = \sum_i a[x_i, x_j] v_i = \sum_i w_{ij} v_i$$

- Weight for output y_j of **value** x_i is proportional to the dot-product of j's **query** and i's **key**

$$w_{ij} = \frac{\exp(q_j^\top k_i)}{\sum_\ell \exp(q_j^\top k_\ell)}$$



(Image: Prince 2022)

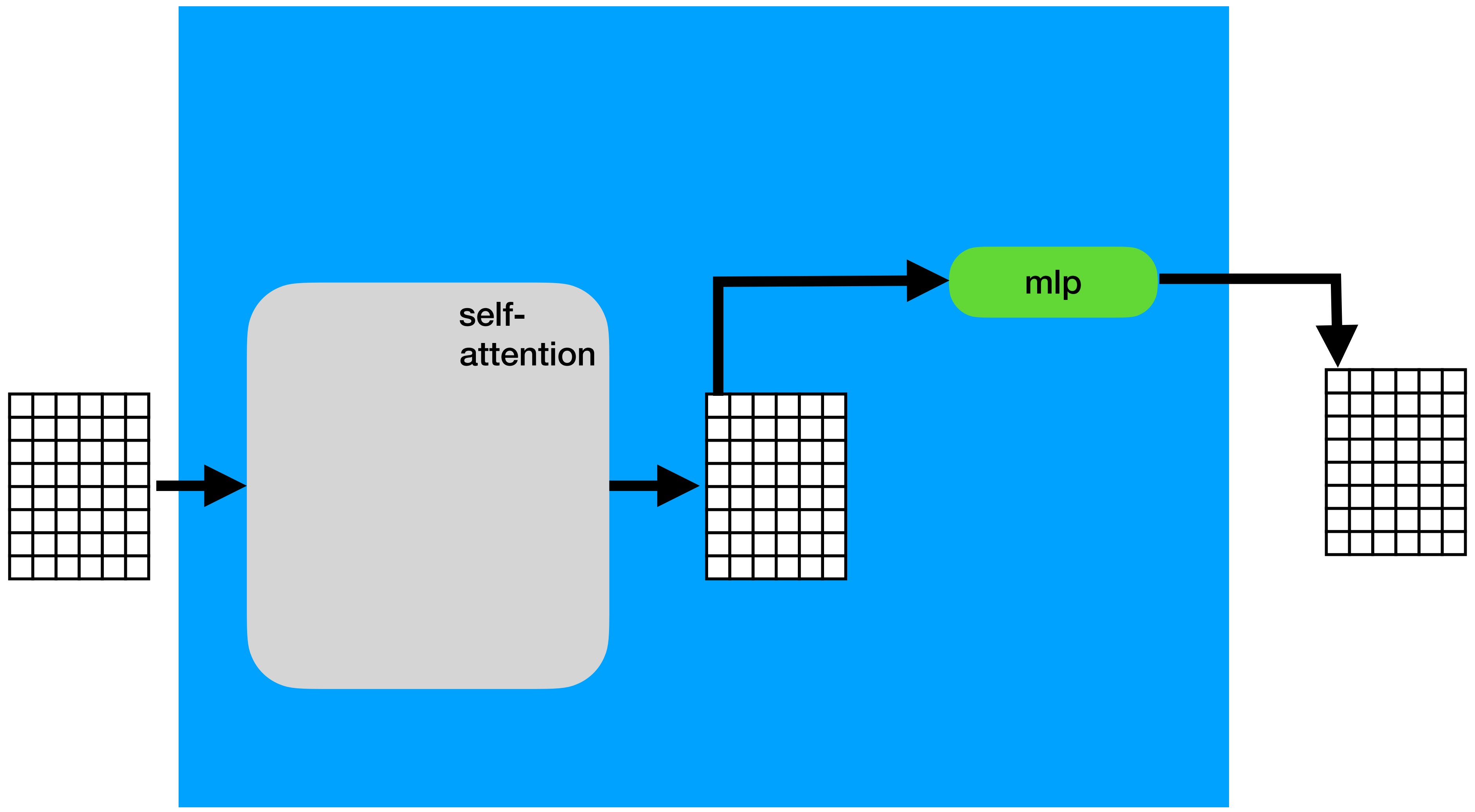
Transformer Blocks

- A **transformer layer** is a **self-attention** unit followed by a **dense feedforward network**

- The **same** feedforward network gets applied to each output of the self-attention unit:

$$y_j = \mathrm{mlp}(x_j; \Omega) \quad \text{for } j = 1, \ldots, N$$

- In a typical transformer architecture, several transformer blocks will be strung together in parallel ("multiple heads")



Training a Transformer Network (for encoding tasks)

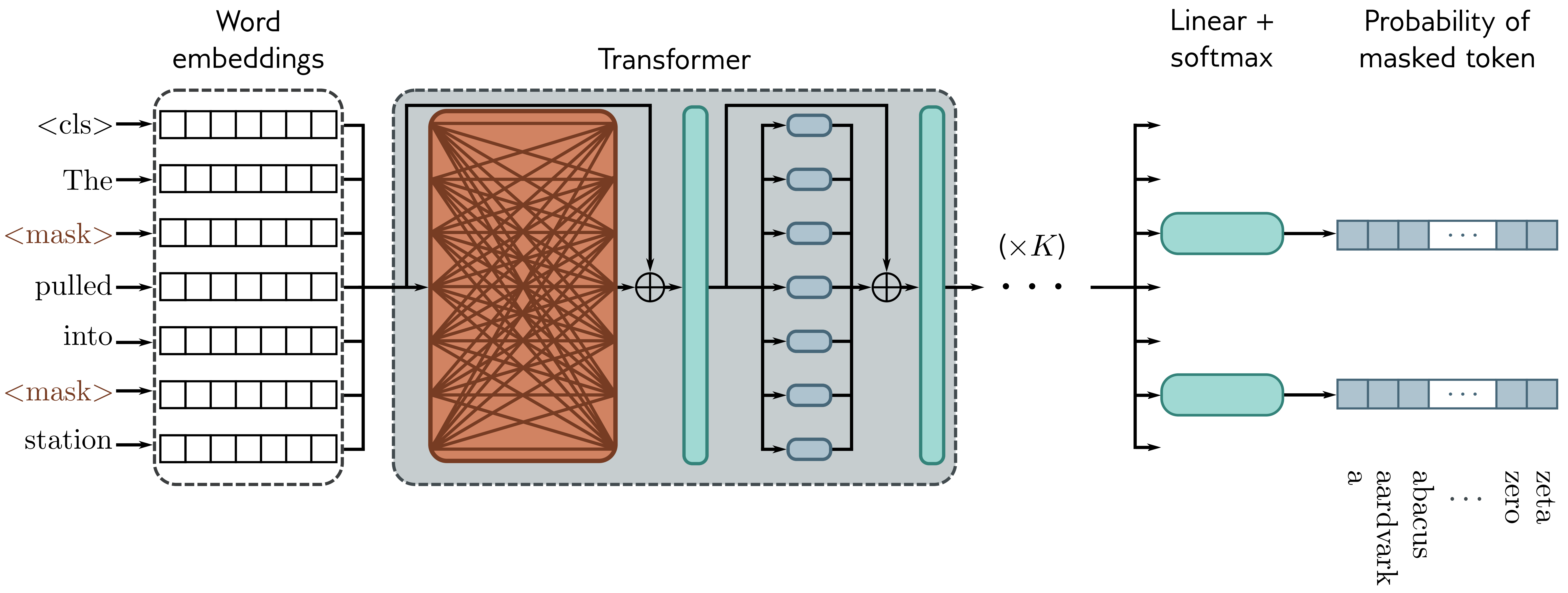
Transformers are trained in two phases:

1. **Semi-supervised pre-training:** Using a very large dataset, train the network to perform task for which dataset implies the answer
   - we **need not label** the examples manually
   - e.g., predicting masked words

Question: Is this important? Why?

2. **Fully supervised fine-tuning:**
   Add another layer or two at the end, and train for the real task using manually labelled examples
   - e.g., sentiment analysis, word classification, text span prediction

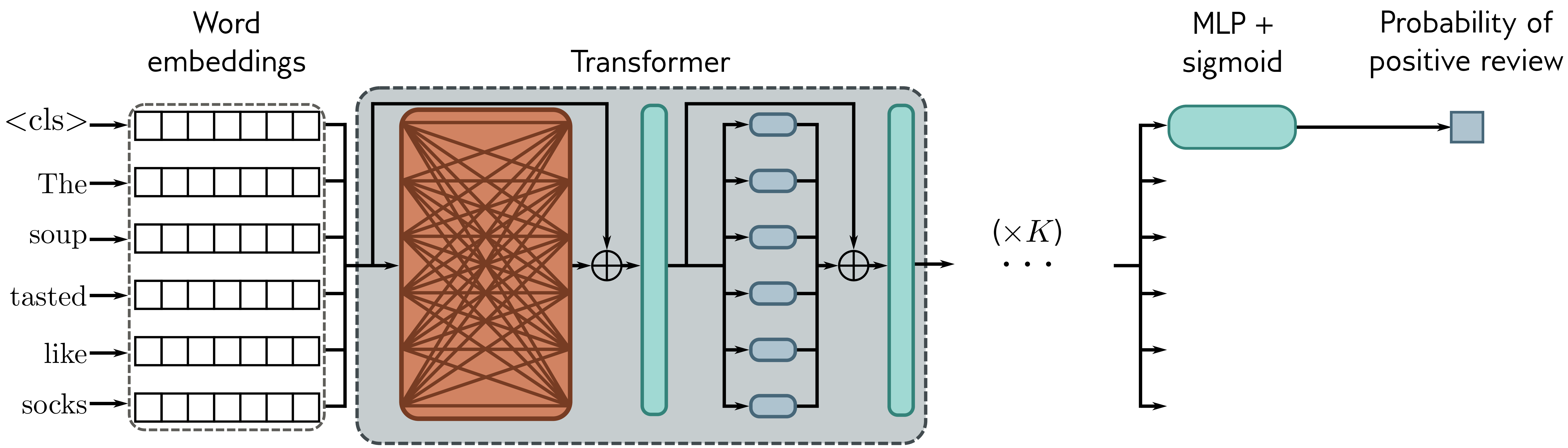
Pre-training



- Subset of tokens in an example sequence are **masked** (replaced with a special token)
- Neural network applied to each masked output predicts probabilities for missing token
    - Loss is back-propagated through entire network
- At end of training, that neural network is **thrown away**

(Image: Prince 2022)

Fine-tuning: BERT for Sentiment classification



- **Sentiment classification:** Predict if a sequence is positive or negative
- First token of sequence is always a special "classify" token
- Neural network trained on corresponding output token using labelled dataset

(Image: Prince 2022)

Summary

- Naïvely representing **sequential inputs** for a neural network requires infeasibly many input nodes (and hence **parameters**)
- One-hot encodings are wastefully large and have no semantic structure
  - **Embeddings** solve these problems and can be trained without explicit labels
- **Recurrent neural networks** are a **specialized architecture** for sequential inputs
  - **State** accumulates across input elements
  - Each stage computed from **previous stage** using **same parameters**
- **Transformers** are another specialized architecture!
  - **Self-attention** to combine inputs instead of accumulating state
  - All states output (not just last in a sequence)
  - Improved ability to attend to long-range dependence
  - Admits of better parallel evaluation
  - Pre-training followed by fine-tuning
  - Classification: attend to the output corresponding to a special "classify" input token