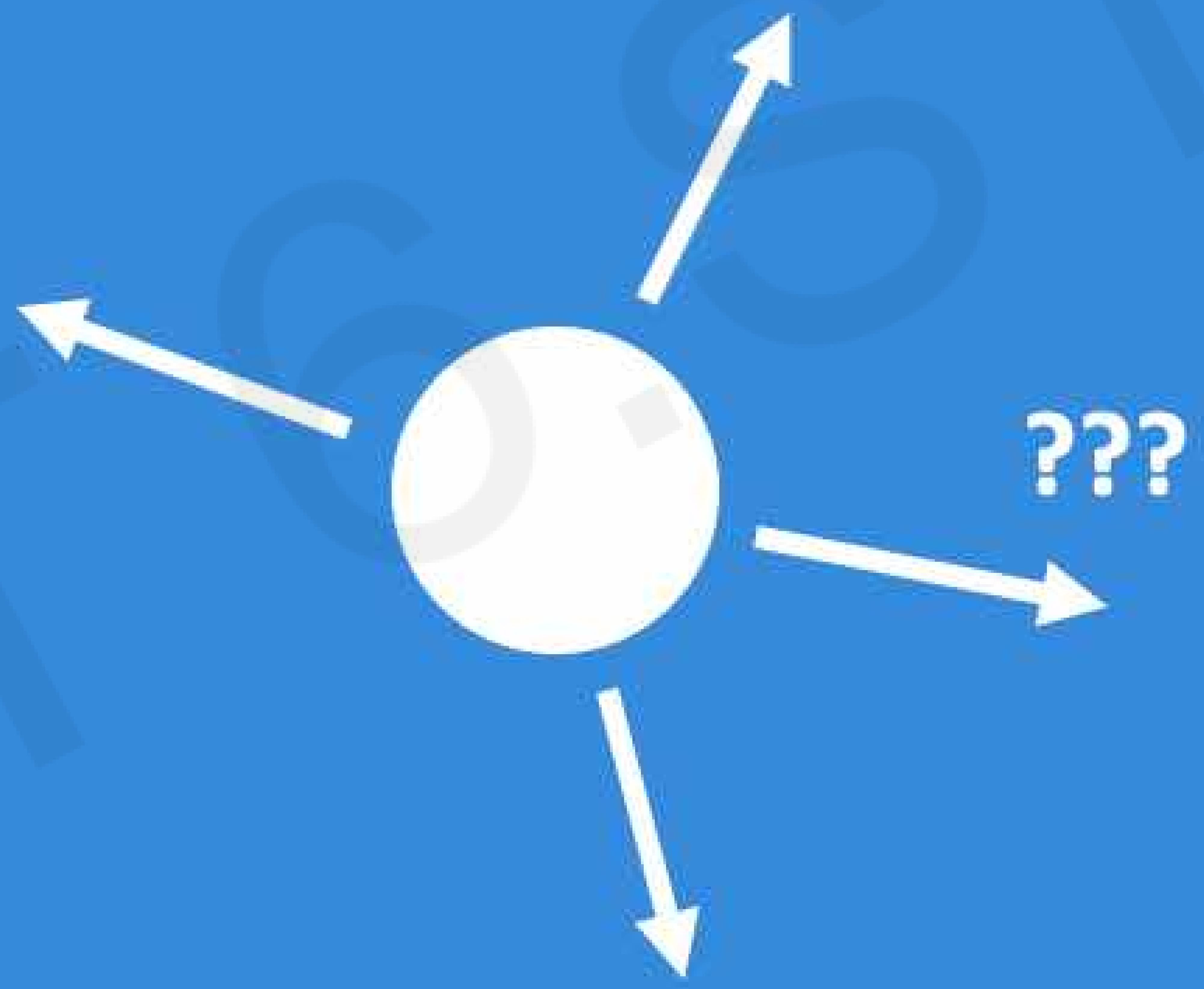
# Deep Sequence Modeling

Ava Soleimany
MIT 6.S191

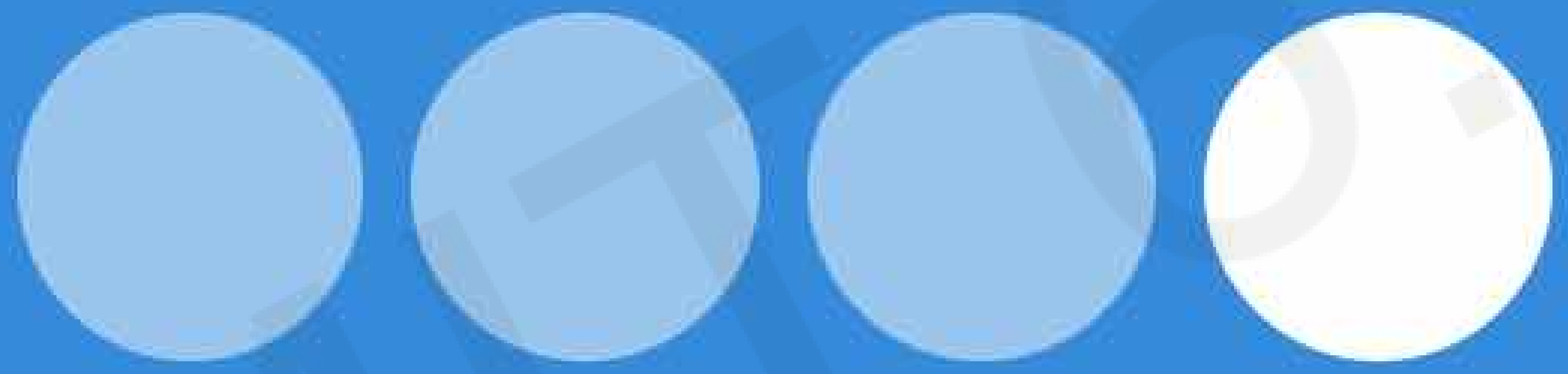
January 24, 2022

Given an image of a ball,
can you predict where it will go next?

# Given an image of a ball, can you predict where it will go next?



Given an image of a ball,
can you predict where it will go next?



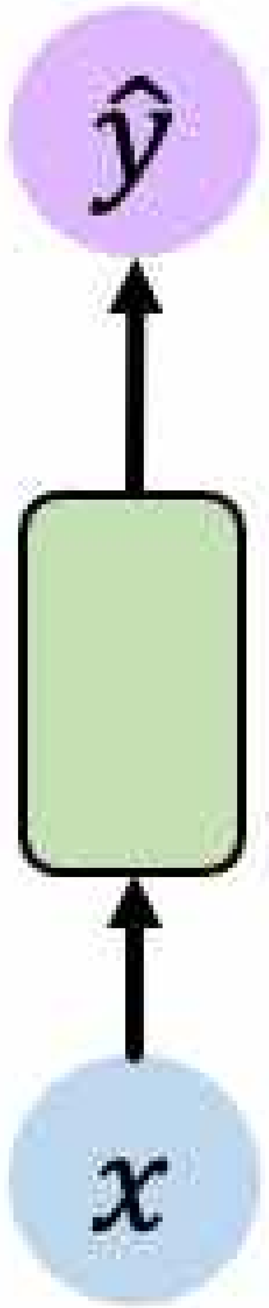
Given an image of a ball,
can you predict where it will go next?

# Sequences in the Wild

Audio

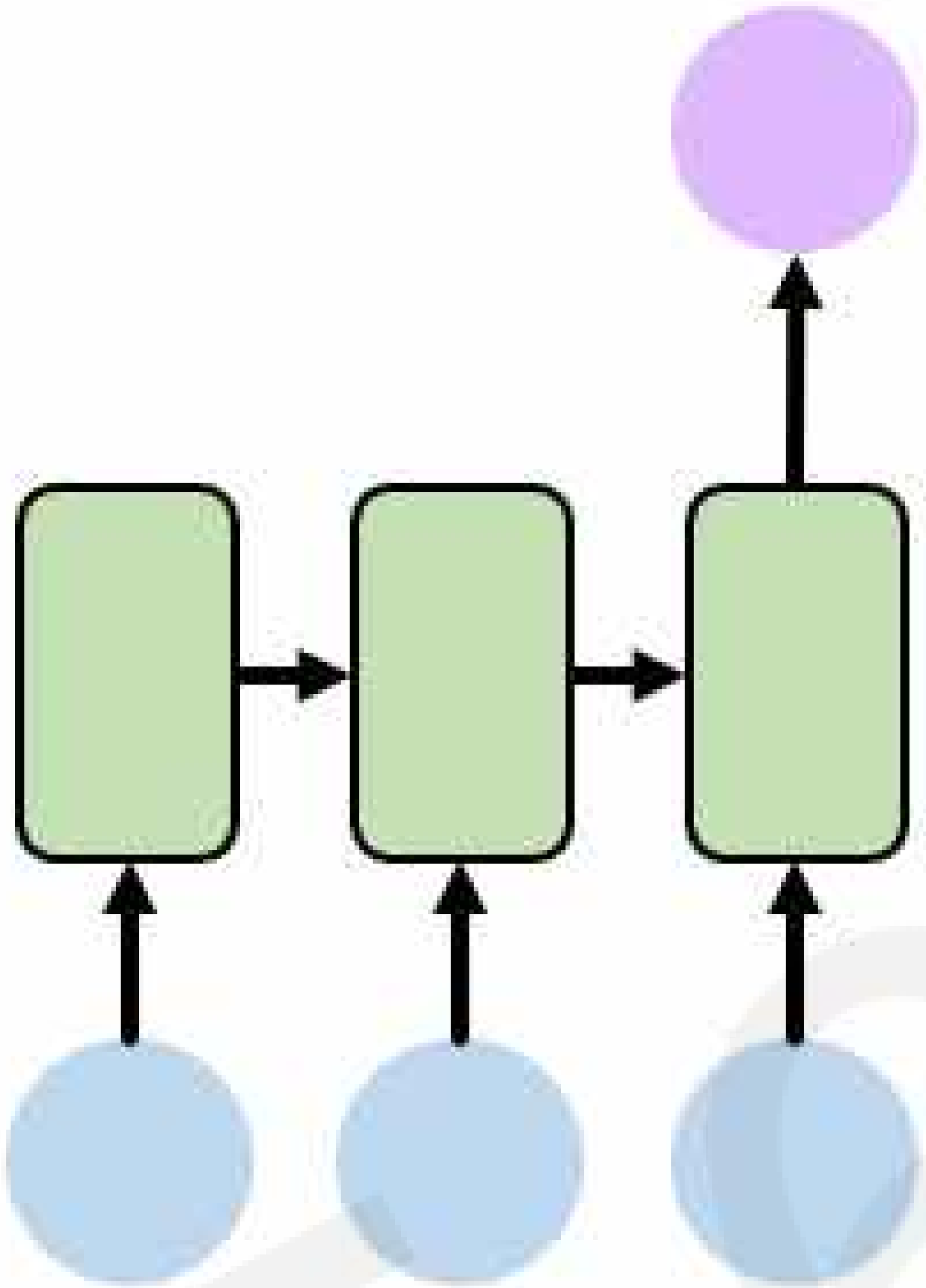
# Sequences in the Wild

# Sequence Modeling Applications

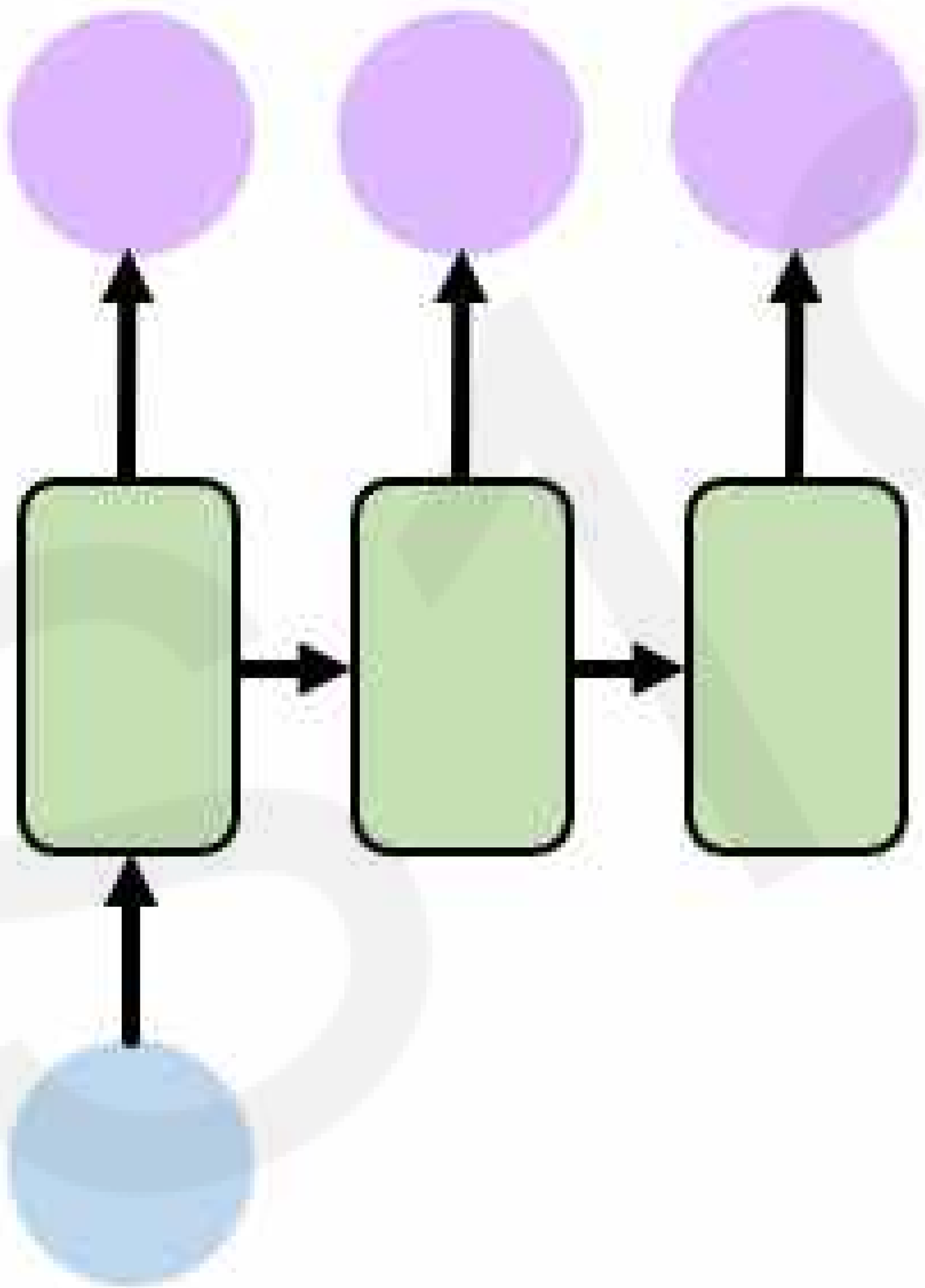



One to One
**Binary Classification**

"Will I pass this class?"
Student → Pass?

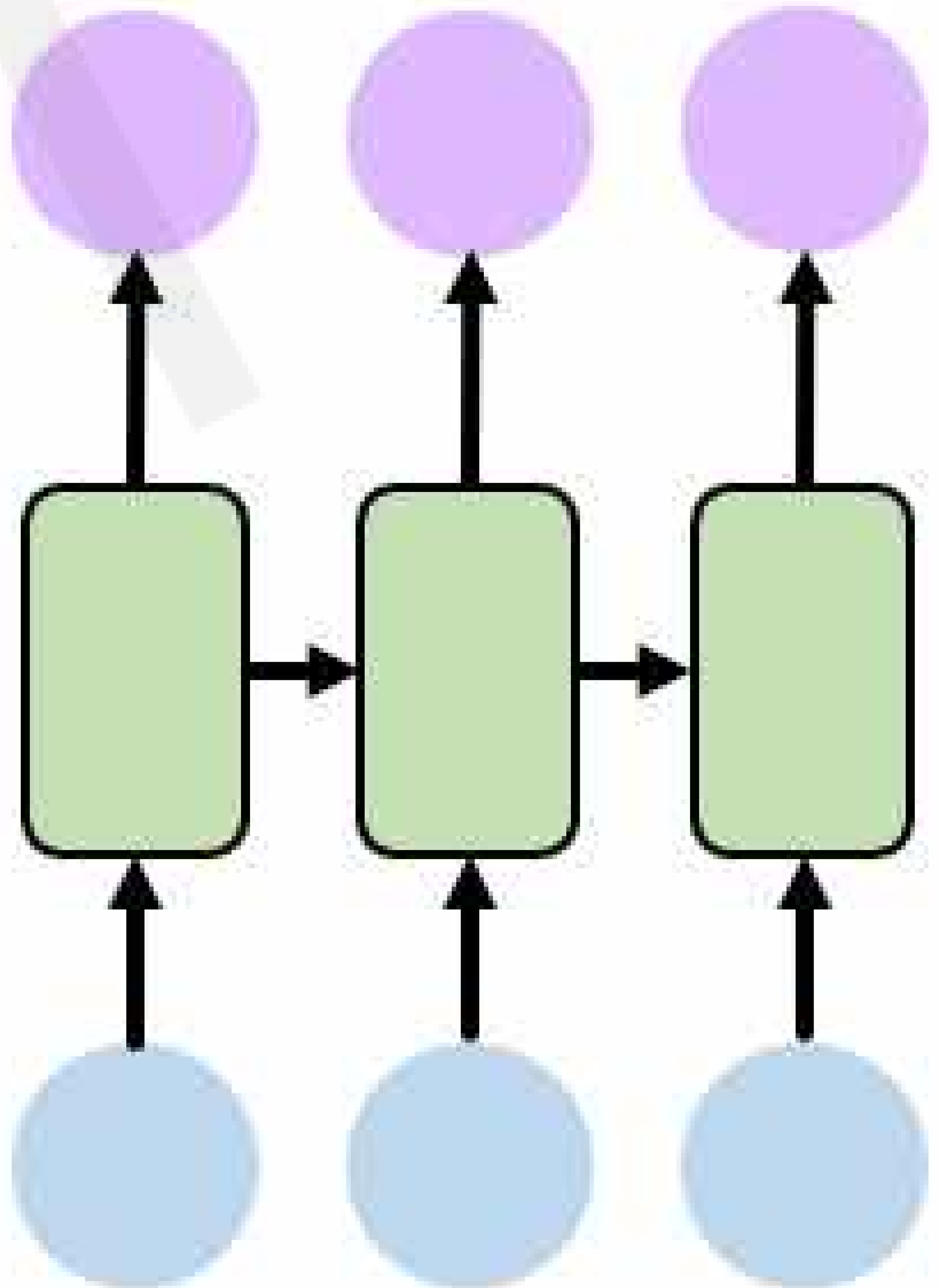


Many to One
**Sentiment Classification**

One to Many
**Image Captioning**

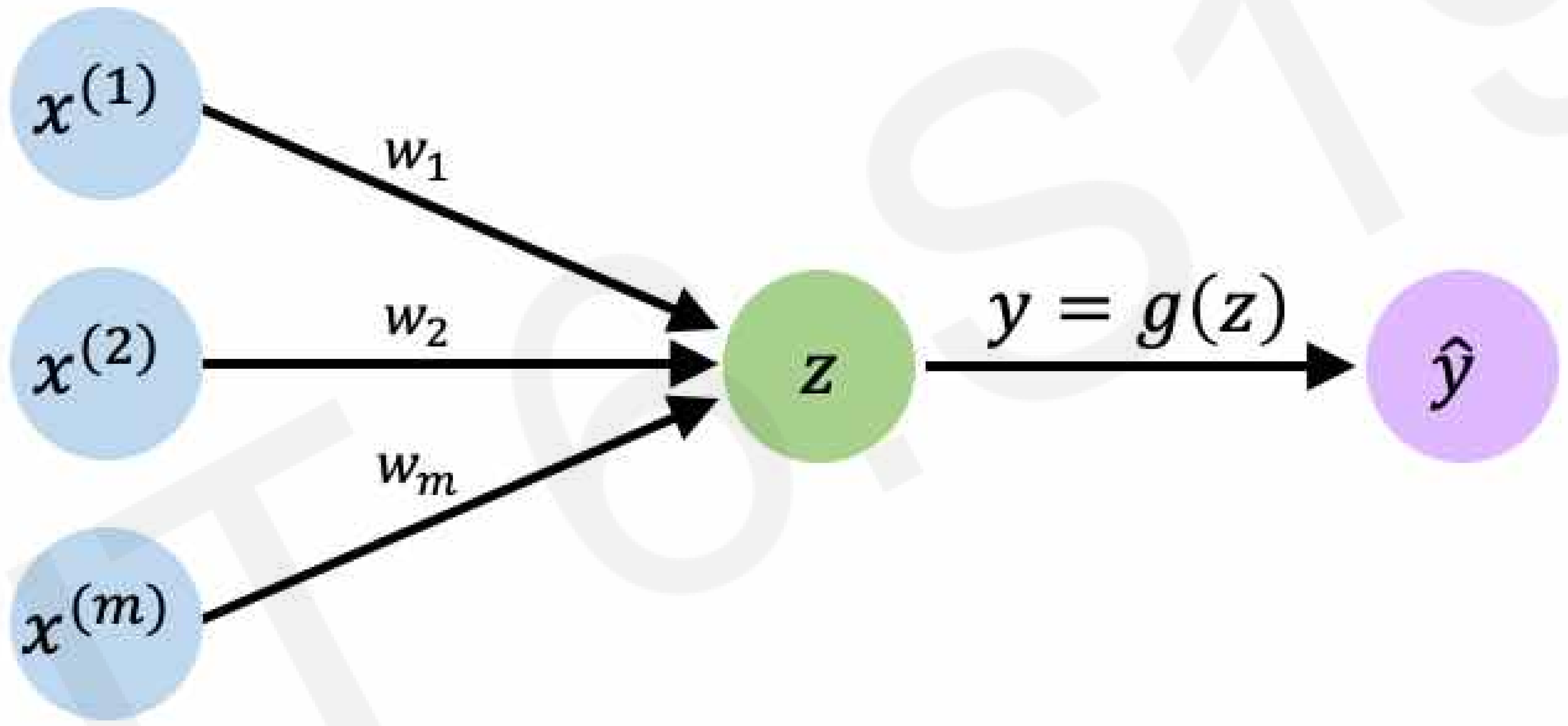
"A baseball player throws a ball."



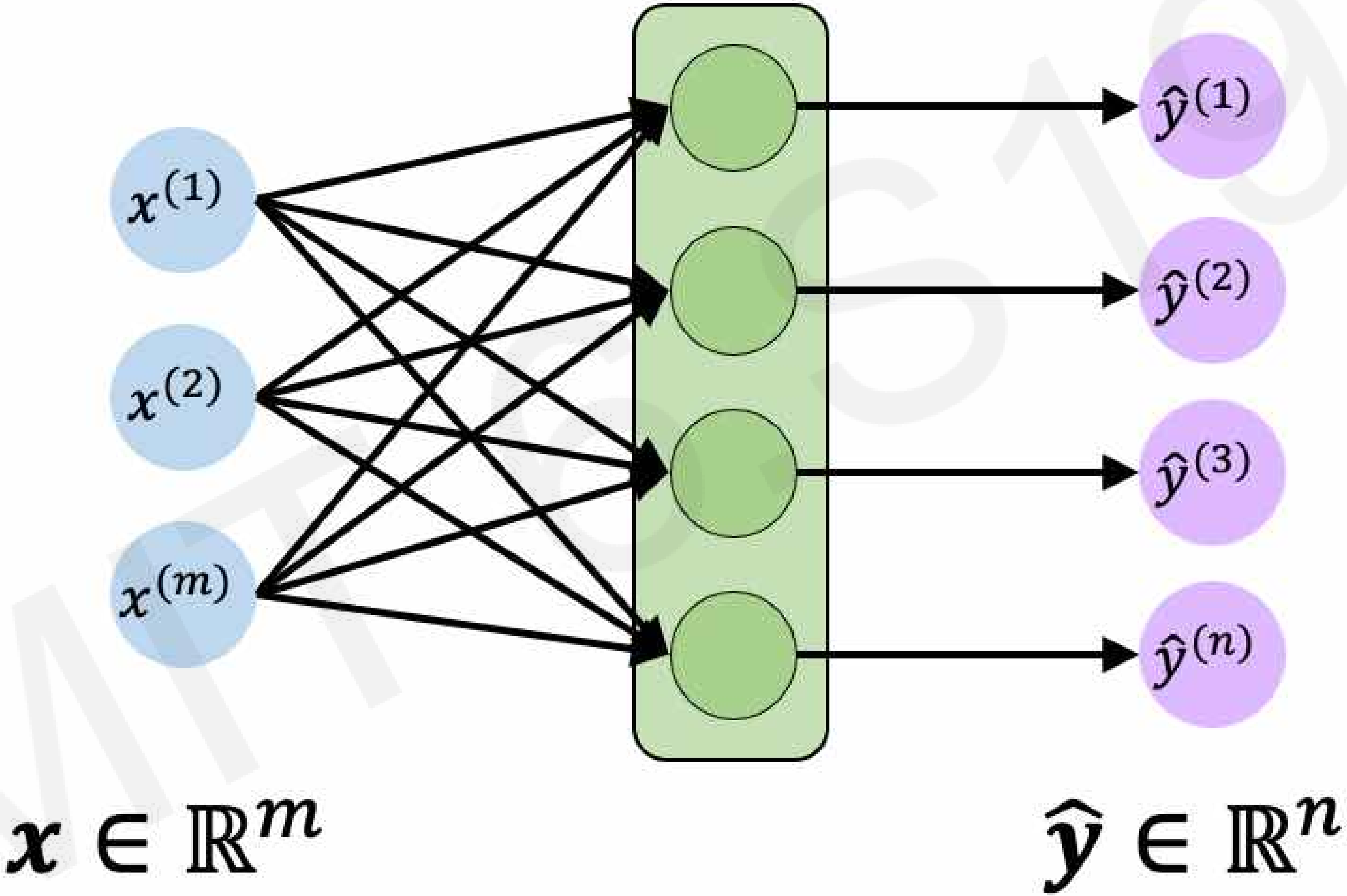
Many to Many
**Machine Translation**

# Neurons with Recurrence

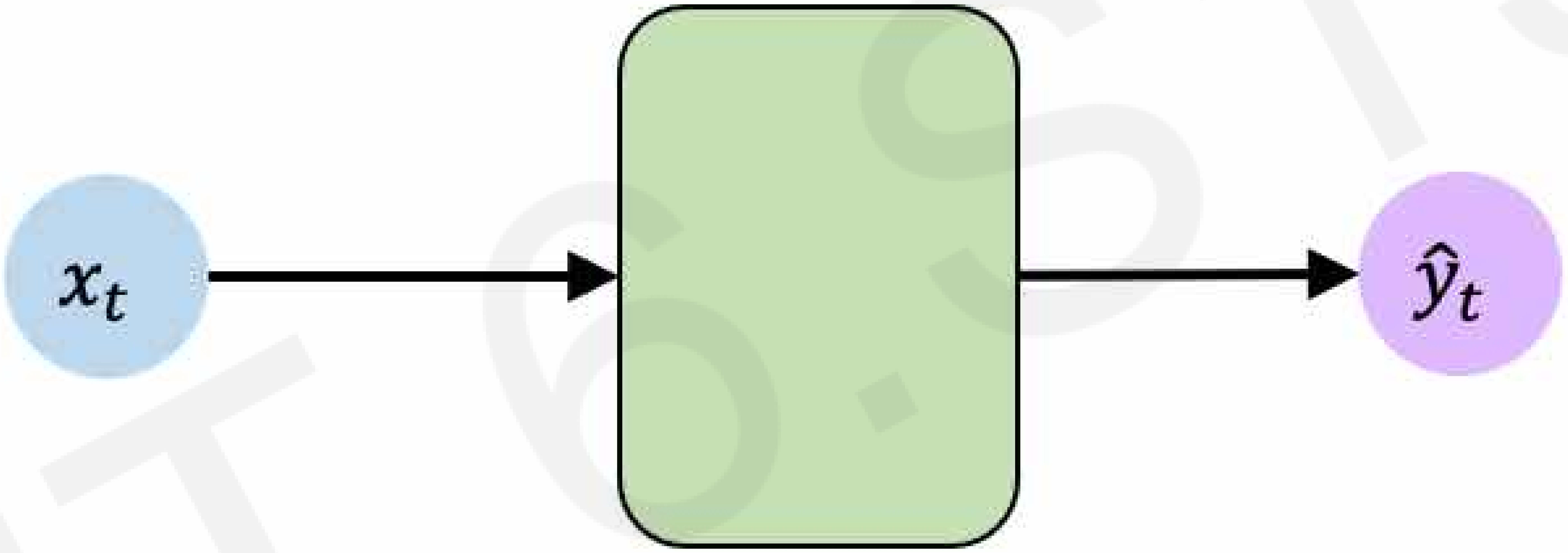
# The Perceptron Revisited



# Feed-Forward Networks Revisited

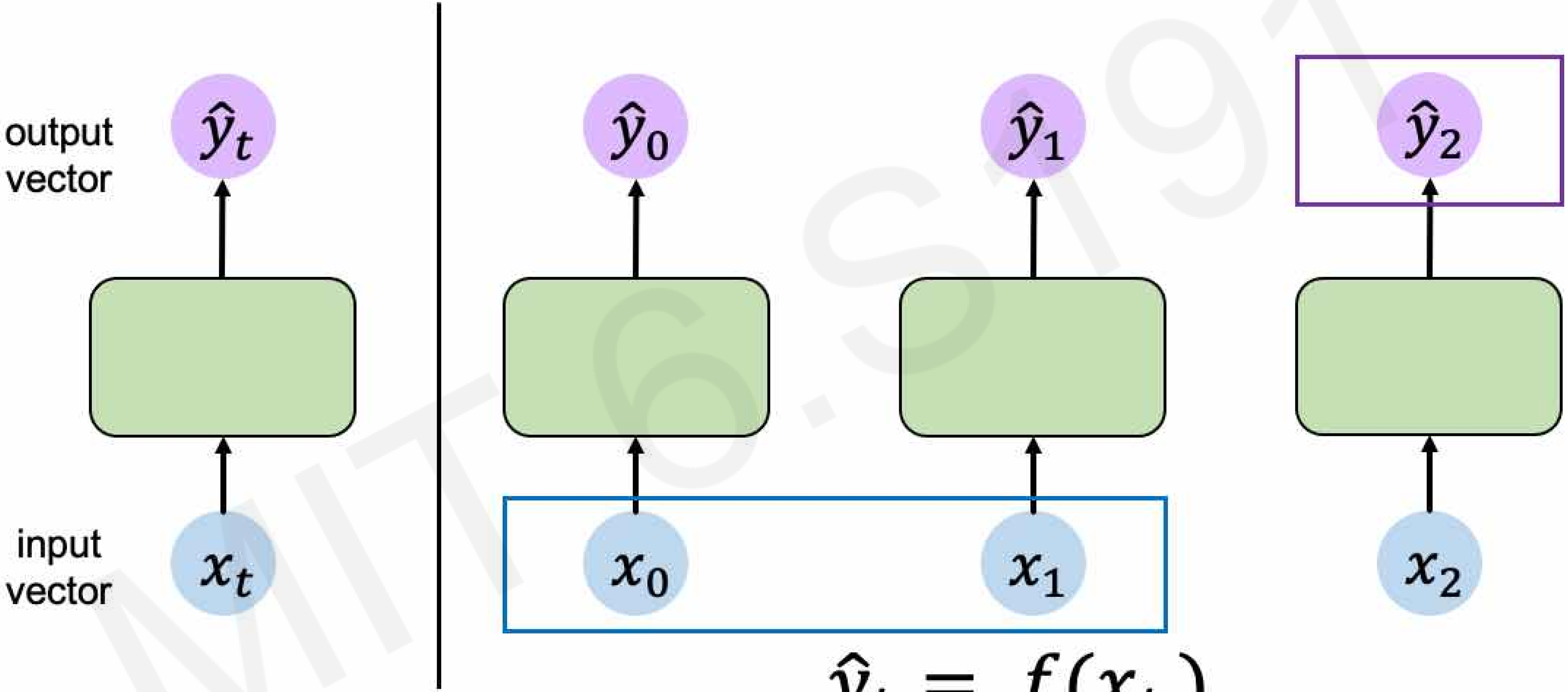


# Feed-Forward Networks Revisited



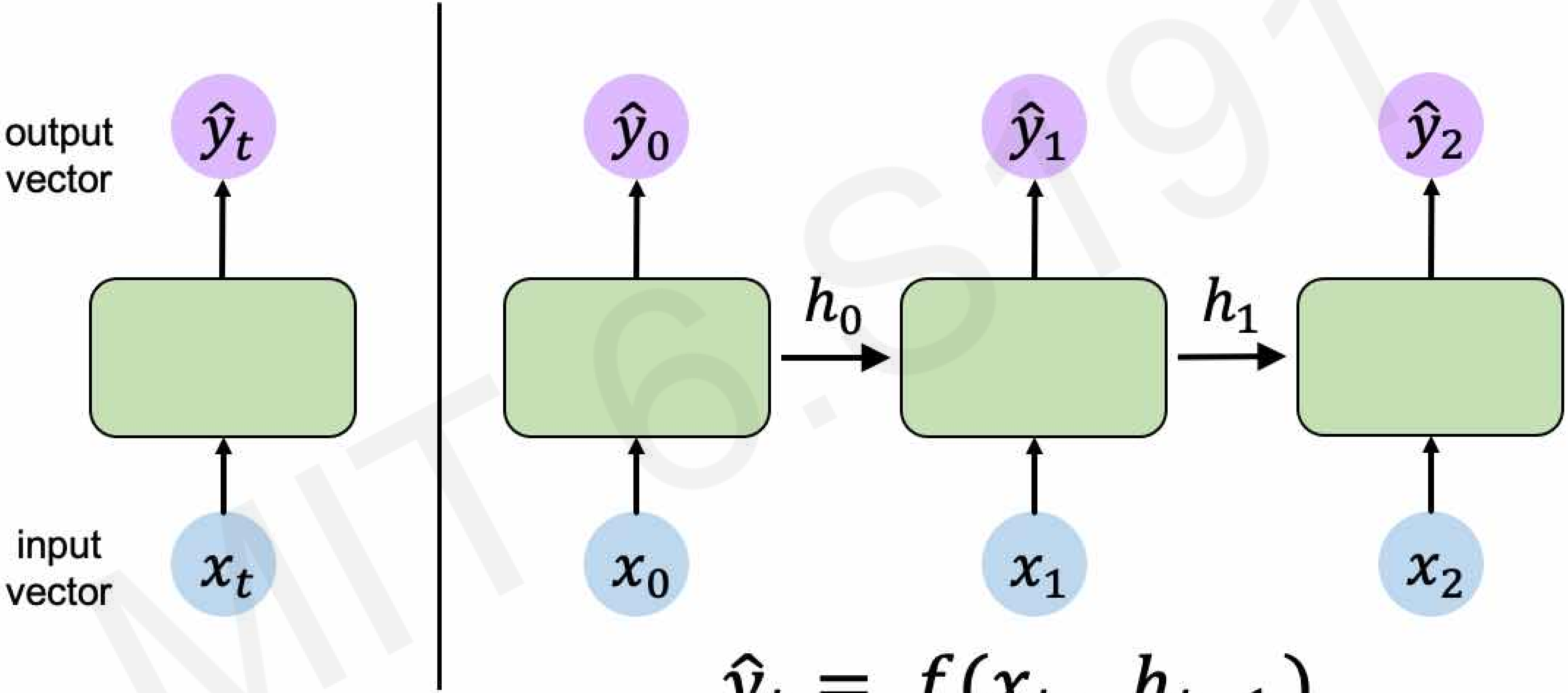

$$\boldsymbol{x_t} \in \mathbb{R}^m \qquad \boldsymbol{\hat{y}_t} \in \mathbb{R}^n$$

# Handling Individual Time Steps



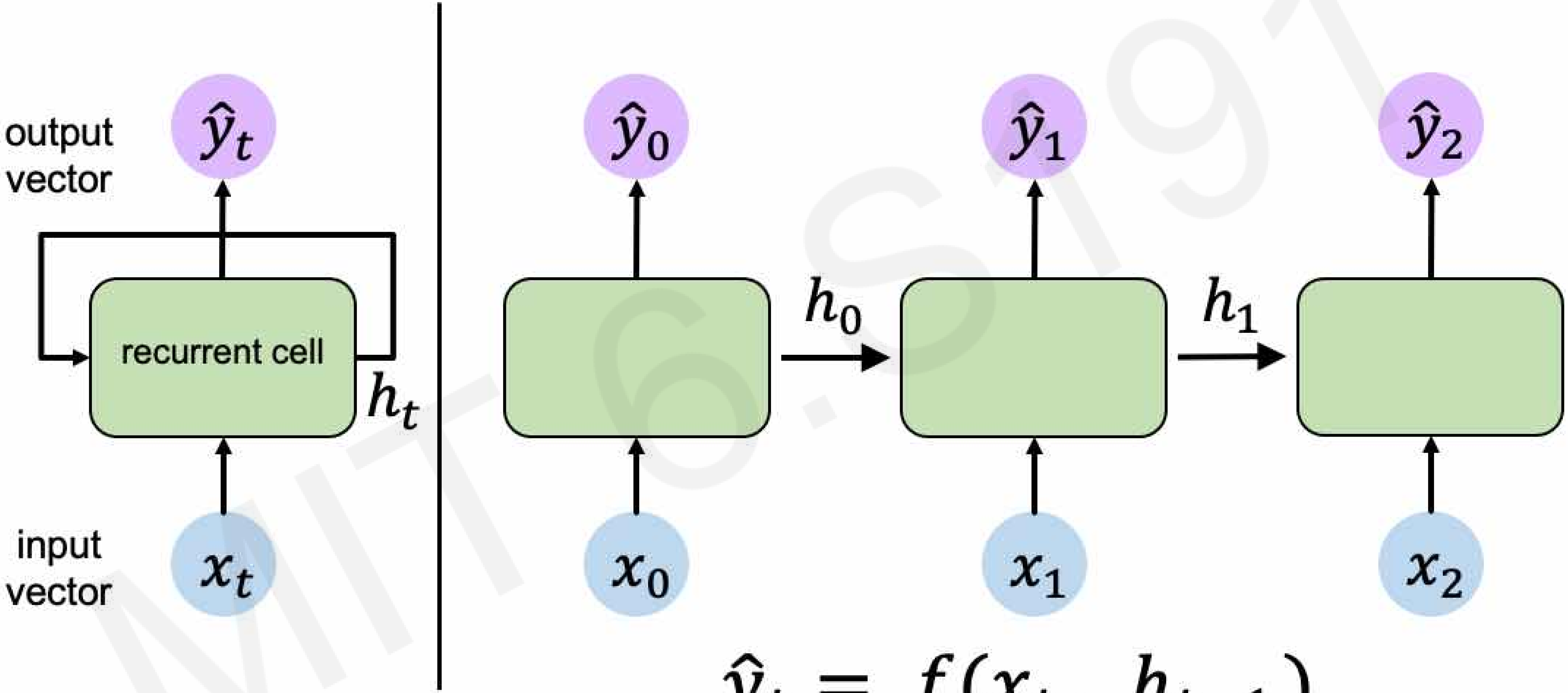

$$\hat{y}_t = f(x_t)$$

# Neurons with Recurrence




$$\underbrace{\hat{y}_t}_{\text{output}} = f(\underbrace{x_t}_{\text{input}},\ \underbrace{h_{t-1}}_{\text{past memory}})$$

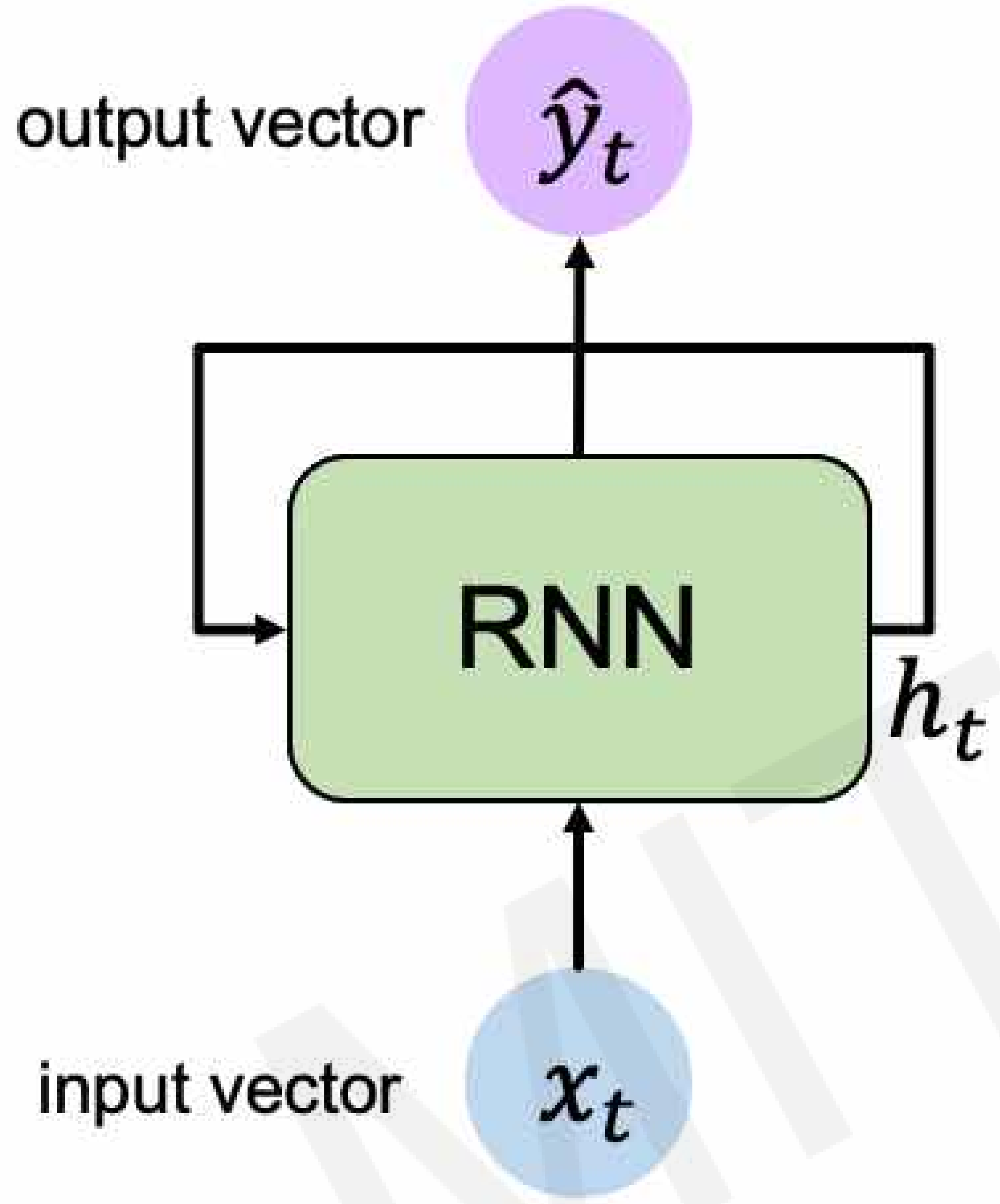
# Neurons with Recurrence




$$\underbrace{\hat{y}_t}_{\text{output}} = f(\underbrace{x_t}_{\text{input}},\ \underbrace{h_{t-1}}_{\text{past memory}})$$

# Recurrent Neural Networks (RNNs)

# Recurrent Neural Networks (RNNs)




Apply a **recurrence relation** at every time step to process a sequence:

$$\boxed{h_t} = \boxed{f_W}(\boxed{x_t}, \boxed{h_{t-1}})$$

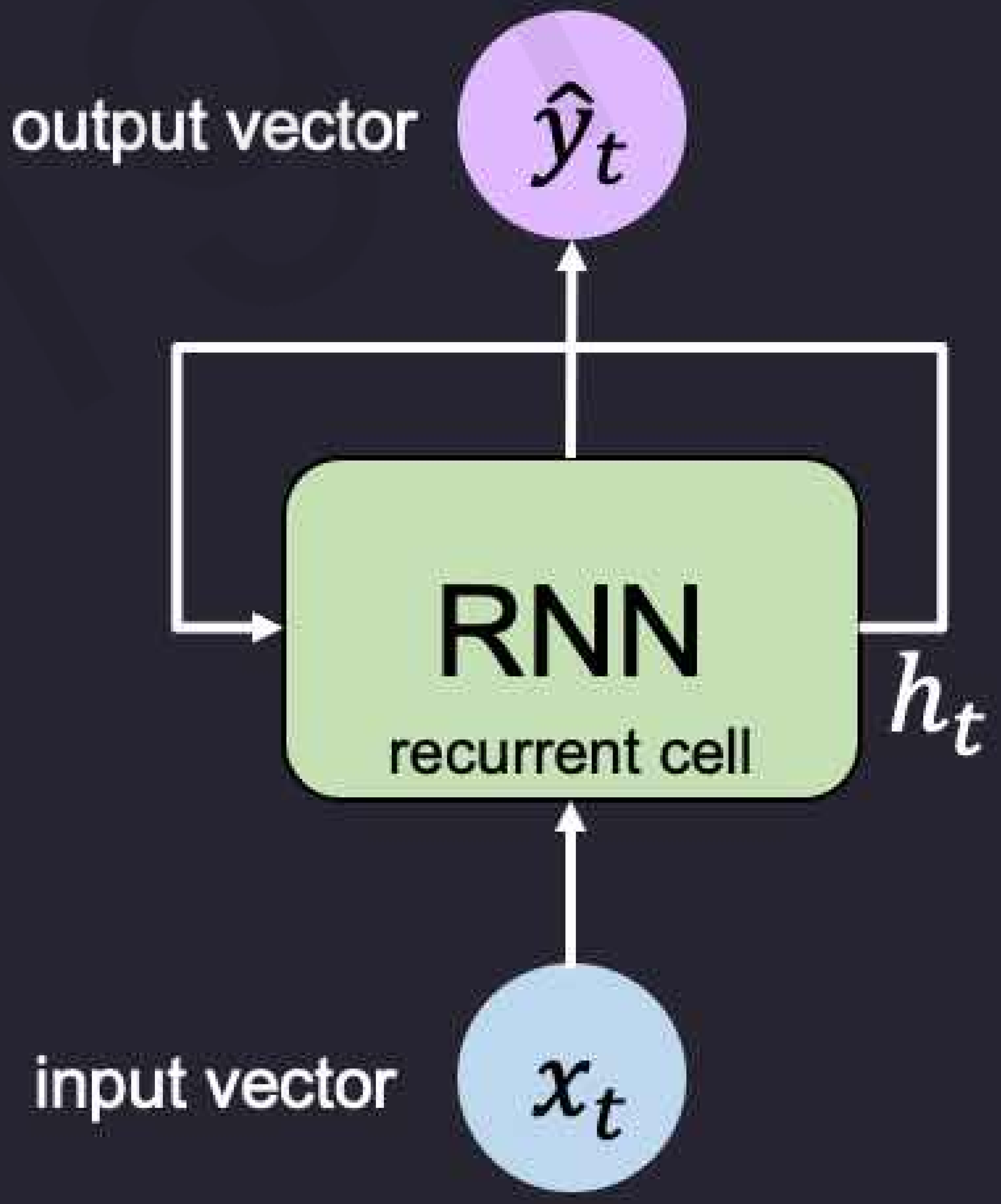
cell state | function with weights W | input | old state

**Note: the same function and set of parameters are used at every time step**

RNNs have a **state**, $h_t$, that is updated **at each time step** as a sequence is processed

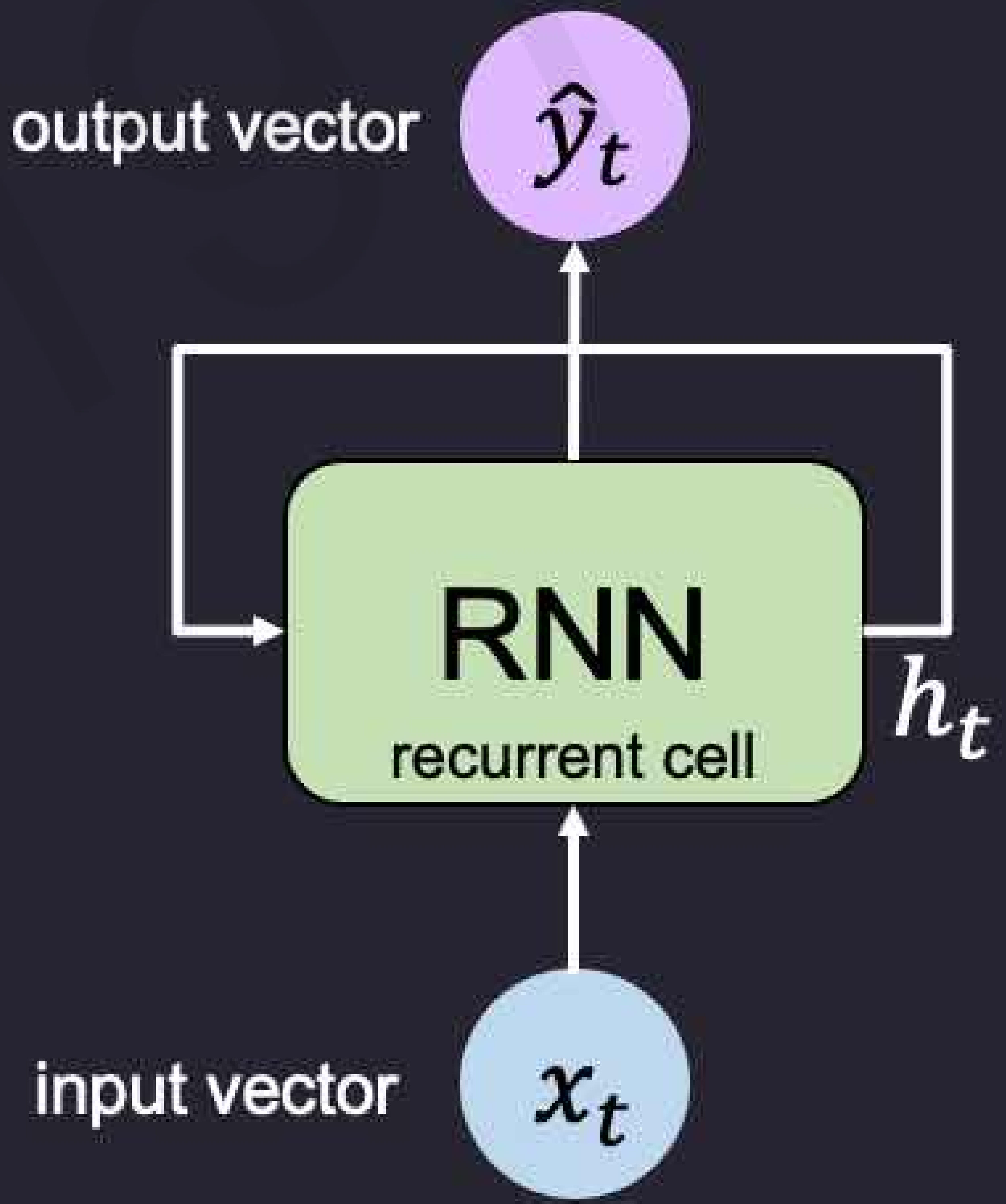
# RNN Intuition

```
my_rnn = RNN()
hidden_state = [0, 0, 0, 0]

sentence = ["I", "love", "recurrent", "neural"]

for word in sentence:
    prediction, hidden_state = my_rnn(word, hidden_state)

next_word_prediction = prediction
# >>> "networks!"
```



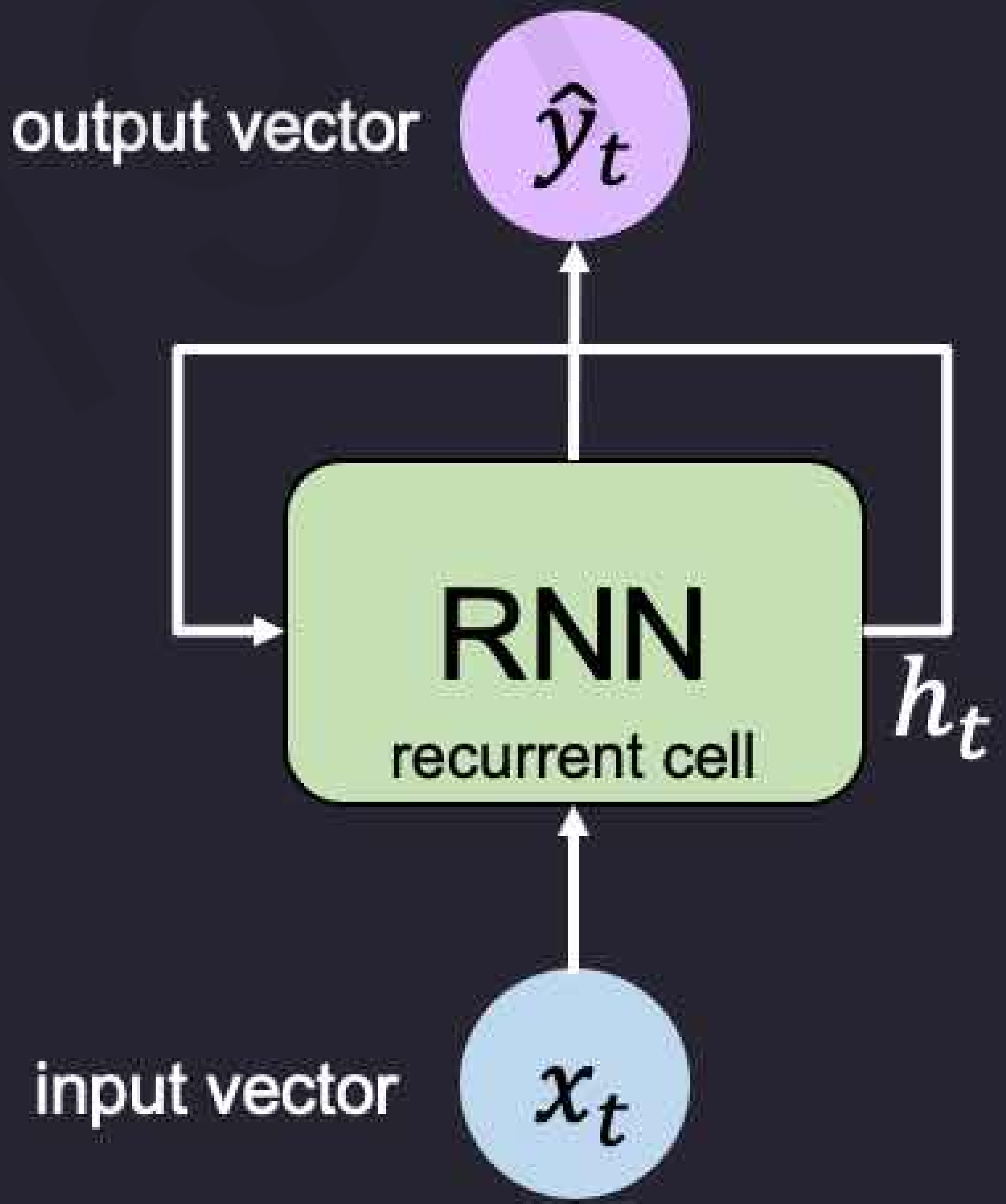
# RNN Intuition

```
my_rnn = RNN()
hidden_state = [0, 0, 0, 0]

sentence = ["I", "love", "recurrent", "neural"]

for word in sentence:
    prediction, hidden_state = my_rnn(word, hidden_state)

next_word_prediction = prediction
# >>> "networks!"
```

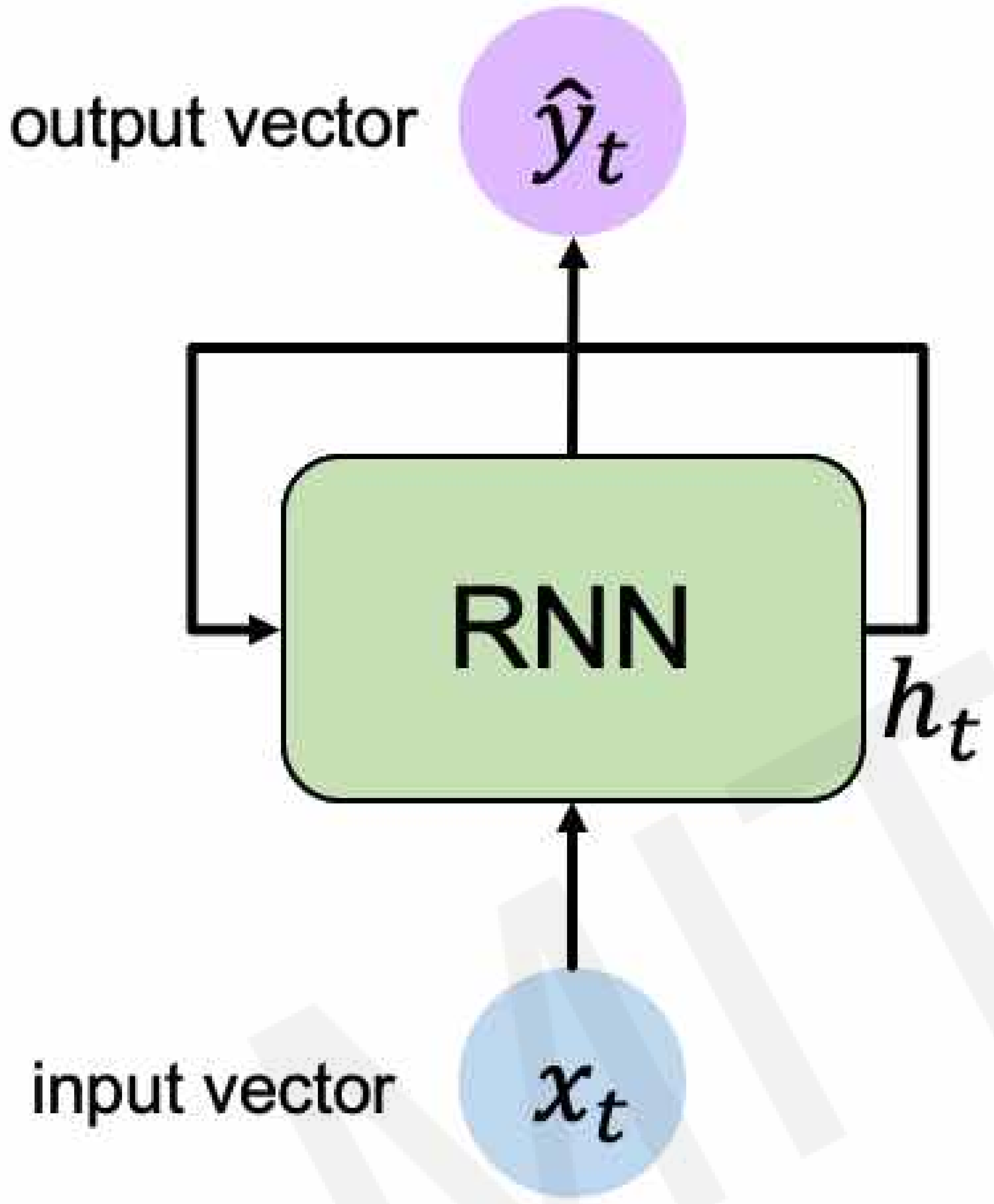


# RNN Intuition

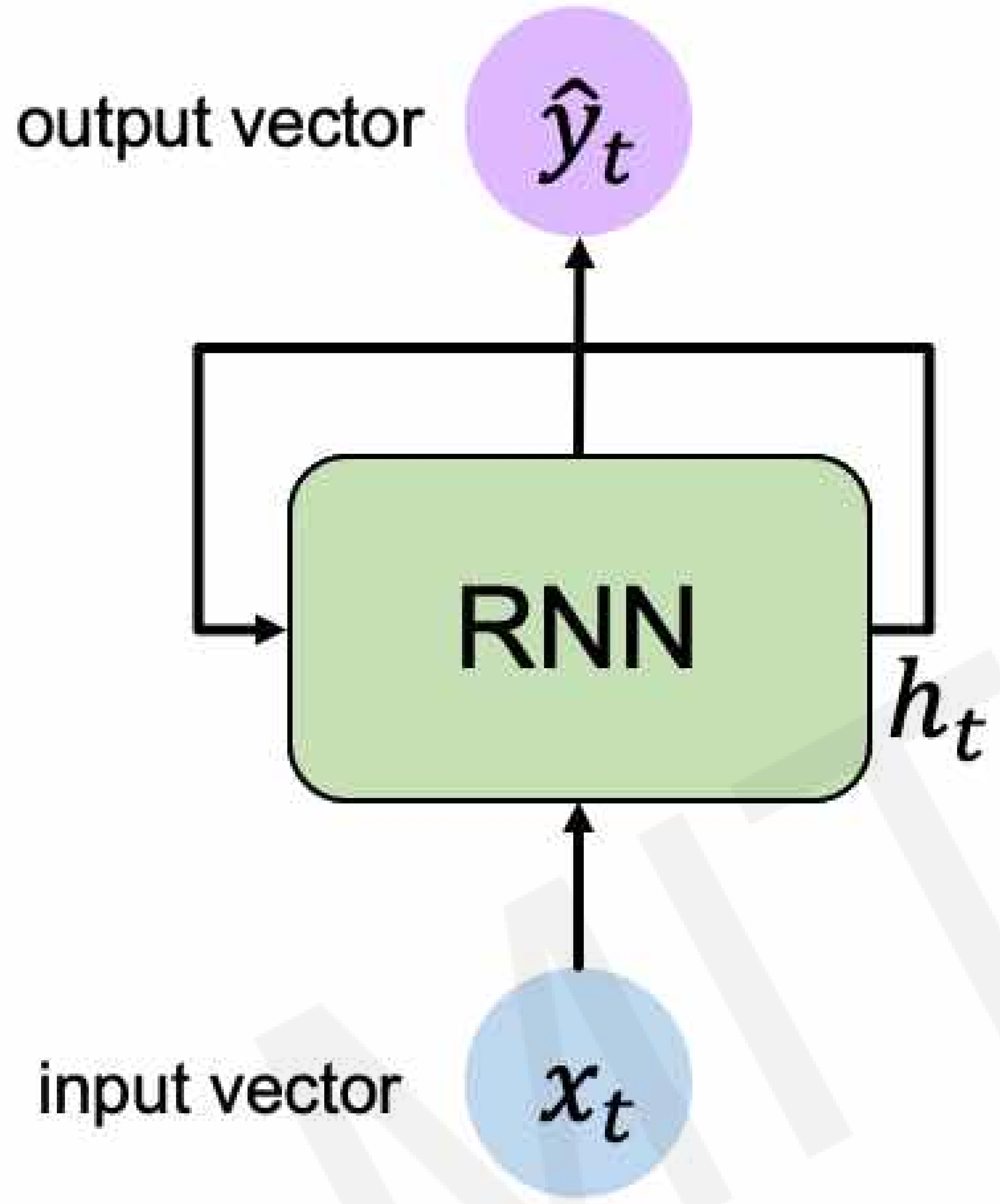
```
my_rnn = RNN()
hidden_state = [0, 0, 0, 0]

sentence = ["I", "love", "recurrent", "neural"]

for word in sentence:
    prediction, hidden_state = my_rnn(word, hidden_state)

next_word_prediction = prediction
# >>> "networks!"
```



# RNN State Update and Output



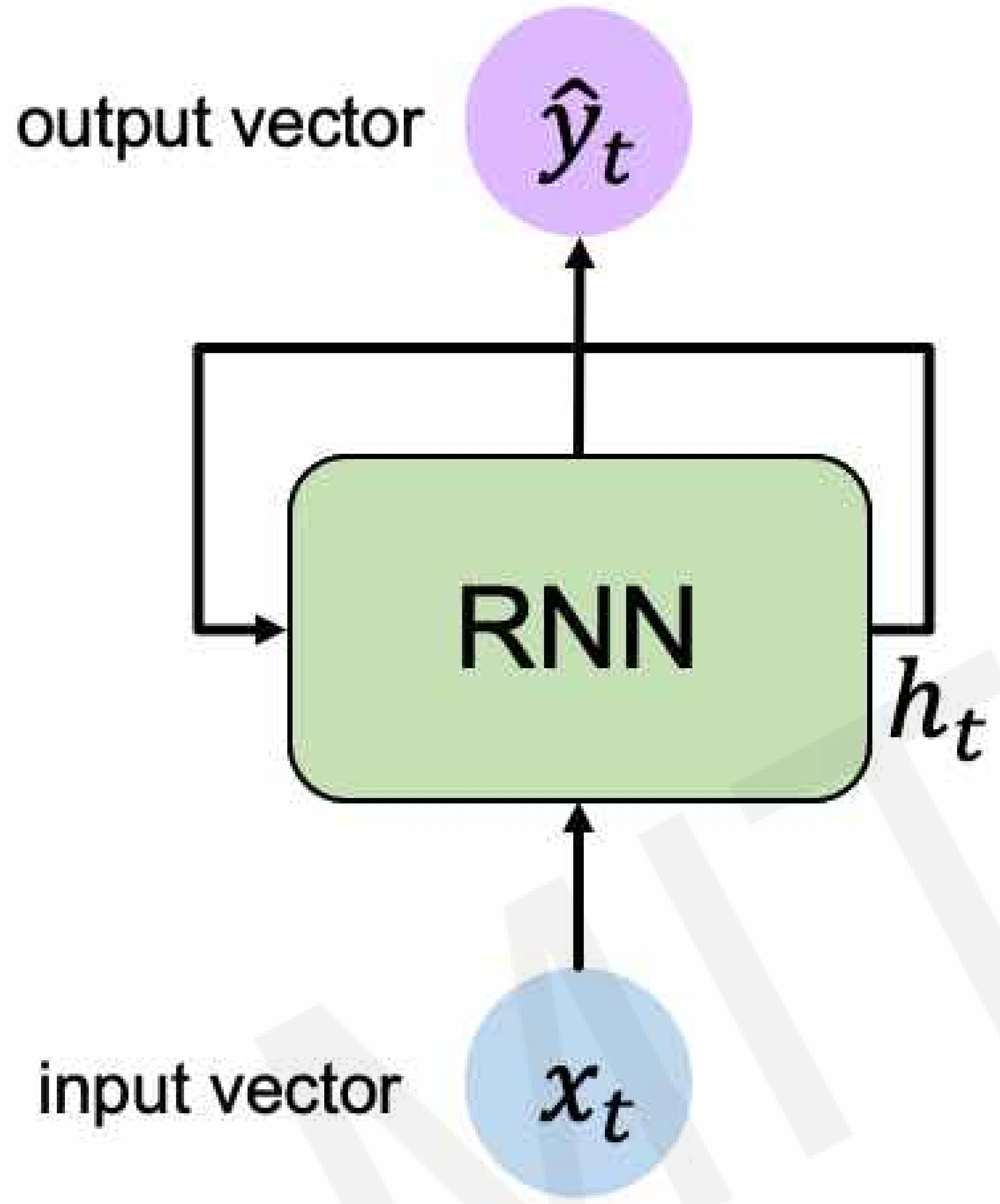
# RNN State Update and Output




**Input Vector**

$x_t$

# RNN State Update and Output



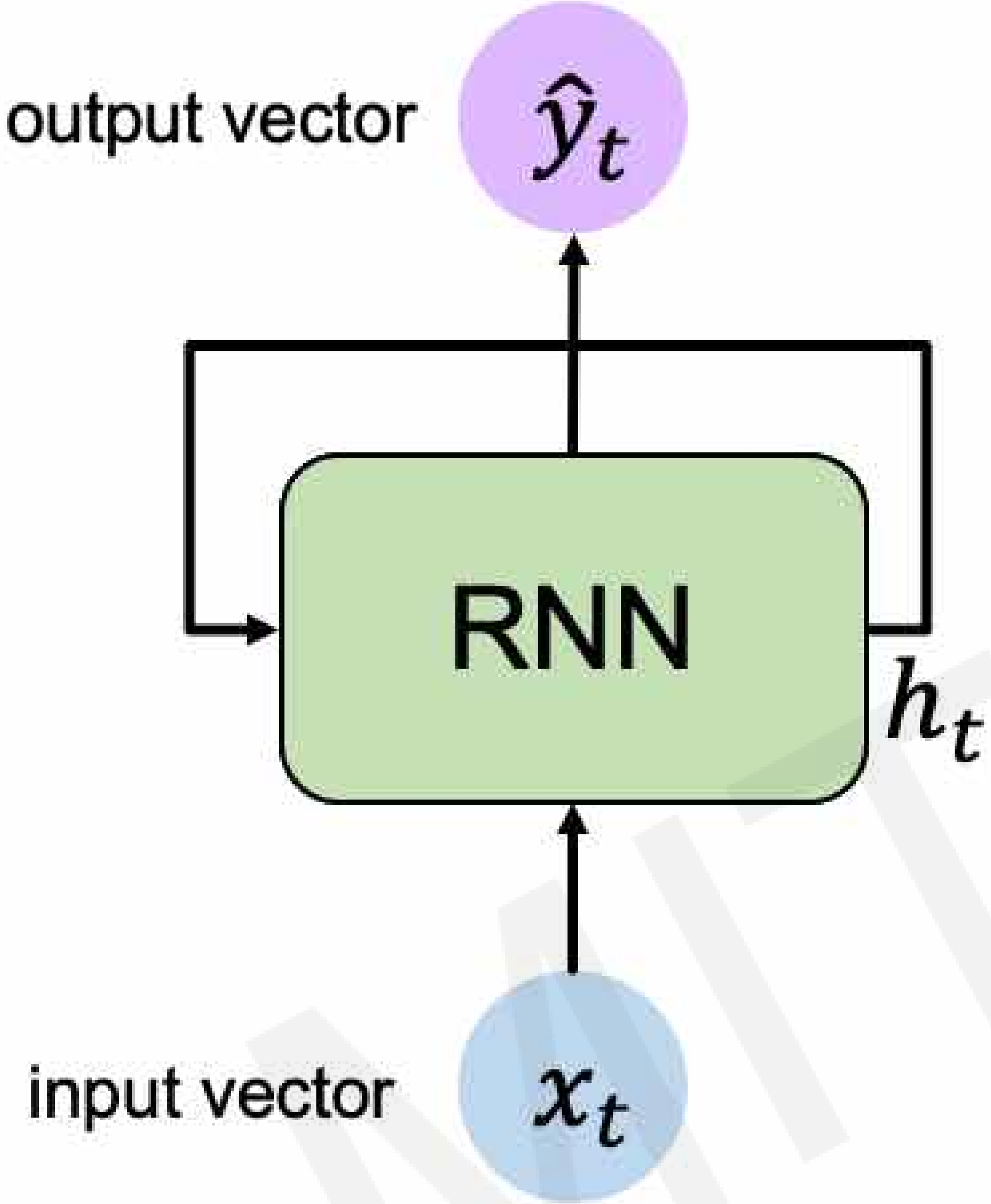

## Update Hidden State

$$h_t = \tanh(\boldsymbol{W}_{hh}^{T} h_{t-1} + \boldsymbol{W}_{xh}^{T} x_t)$$

## Input Vector

$$x_t$$

# RNN State Update and Output




**Output Vector**

$$\hat{y}_t = \boldsymbol{W}_{hy}^T h_t$$

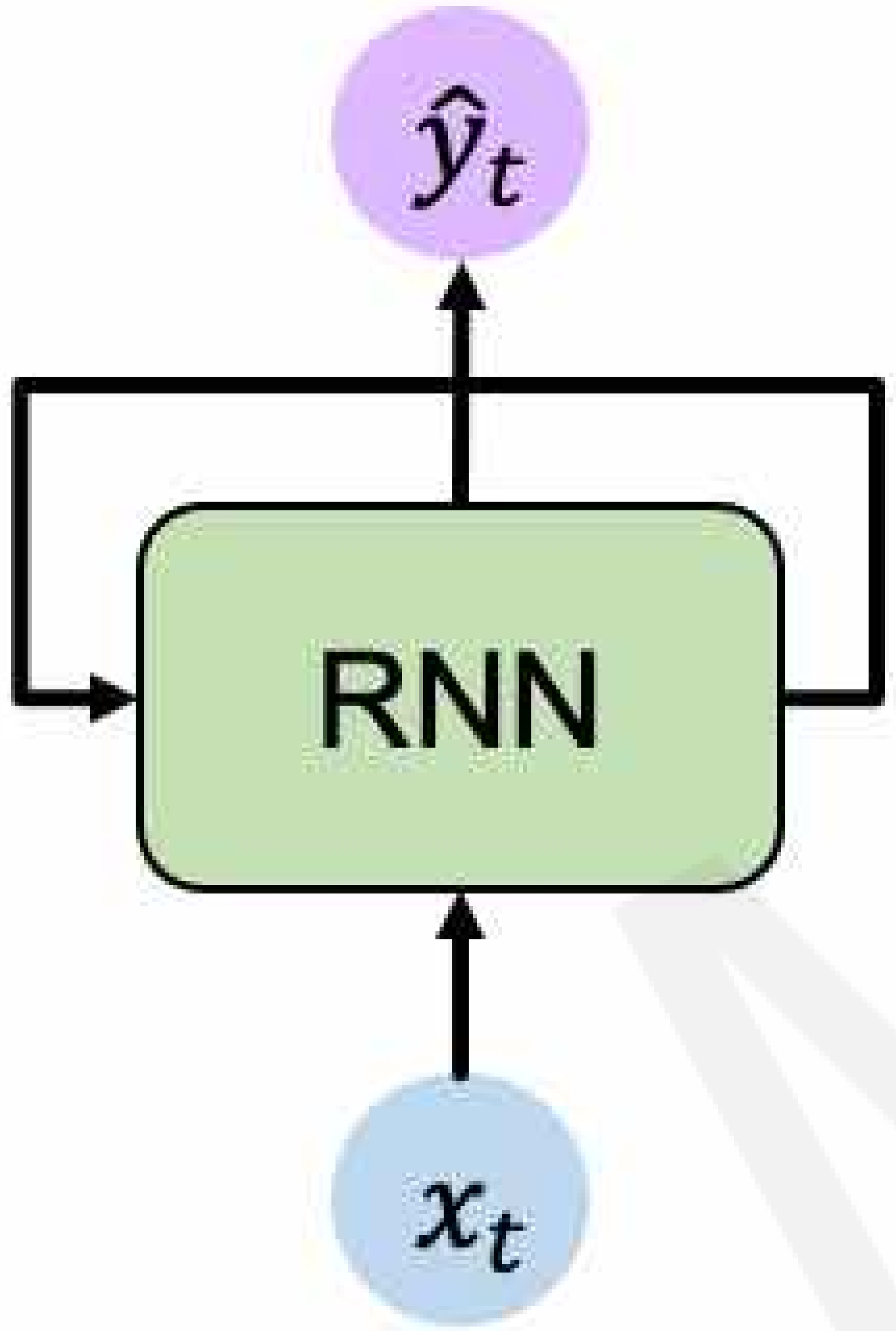
**Update Hidden State**

$$h_t = \tanh(\boldsymbol{W}_{hh}^T h_{t-1} + \boldsymbol{W}_{xh}^T x_t)$$

**Input Vector**

$$x_t$$

# RNNs: Computational Graph Across Time



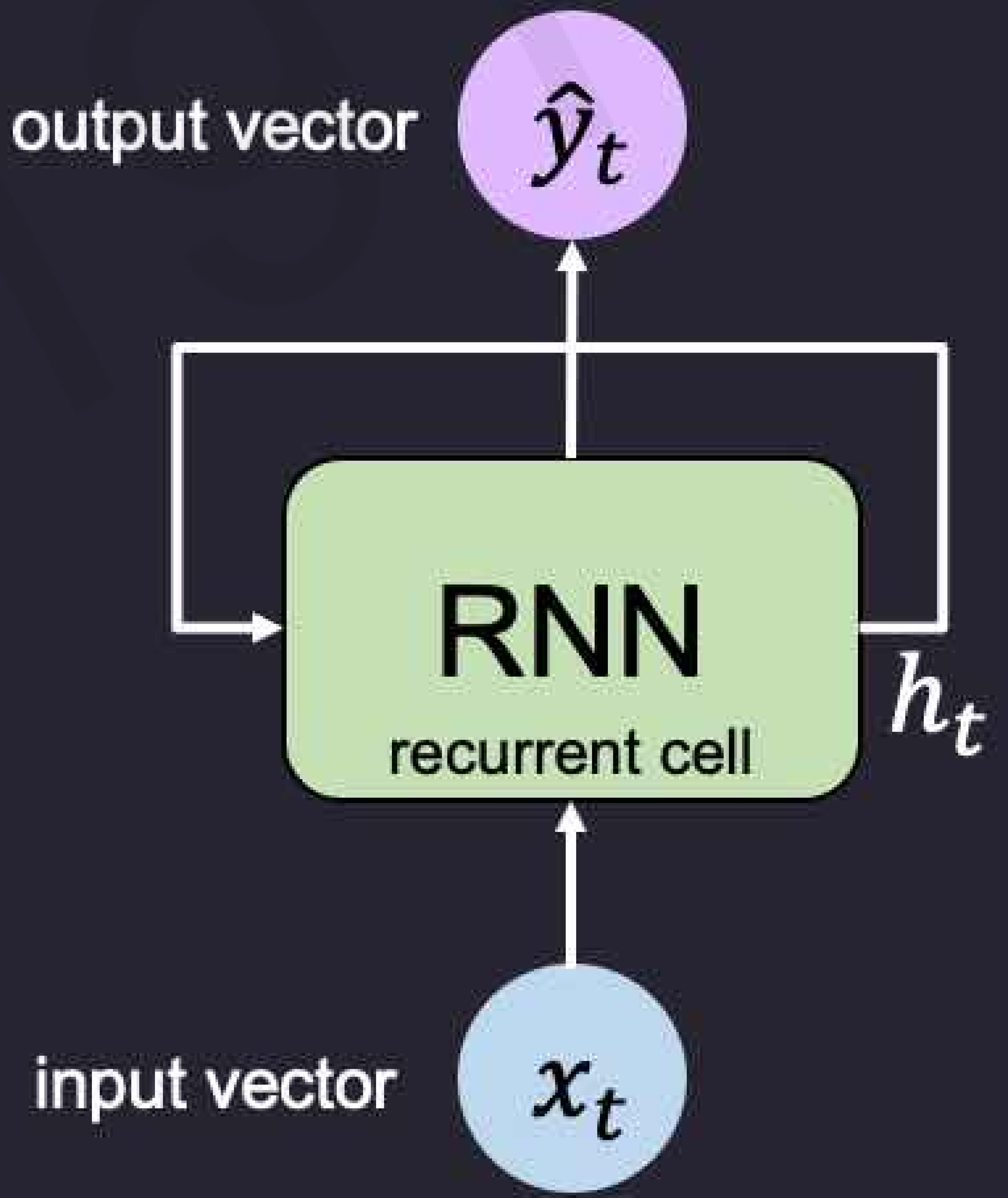

=

Represent as computational graph unrolled across time

# RNNs: Computational Graph Across Time

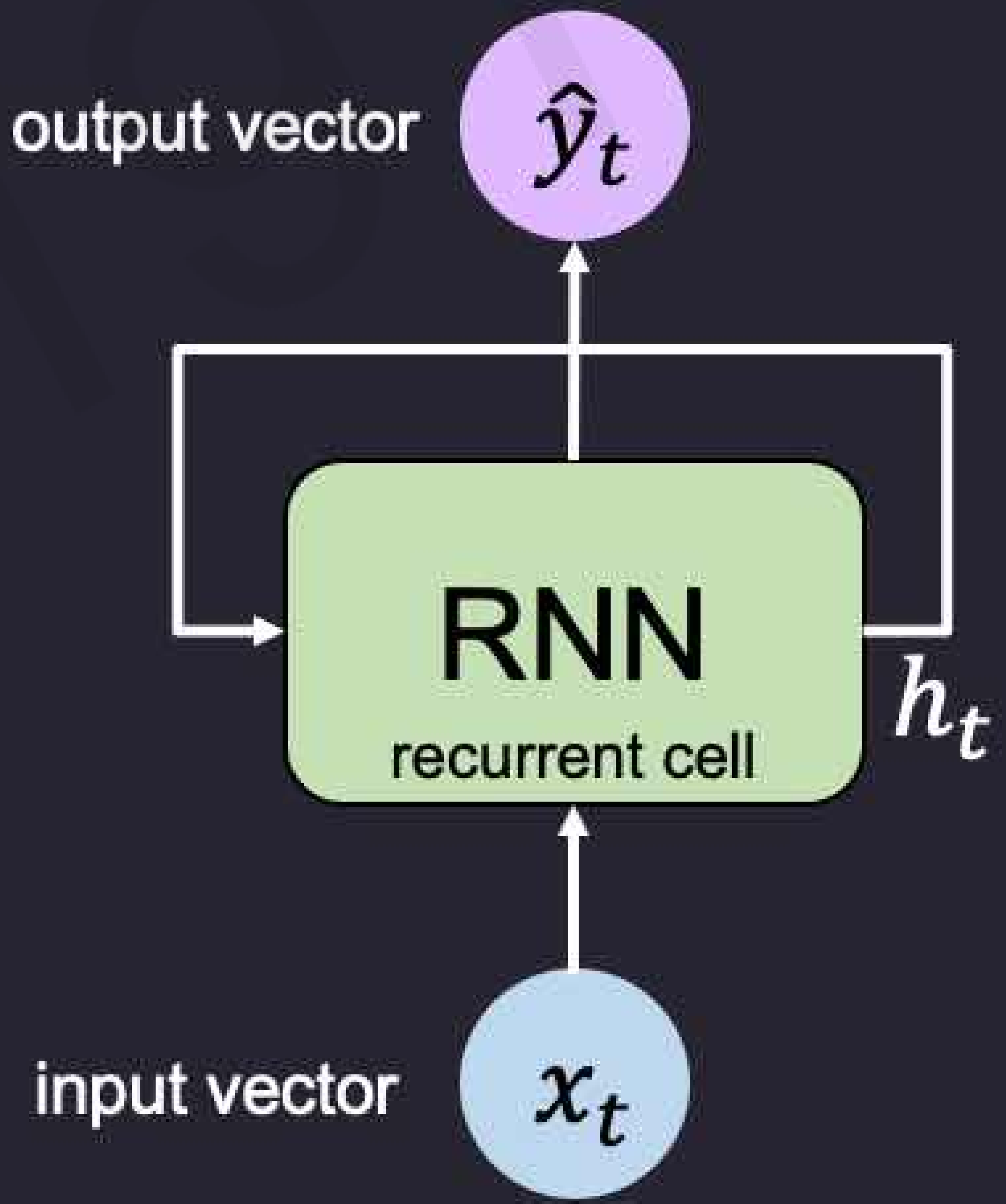
# RNNs from Scratch

```python
class MyRNNCell(tf.keras.layers.Layer):
  def __init__(self, rnn_units, input_dim, output_dim):
    super(MyRNNCell, self).__init__()

    # Initialize weight matrices
    self.W_xh = self.add_weight([rnn_units, input_dim])
    self.W_hh = self.add_weight([rnn_units, rnn_units])
    self.W_hy = self.add_weight([output_dim, rnn_units])

    # Initialize hidden state to zeros
    self.h = tf.zeros([rnn_units, 1])

  def call(self, x):
    # Update the hidden state
    self.h = tf.math.tanh( self.W_hh * self.h + self.W_xh * x )

    # Compute the output
    output = self.W_hy * self.h

    # Return the current output and hidden state
    return output, self.h
```



# RNN Implementation in TensorFlow

```
tf.keras.layers.SimpleRNN(rnn_units)
```



# RNNs for Sequence Modeling


One to One
"Vanilla" NN
*Binary classification*

Many to One
*Sentiment Classification*

One to Many
*Text Generation*
*Image Captioning*

Many to Many
*Translation & Forecasting*
*Music Generation*

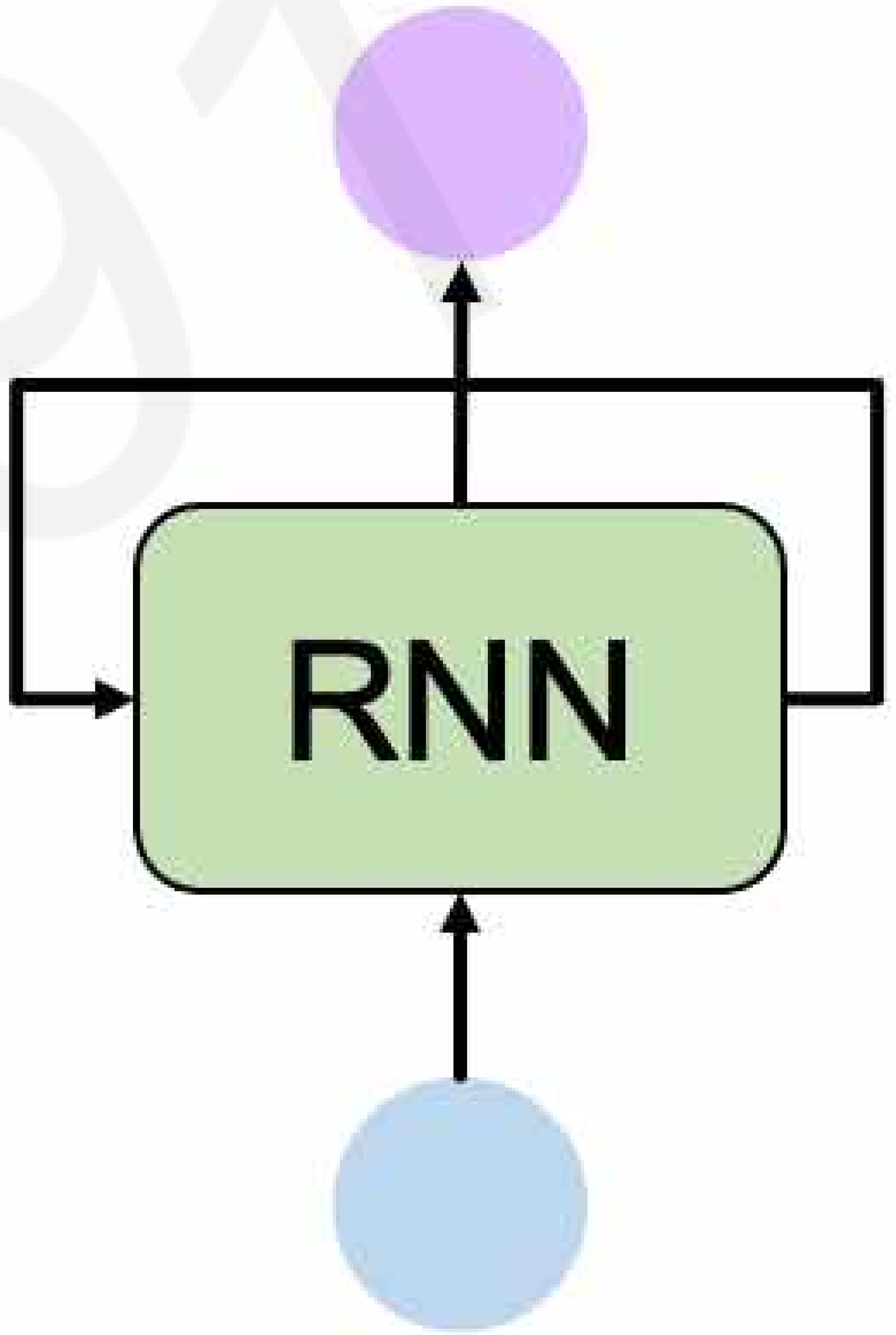
... and many other architectures and applications

**6.S191 Lab!**

# Sequence Modeling: Design Criteria

To model sequences, we need to:

1. Handle **variable-length** sequences
2. Track **long-term** dependencies
3. Maintain information about **order**
4. **Share parameters** across the sequence




**Recurrent Neural Networks (RNNs)** meet these sequence modeling design criteria

# A Sequence Modeling Problem: Predict the Next Word

# A Sequence Modeling Problem: Predict the Next Word

"This morning I took my cat for a walk."

# A Sequence Modeling Problem: Predict the Next Word

"This morning I took my cat for a walk."

given these words

# A Sequence Modeling Problem: Predict the Next Word

"This morning I took my cat for a walk."

given these words

predict the next word

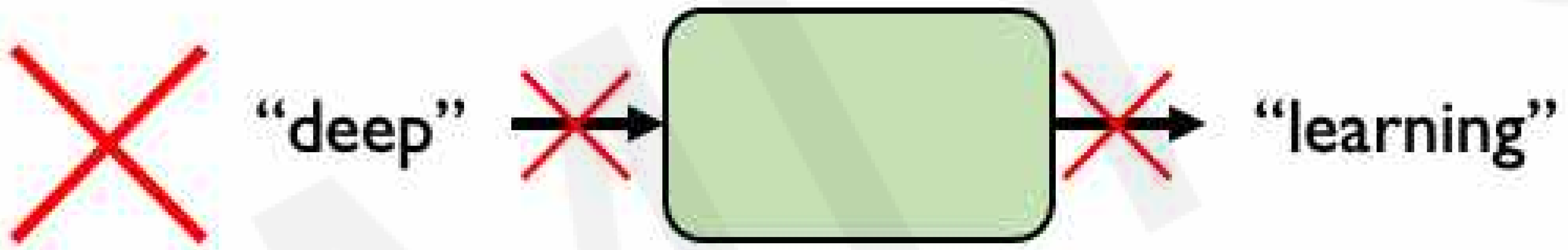
# A Sequence Modeling Problem: Predict the Next Word

"This morning I took my cat for a walk."

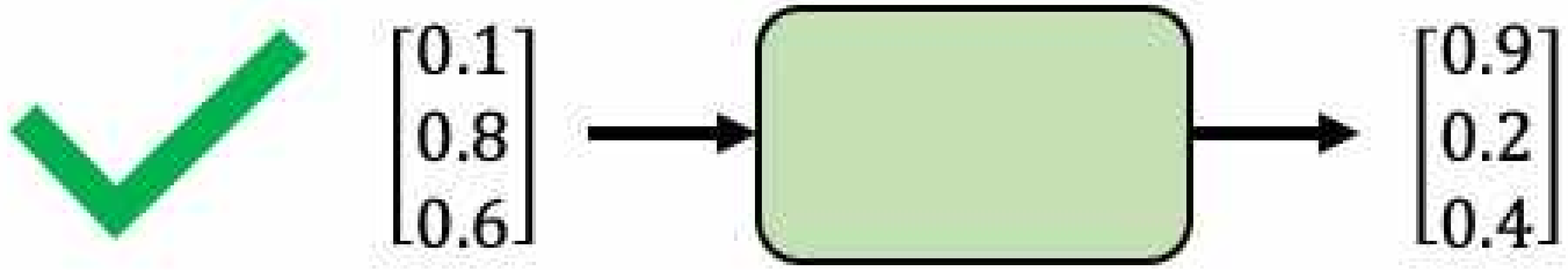
given these words

predict the next word

## Representing Language to a Neural Network

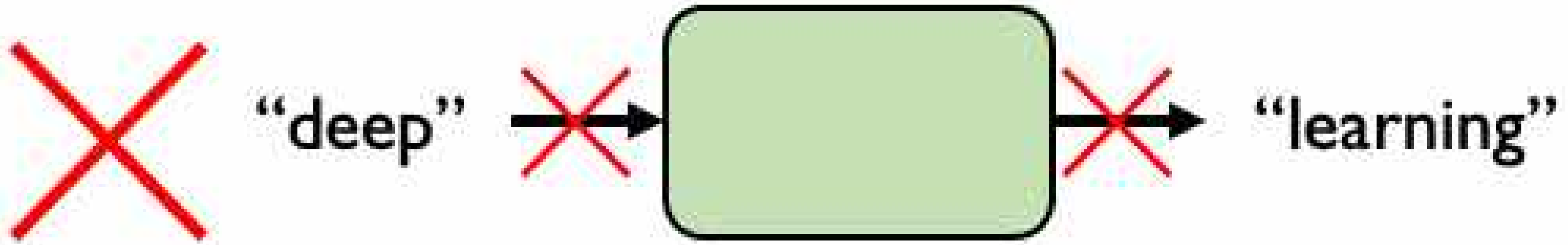



*Neural networks cannot interpret words*

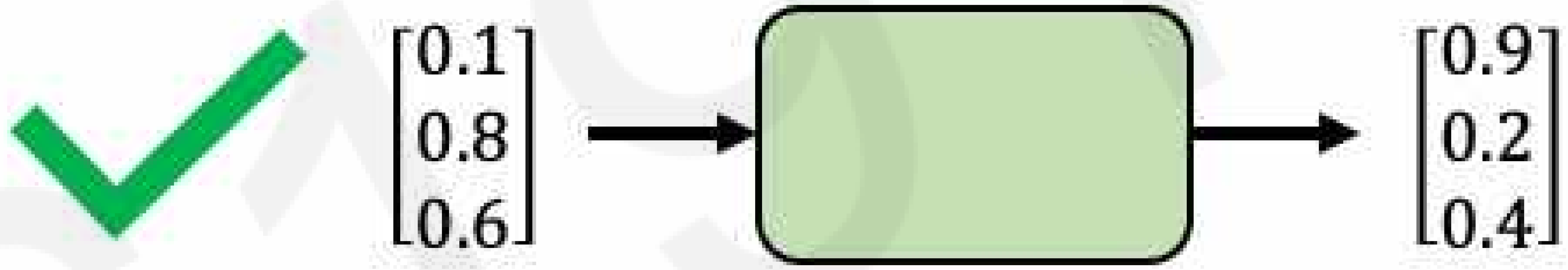



*Neural networks require numerical inputs*

# Encoding Language for a Neural Network




*Neural networks cannot interpret words*




*Neural networks require numerical inputs*

## Embedding: transform indexes into a vector of fixed size.

this cat for took my I walk a morning

**1. Vocabulary:**
Corpus of words

a → 1
cat → 2
… …
walk → N

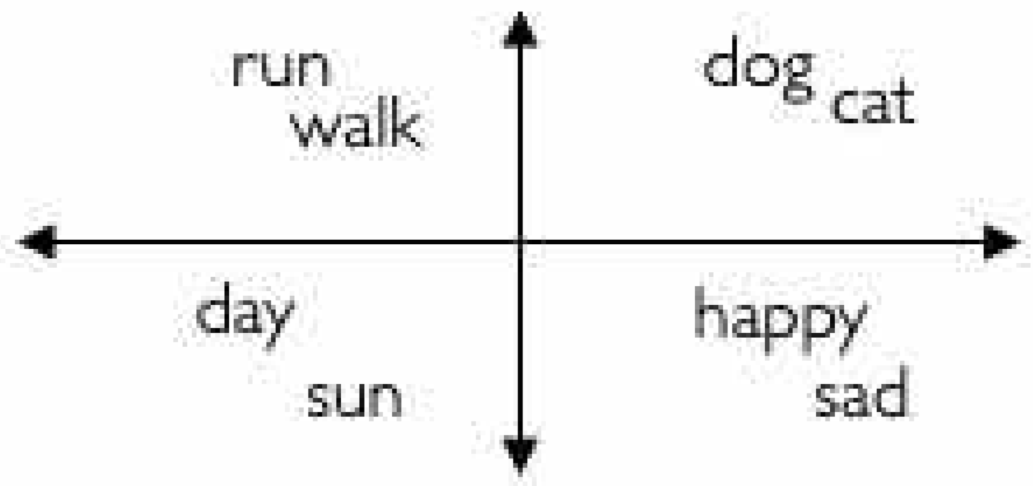
**2. Indexing:**
Word to index

**One-hot embedding**

"cat" = [ 0, 1, 0, 0, 0, 0 ]

*i*-th index

**Learned embedding**




**3. Embedding:**
Index to fixed-sized vector

# Handle Variable Sequence Lengths

The food was <u>great</u>

vs.

We visited a restaurant for <u>lunch</u>

vs.

We were hungry but cleaned the house before <u>eating</u>

# Model Long-Term Dependencies

"**France** is where I grew up, but I now live in Boston. I speak fluent ___."

***J'aime 6.S191!***

We need information from **the distant past** to accurately predict the correct word.

# Capture Differences in Sequence Order

The food was good, not bad at all.

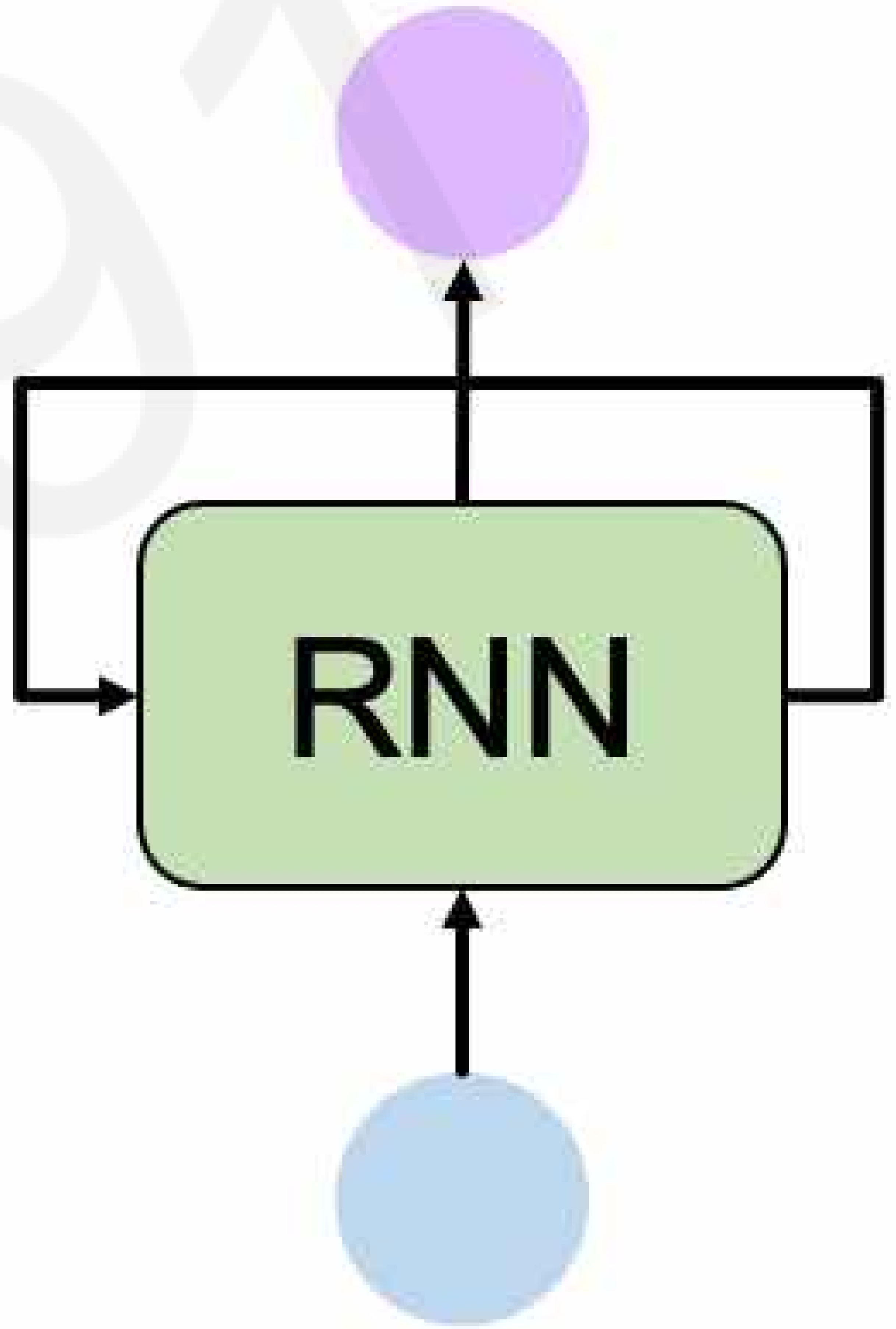
vs.

The food was bad, not good at all.

# Sequence Modeling: Design Criteria

To model sequences, we need to:

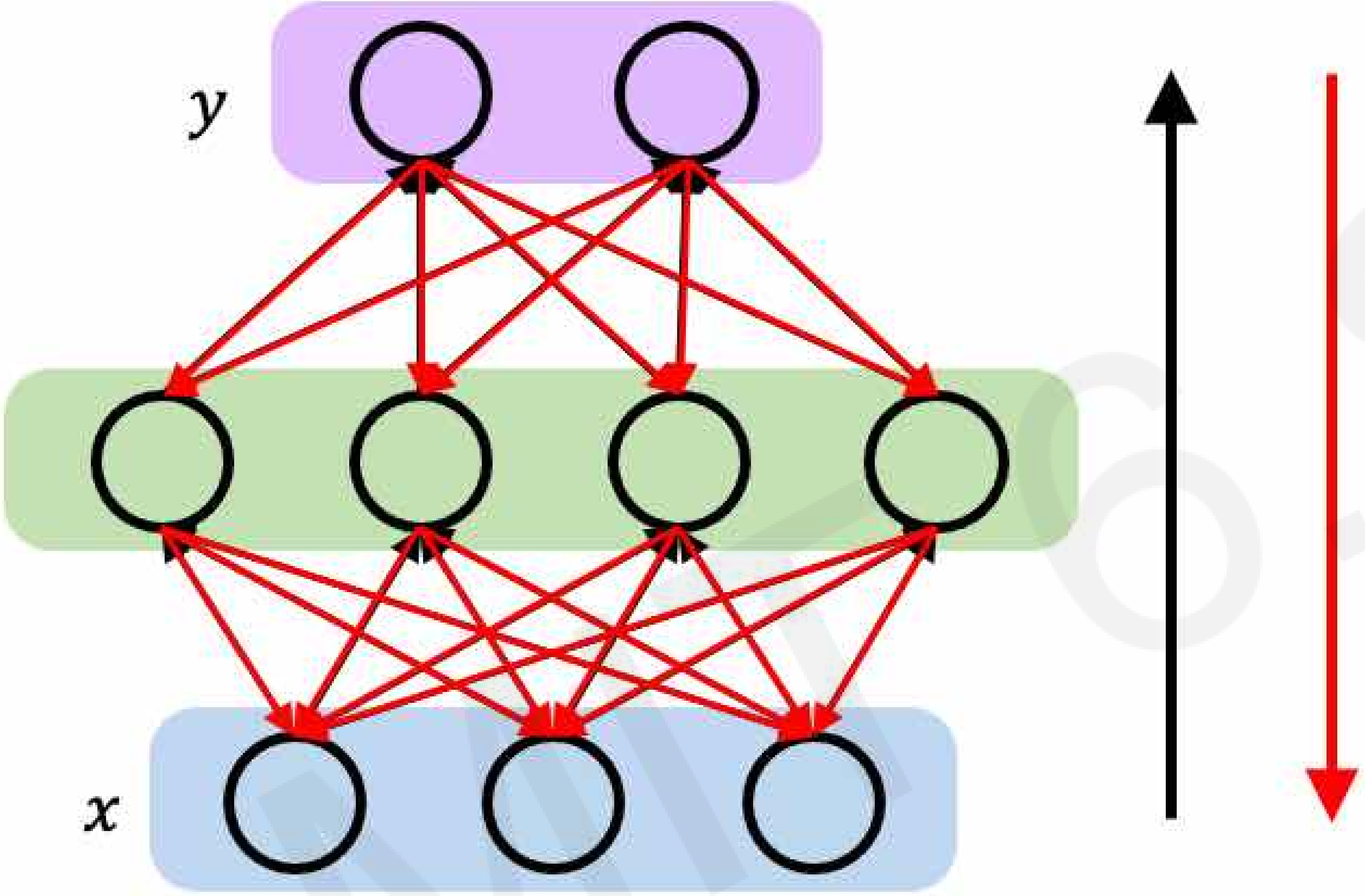
1. Handle **variable-length** sequences
2. Track **long-term** dependencies
3. Maintain information about **order**
4. **Share parameters** across the sequence




**Recurrent Neural Networks (RNNs)** meet these sequence modeling design criteria

# Backpropagation Through Time (BPTT)

# Recall: Backpropagation in Feed Forward Models



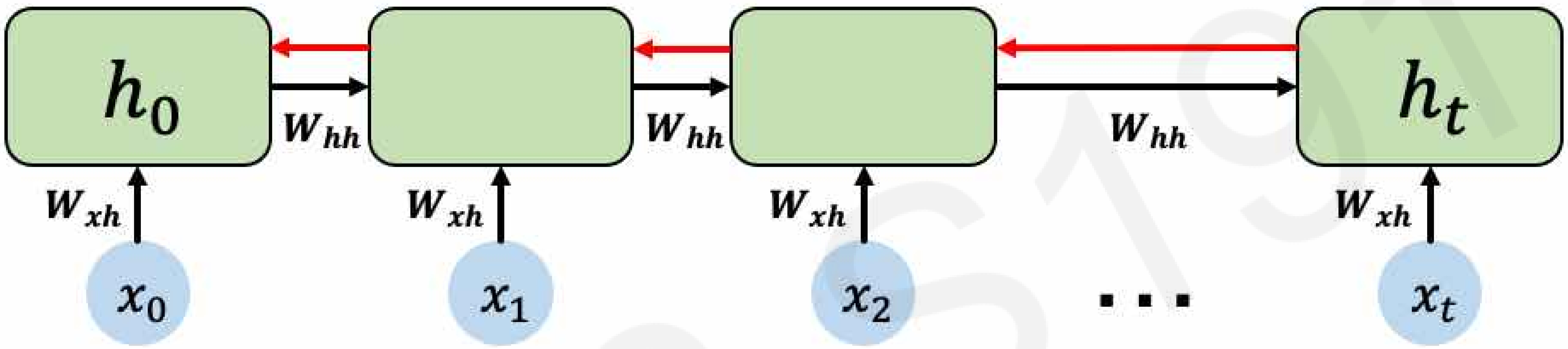

**Backpropagation algorithm:**

1. Take the derivative (gradient) of the loss with respect to each parameter
2. Shift parameters in order to minimize loss

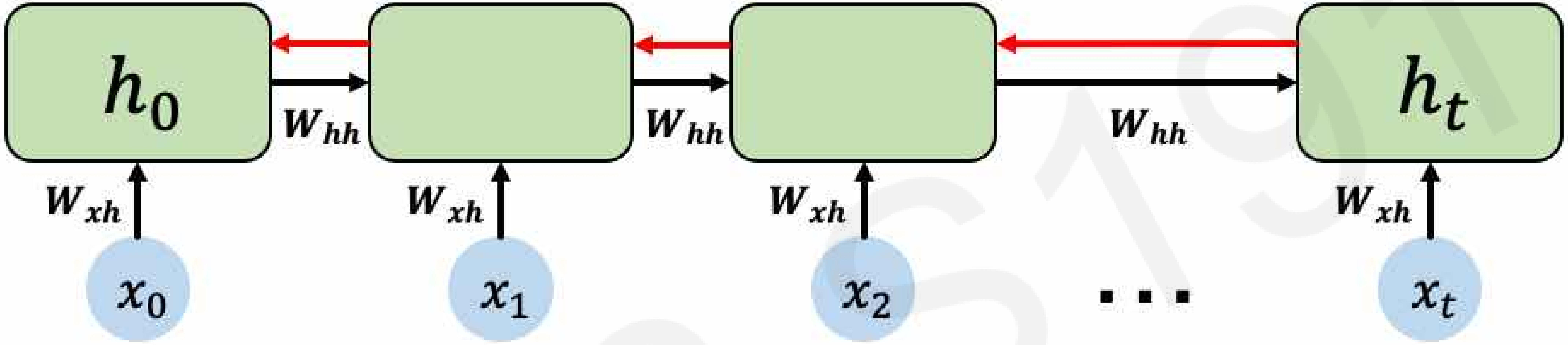
# RNNs: Backpropagation Through Time

# RNNs: Backpropagation Through Time

# Standard RNN Gradient Flow

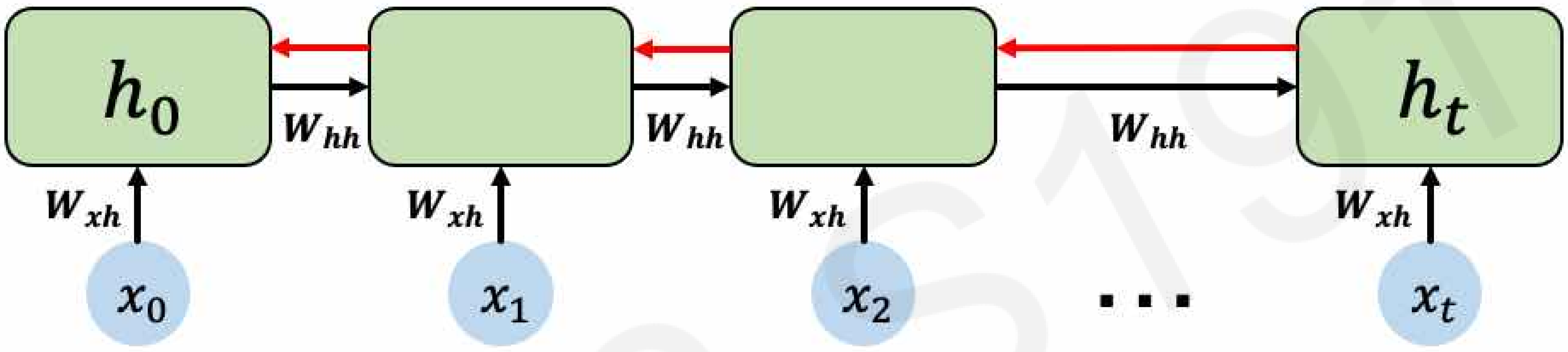


# Standard RNN Gradient Flow




Computing the gradient wrt $h_0$ involves **many factors of $W_{hh}$** + **repeated gradient computation**!

# Standard RNN Gradient Flow: Exploding Gradients

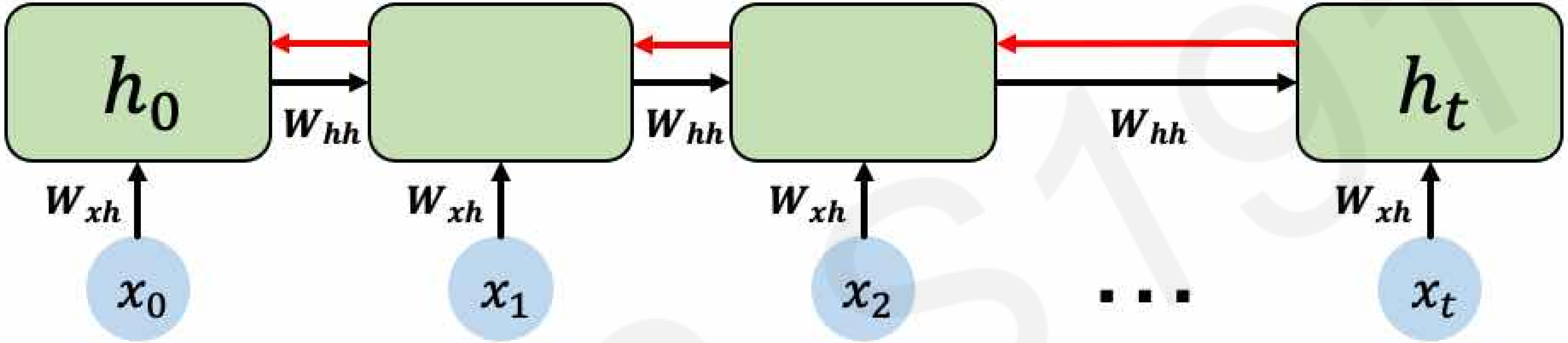



Computing the gradient wrt $h_0$ involves **many factors of $W_{hh}$ + repeated gradient computation!**

Many values > 1:
**exploding gradients**

**Gradient clipping** to scale big gradients

# Standard RNN Gradient Flow: Vanishing Gradients




Computing the gradient wrt $h_0$ involves **many factors of $W_{hh}$ + repeated gradient computation!**

Many values > 1:
**exploding gradients**

**Gradient clipping** to scale big gradients

Many values < 1:
**vanishing gradients**

1. Activation function
2. Weight initialization
3. Network architecture

# The Problem of Long-Term Dependencies

**Why are vanishing gradients a problem?**

Multiply many **small numbers** together

↓

Errors due to further back time steps have smaller and smaller gradients

↓

Bias parameters to capture short-term dependencies

# The Problem of Long-Term Dependencies

"The clouds are in the ___"

**Why are vanishing gradients a problem?**

Multiply many **small numbers** together

↓

Errors due to further back time steps have smaller and smaller gradients

↓

Bias parameters to capture short-term dependencies

# The Problem of Long-Term Dependencies

**Why are vanishing gradients a problem?**

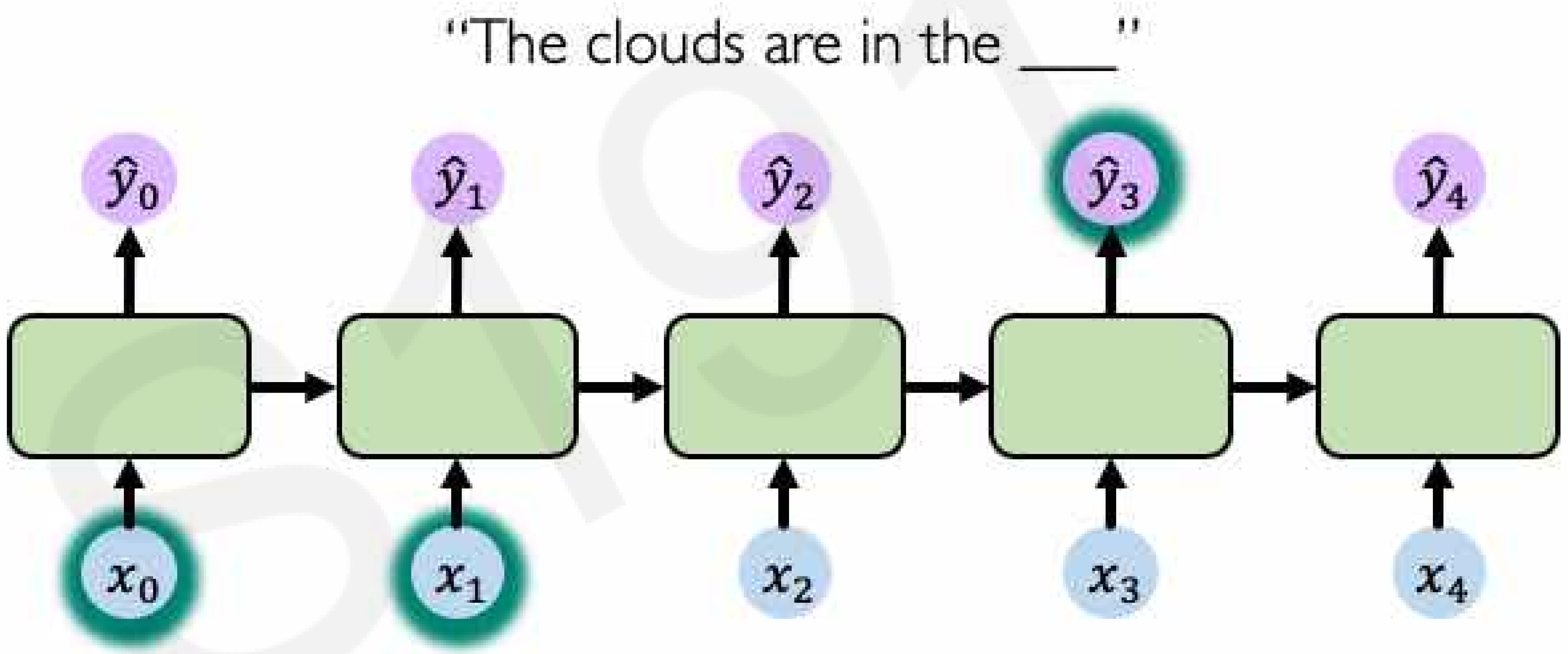
Multiply many **small numbers** together

↓

Errors due to further back time steps have smaller and smaller gradients

↓

Bias parameters to capture short-term dependencies



# The Problem of Long-Term Dependencies

**Why are vanishing gradients a problem?**

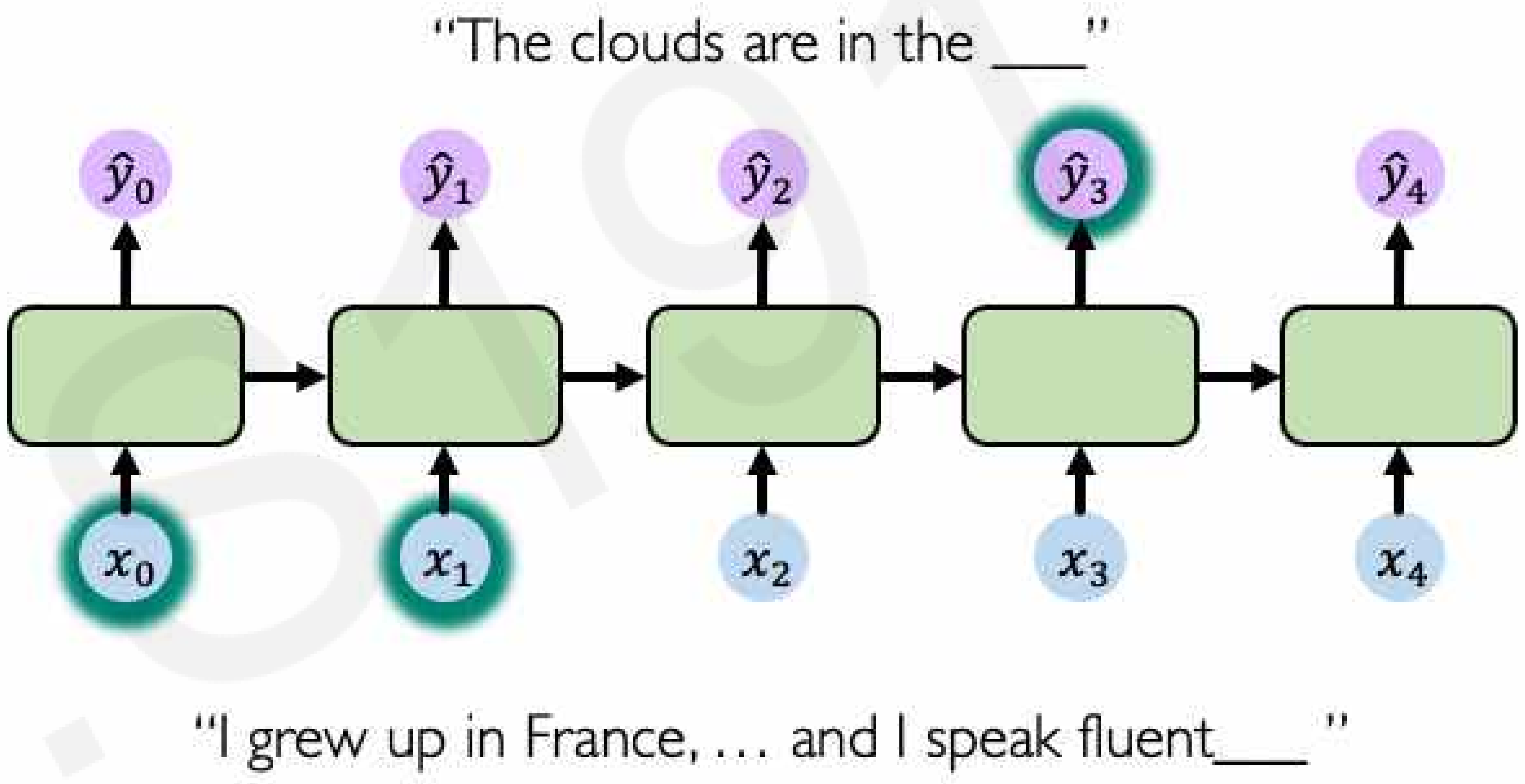
Multiply many **small numbers** together

↓

Errors due to further back time steps have smaller and smaller gradients

↓

Bias parameters to capture short-term dependencies



# The Problem of Long-Term Dependencies

**Why are vanishing gradients a problem?**

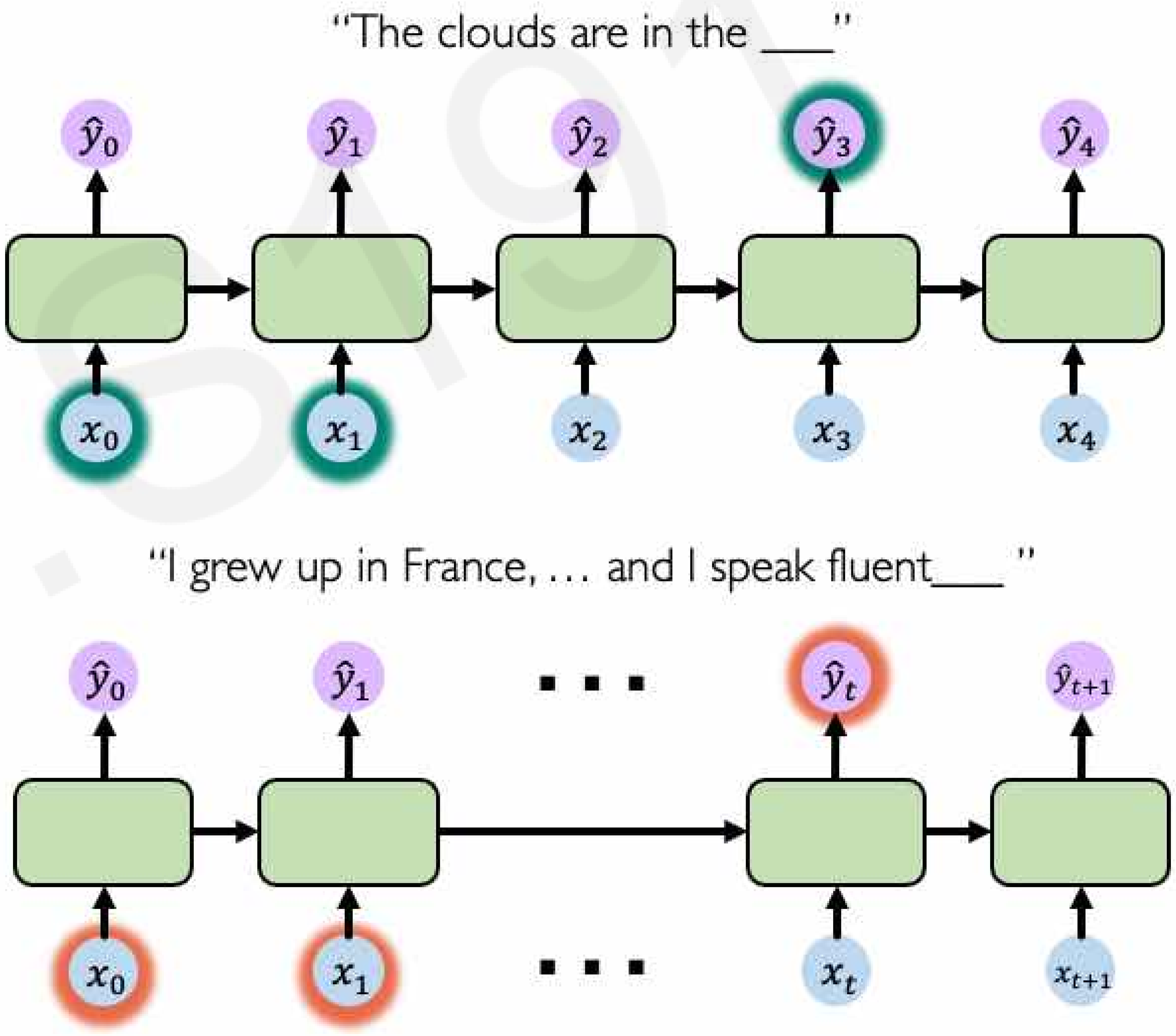
Multiply many **small numbers** together

↓

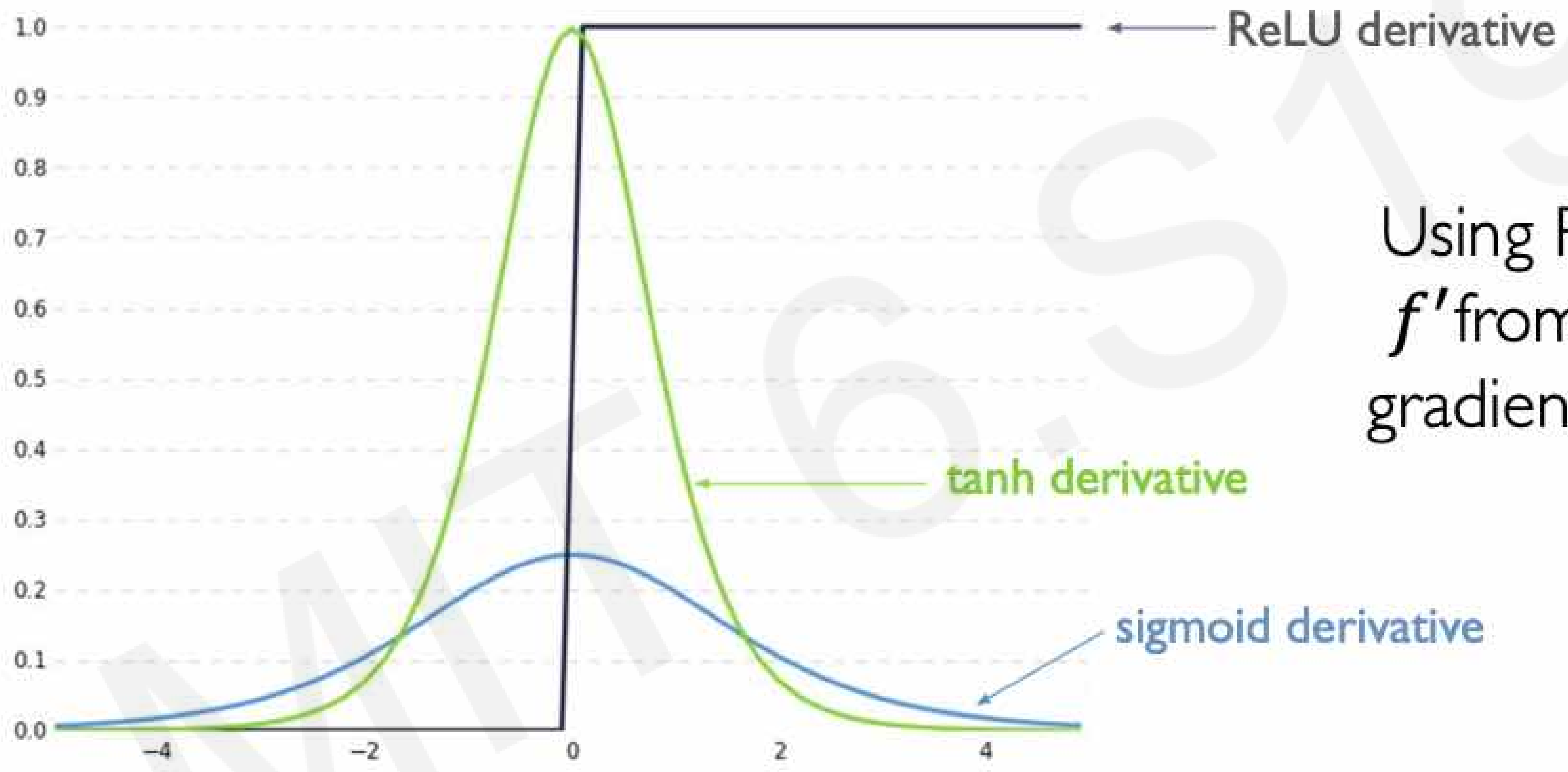
Errors due to further back time steps have smaller and smaller gradients

↓

Bias parameters to capture short-term dependencies



# Trick #1: Activation Functions




Using ReLU prevents $f'$ from shrinking the gradients when $x > 0$

# Trick #2: Parameter Initialization

Initialize **weights** to identity matrix

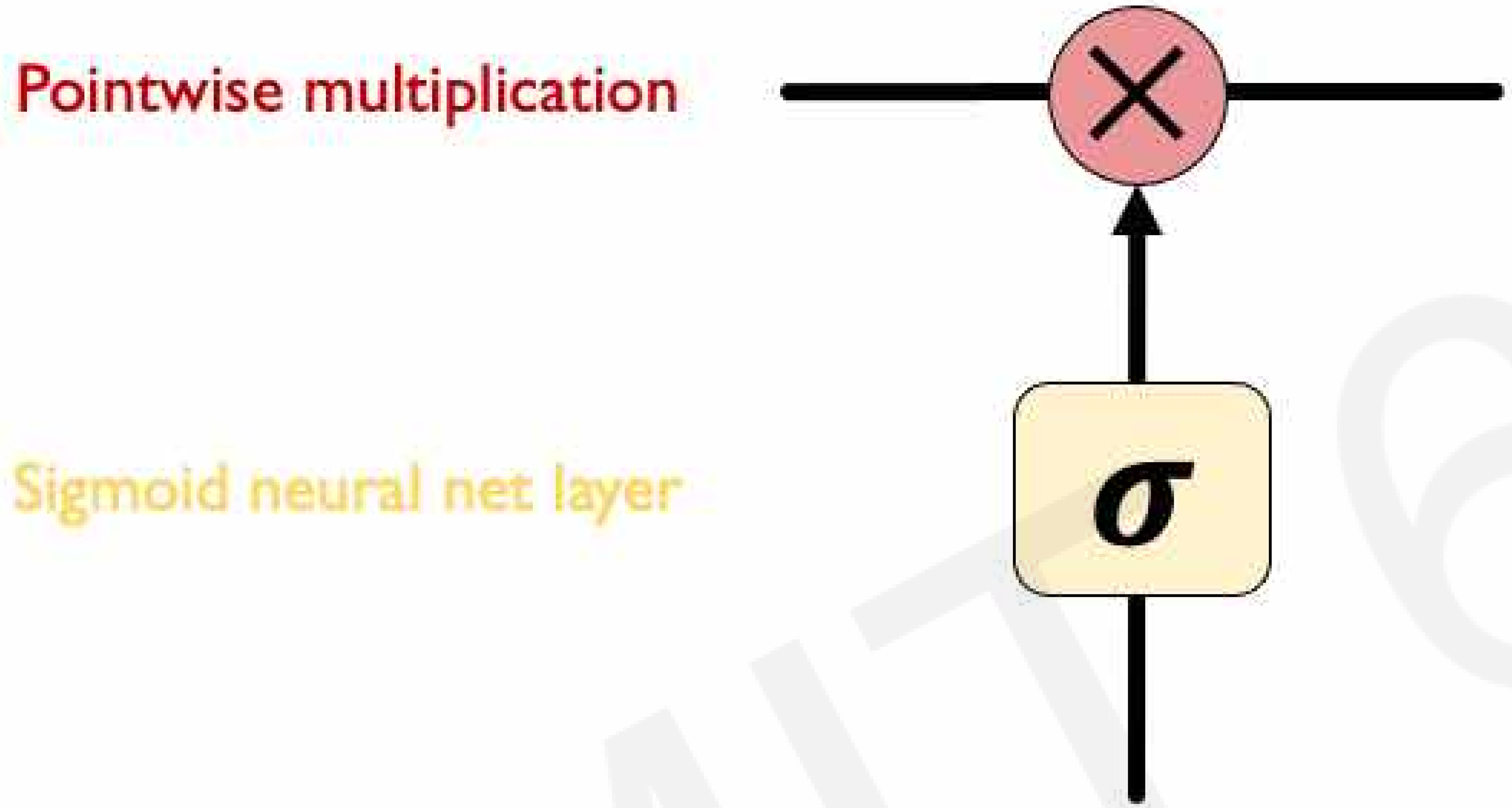
Initialize **biases** to zero

$$I_n = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

This helps prevent the weights from shrinking to zero.

# Trick #3: Gated Cells

Idea: use **gates** to selectively **add** or **remove** information within **each recurrent unit with**





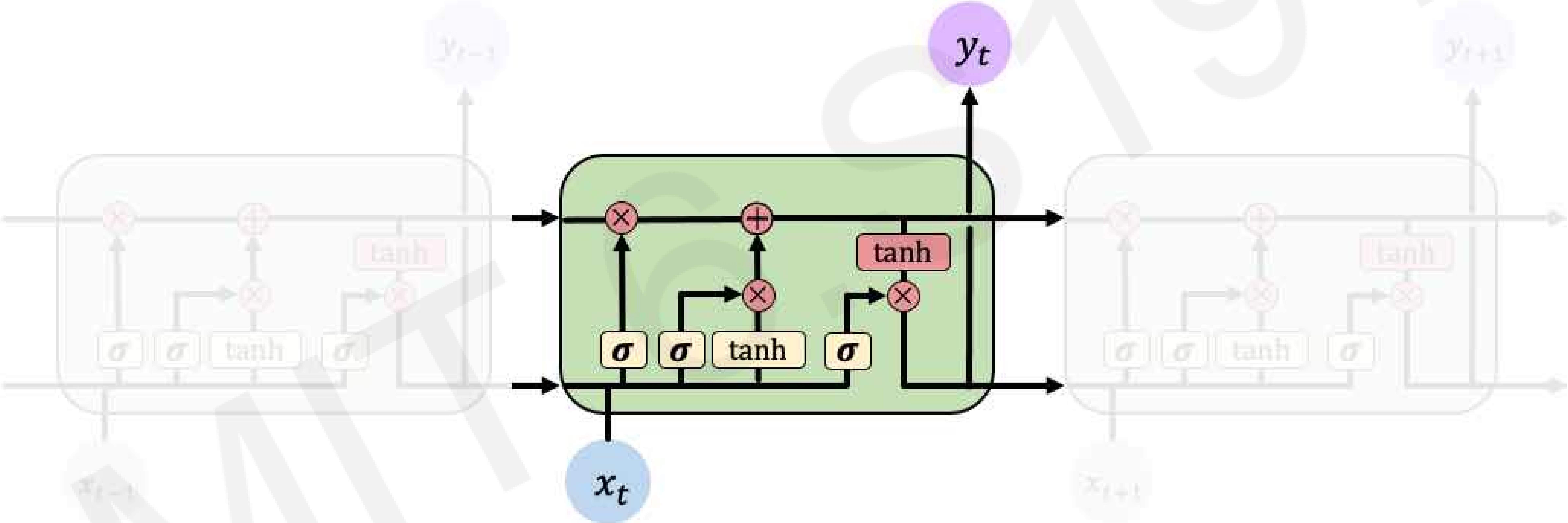
Gates optionally let information through the cell

**Long Short Term Memory (LSTMs)** networks rely on a gated cell to track information throughout many time steps.

# Long Short Term Memory (LSTMs)

Gated LSTM cells **control information flow:**

**1) Forget  2) Store  3) Update  4) Output**




LSTM cells are able to track information throughout many timesteps

```
tf.keras.layers.LSTM(num_units)
```



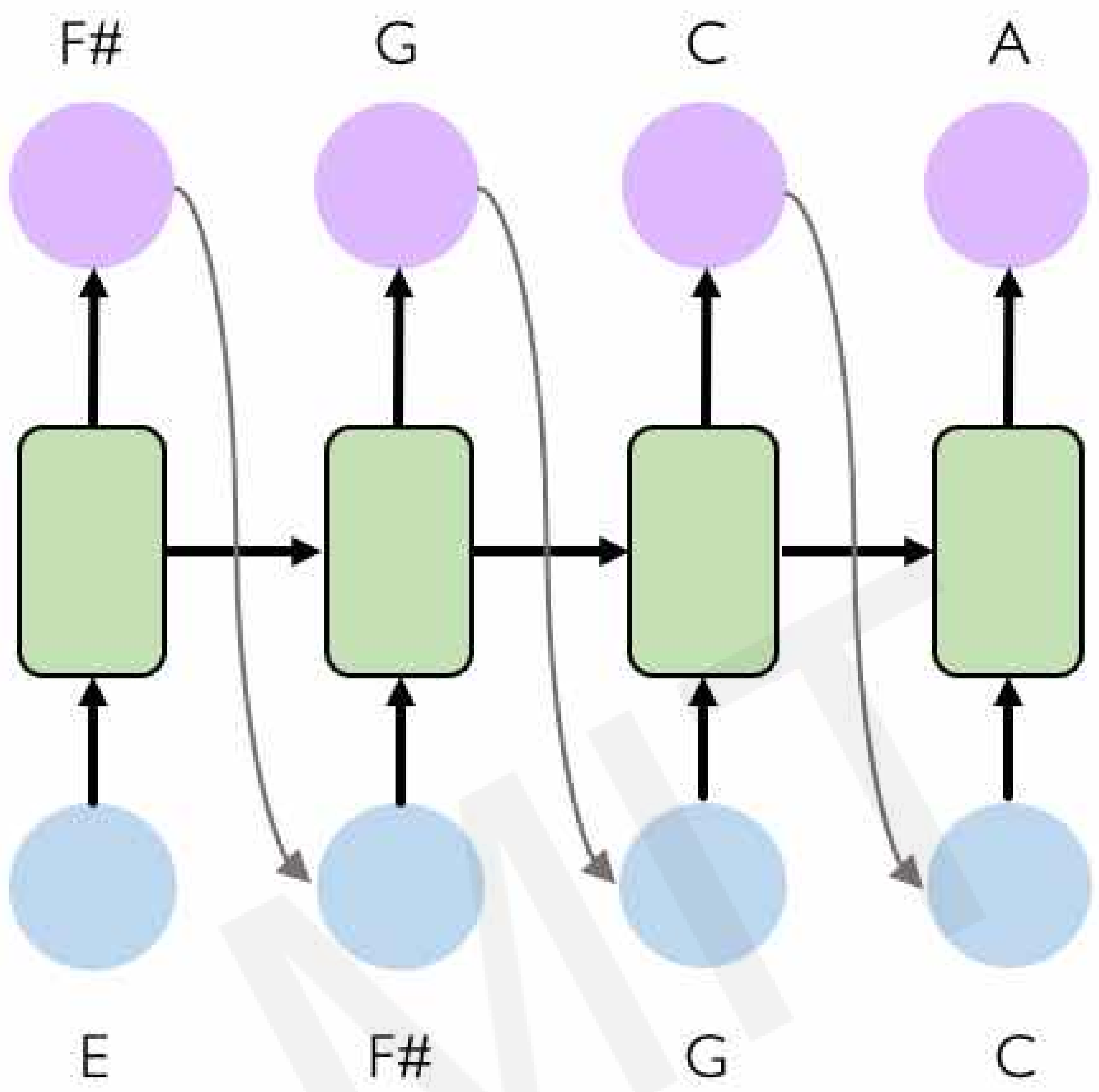

Hochreiter & Schmidhuber, *Neural Computation* 1997.

# LSTMs: Key Concepts

1. Maintain a **cell state**
2. Use **gates** to control the **flow of information**
   - **Forget** gate gets rid of irrelevant information
   - **Store** relevant information from current input
   - Selectively **update** cell state
   - **Output** gate returns a filtered version of the cell state
3. Backpropagation through time with partially **uninterrupted gradient flow**

# RNN Applications & Limitations

# Example Task: Music Generation

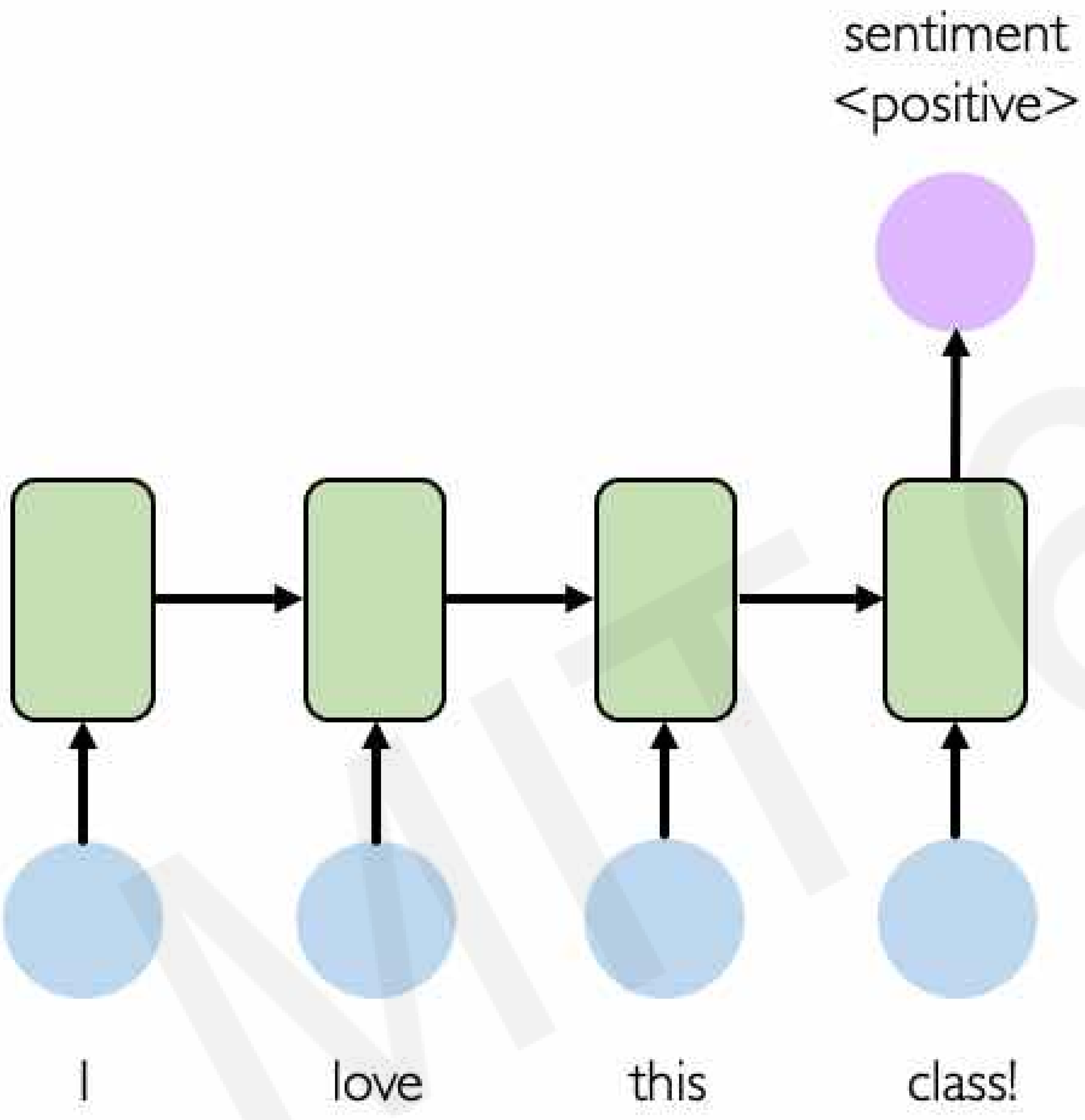



**Input:** sheet music

**Output:** next character in sheet music

6.S191 Lab!

# Example Task: Sentiment Classification

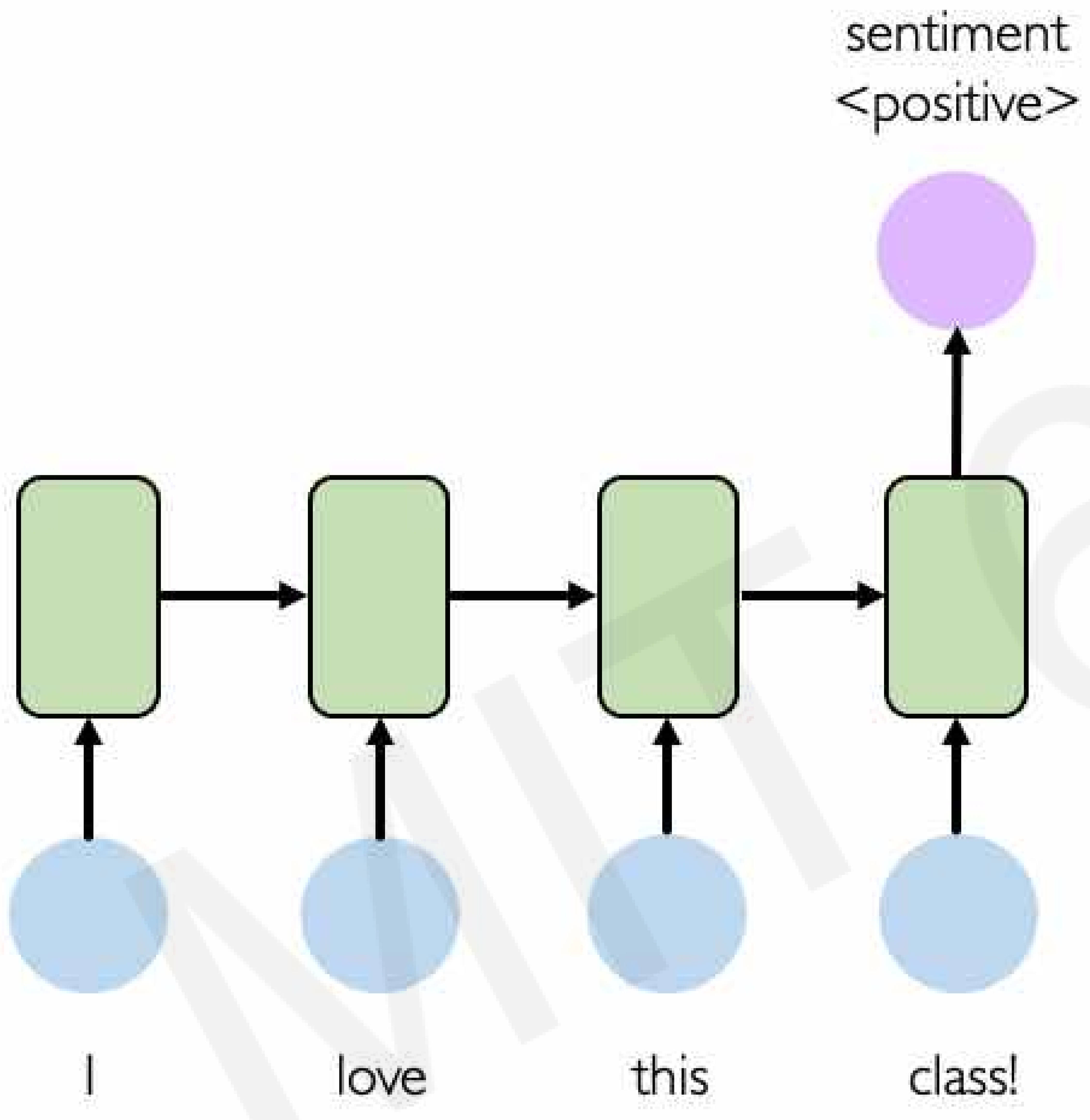



**Input:** sequence of words

**Output:** probability of having positive sentiment

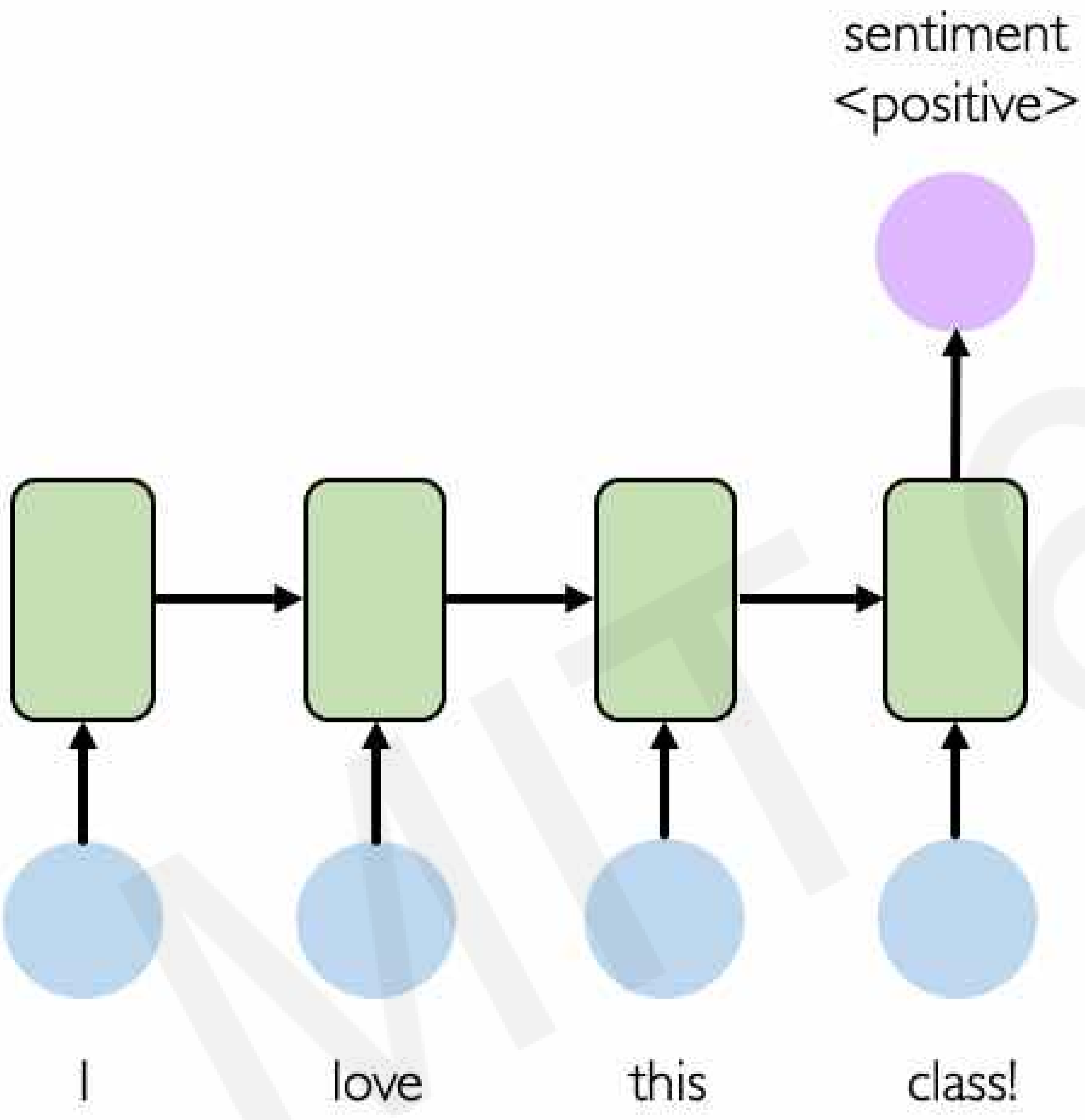
```
loss = tf.nn.softmax_cross_entropy_with_logits(y, predicted)
```

# Example Task: Sentiment Classification




**Tweet sentiment classification**

# Limitations of Recurrent Models




## Limitations of RNNs

- Encoding bottleneck
- Slow, no parallelization
- Not long memory

# Goal of Sequence Modeling

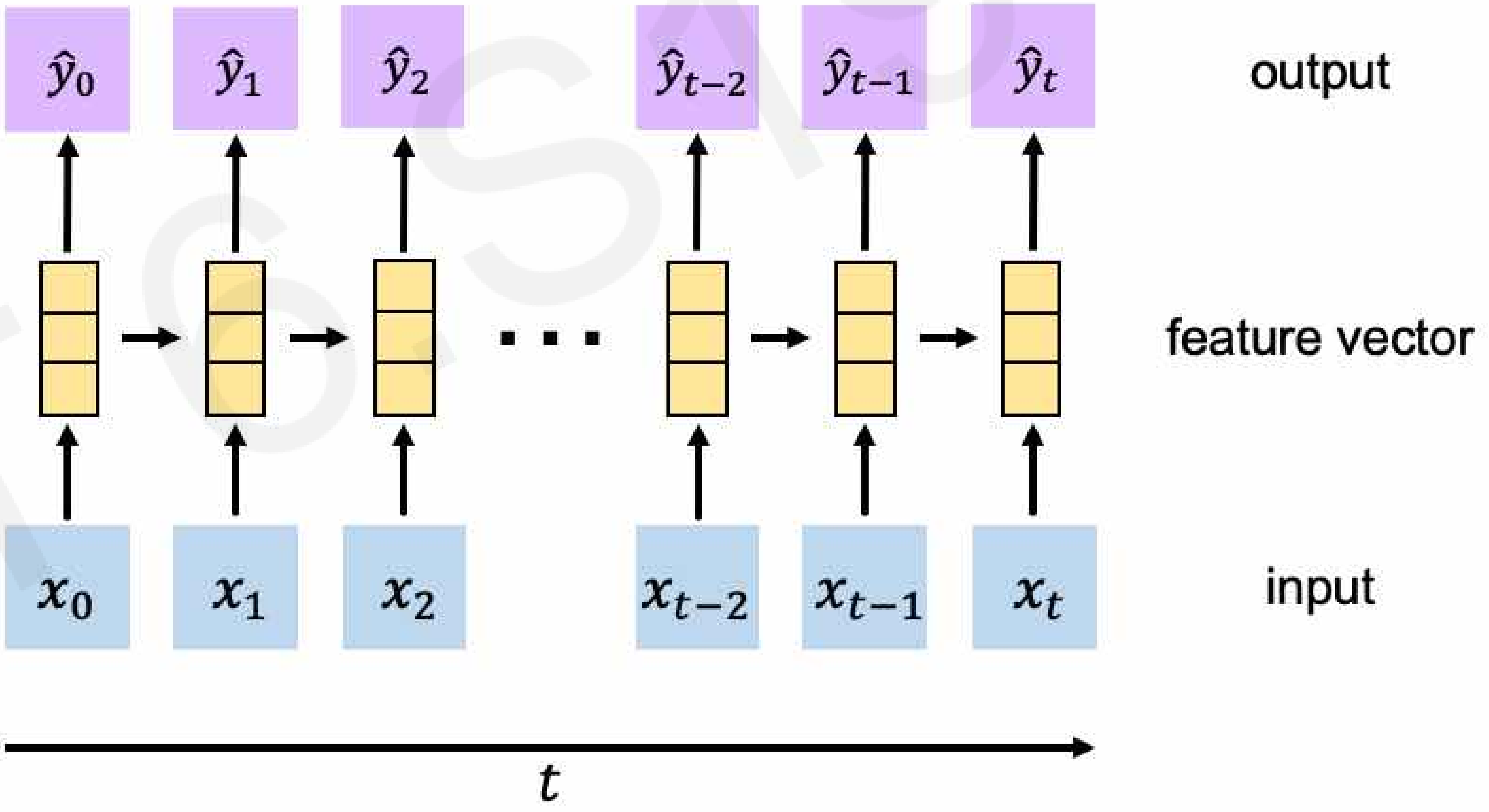
RNNs: recurrence to model sequence dependencies

# Goal of Sequence Modeling

## Limitations of RNNs

- Encoding bottleneck
- Slow, no parallelization
- Not long memory

RNNs: recurrence to model sequence dependencies

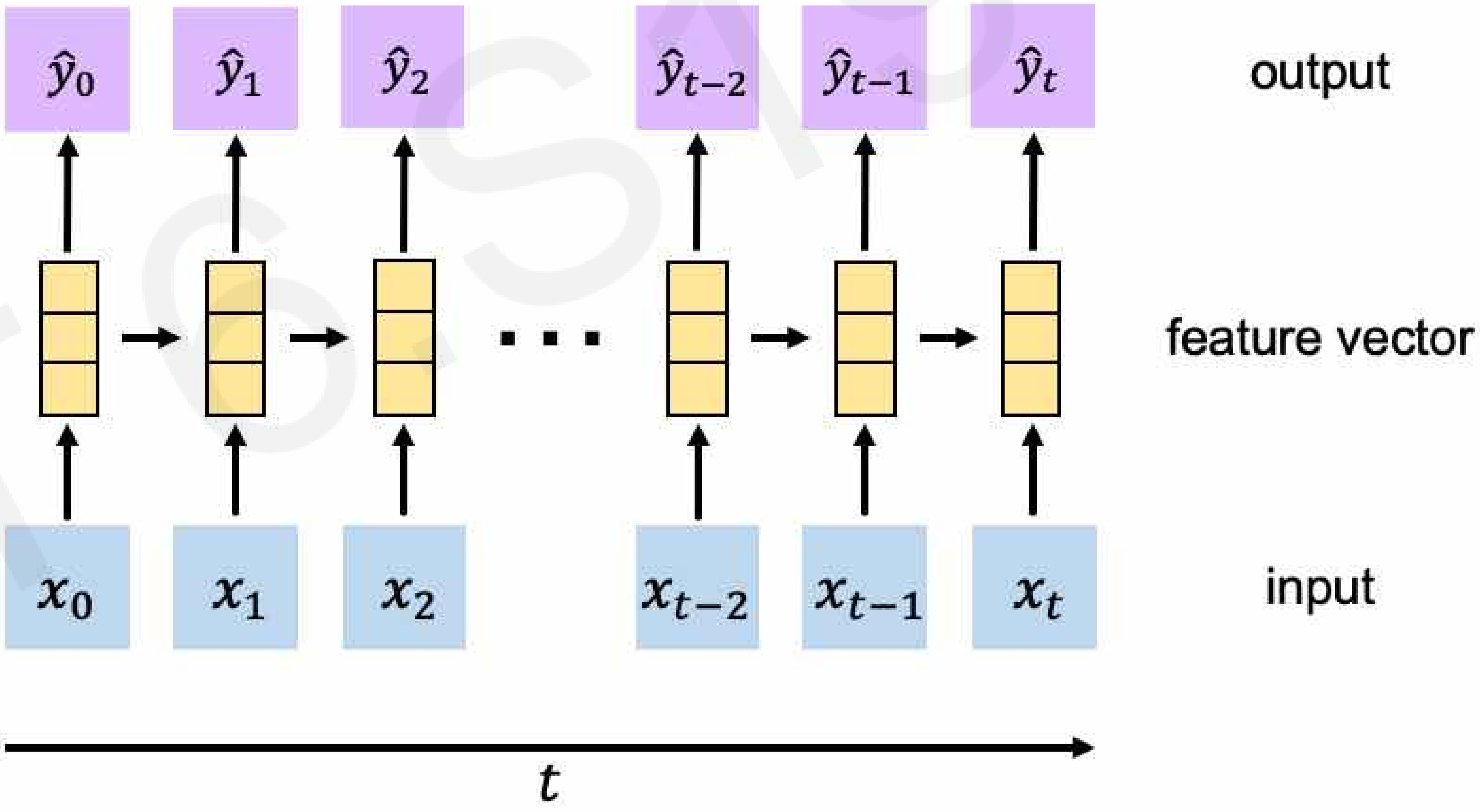


# Goal of Sequence Modeling

Can we eliminate the need for recurrence entirely?

## Desired Capabilities

- Continuous stream
- Parallelization
- Long memory

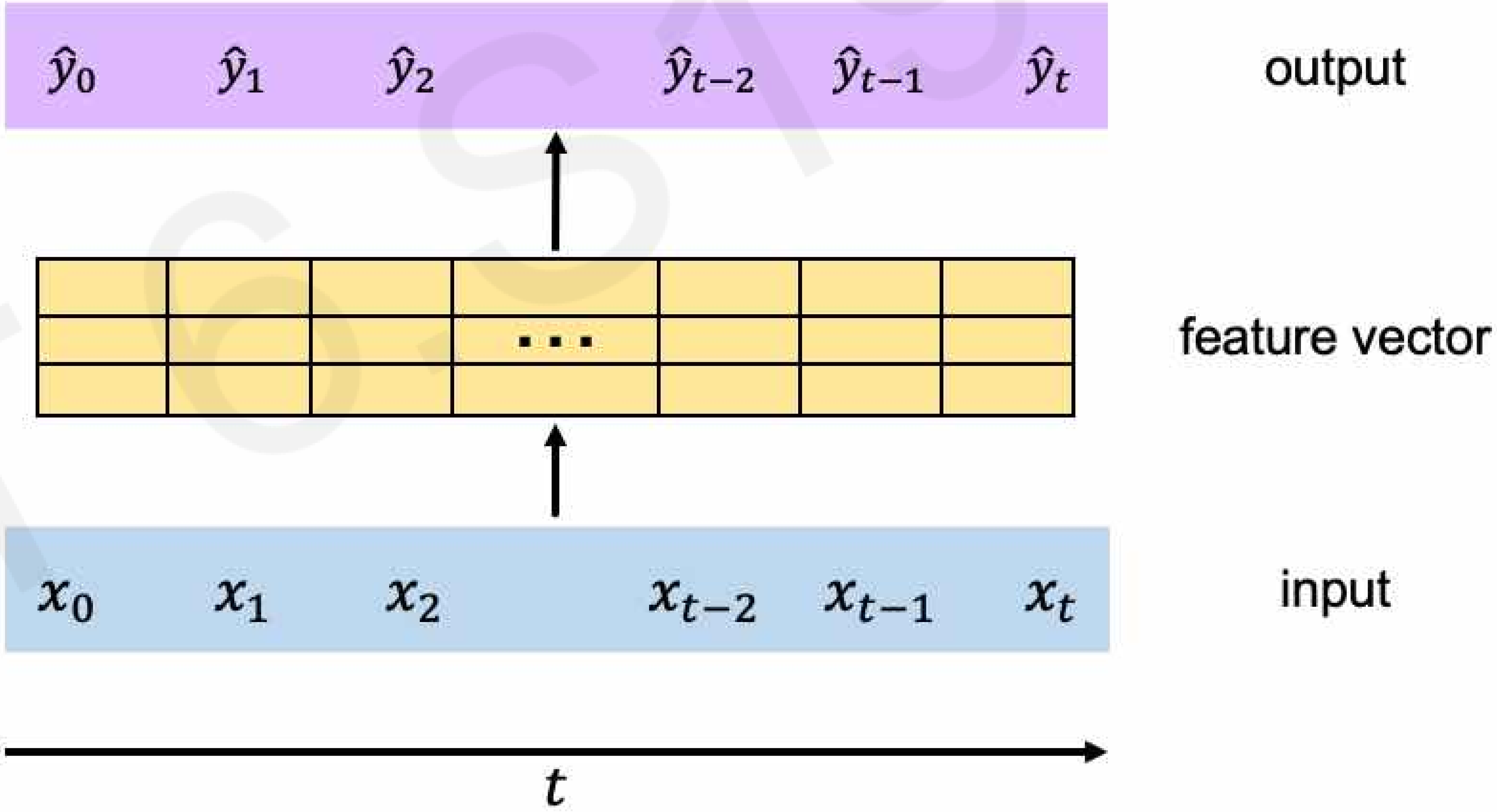


# Goal of Sequence Modeling

Can we eliminate the need for recurrence entirely?

## Desired Capabilities

- Continuous stream
- Parallelization
- Long memory



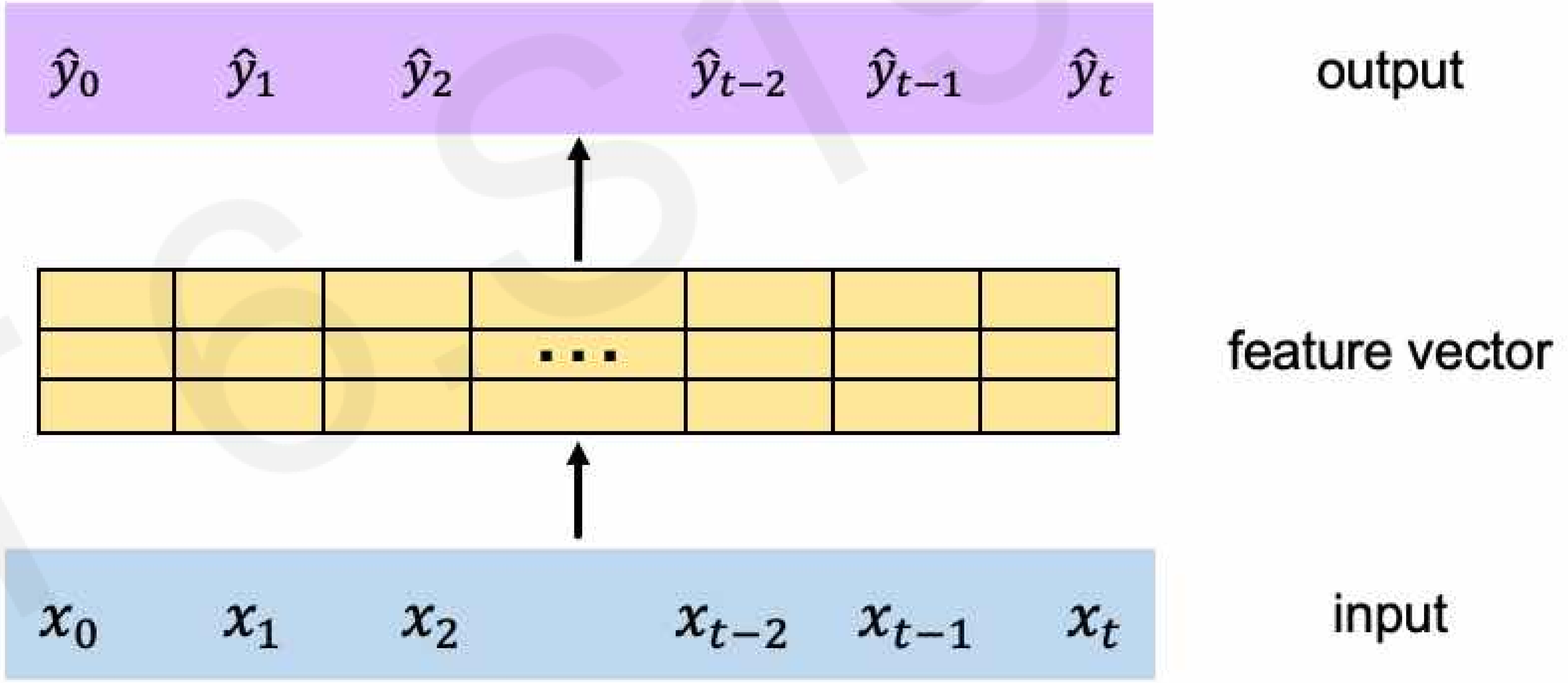
# Goal of Sequence Modeling

Idea 1: Feed everything into dense network

- ✓ No recurrence
- ✗ Not scalable
- ✗ No order
- ✗ No long memory

Idea: Identify and attend to what's important

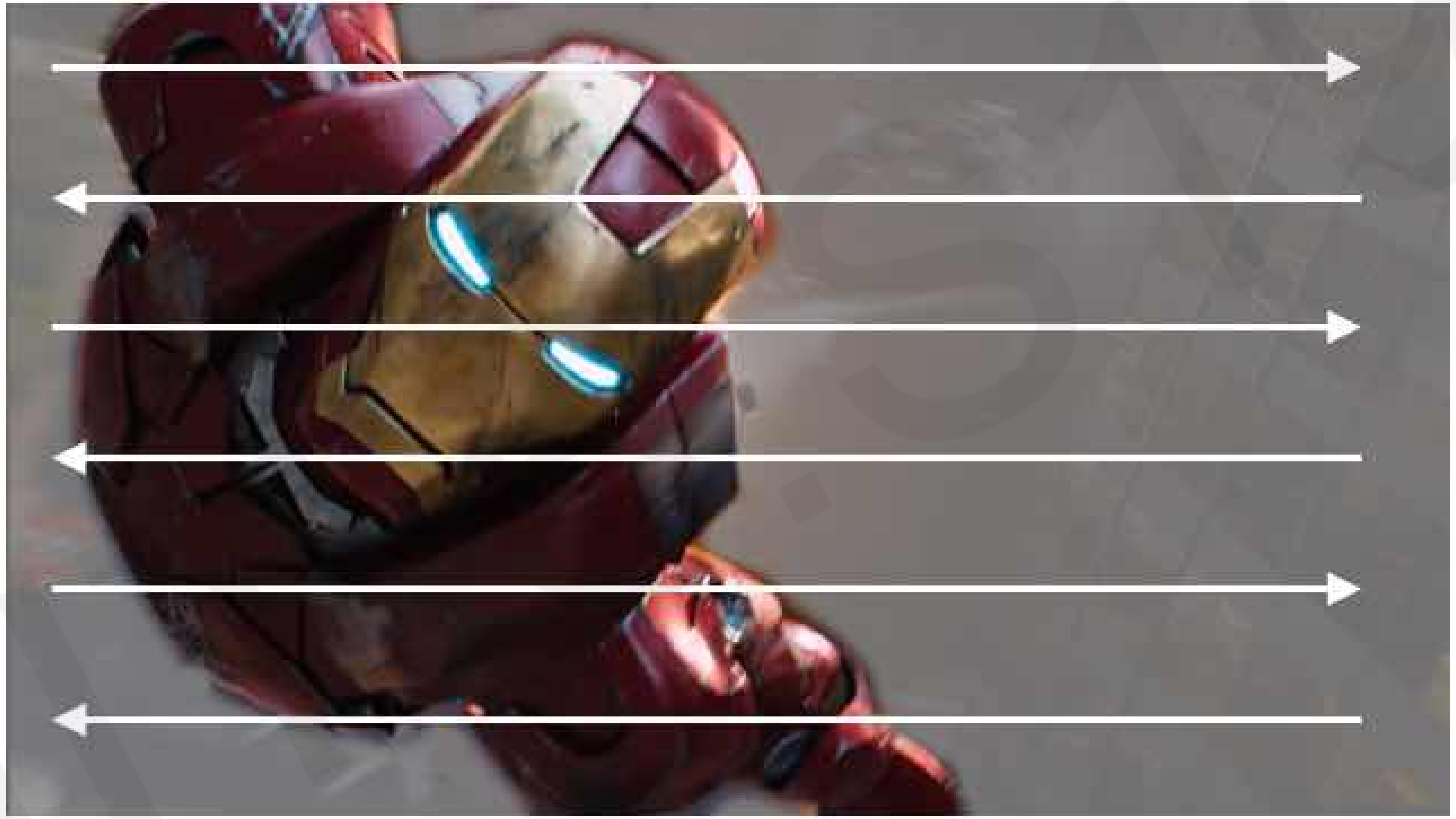
**Can we eliminate the need for recurrence entirely?**



# Attention Is All You Need

# Intuition Behind Self-Attention

Attending to the most important parts of an input.

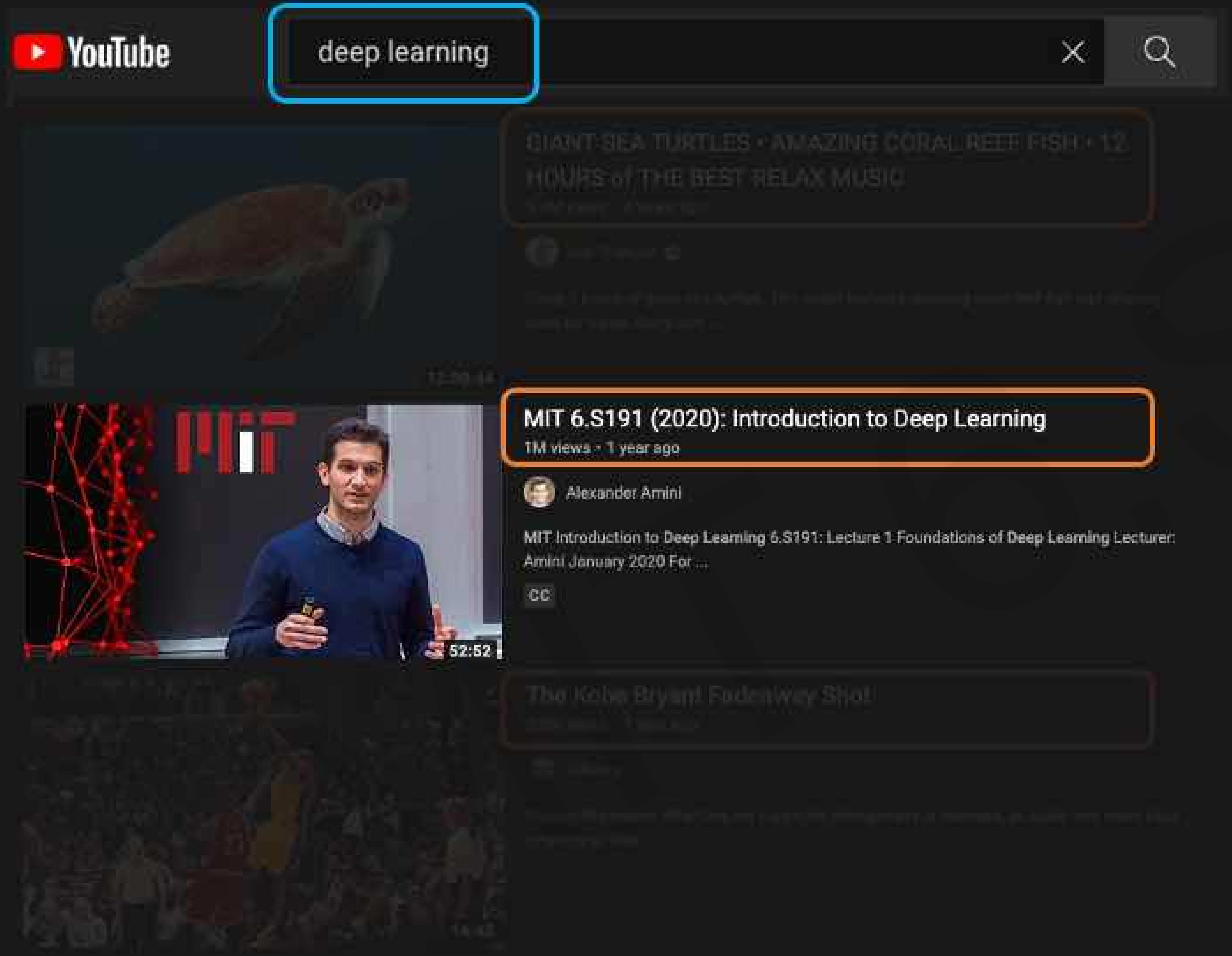


1. $\boxed{\text{Identify which parts to attend to}}$
2. Extract the features with high attention

Similar to a search problem!

# A Simple Example: Search

# Understanding Attention with Search




Query (Q)

Key ($K_1$)

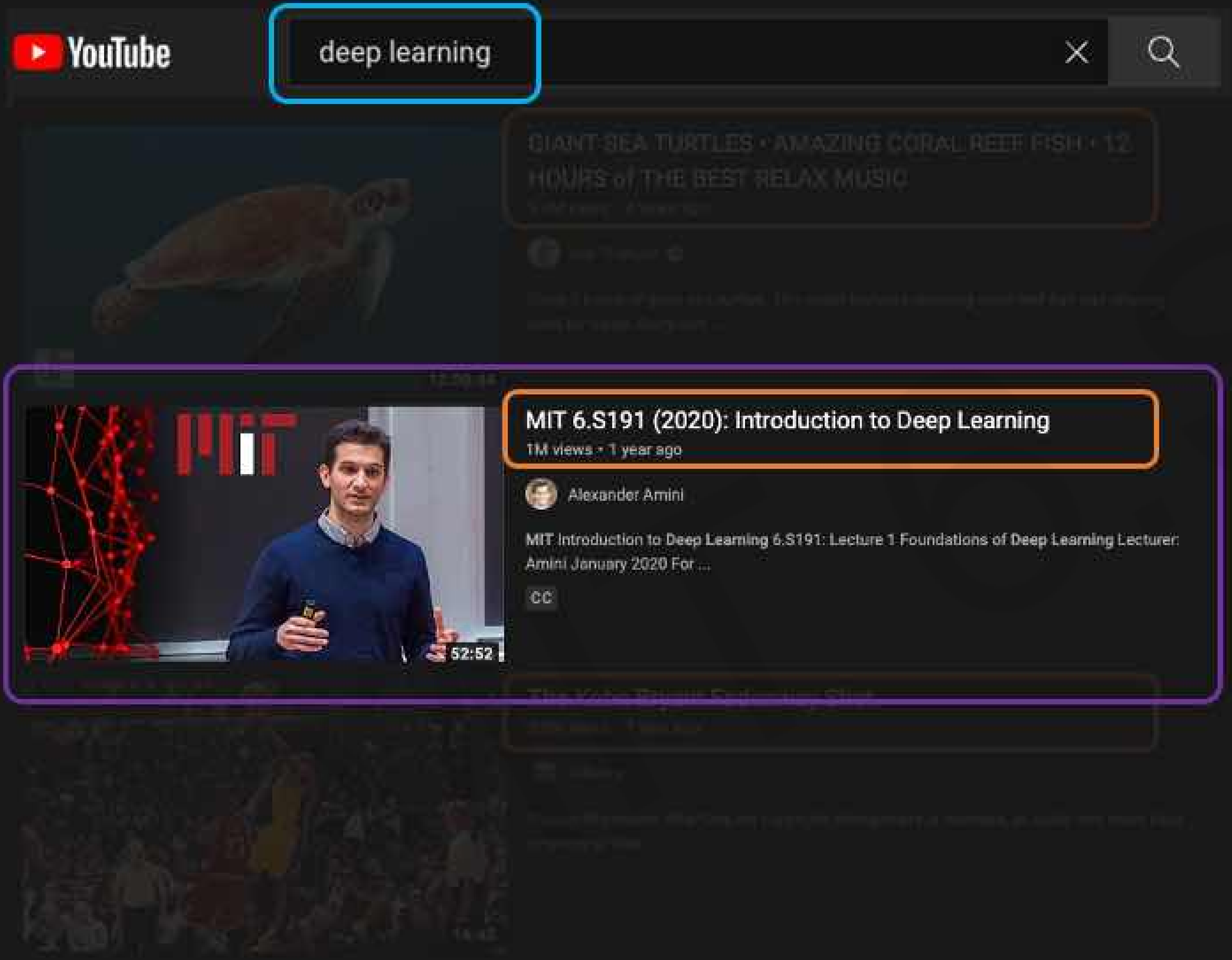
Key ($K_2$)

Key ($K_3$)

How similar is the key to the query?

1. **Compute attention mask:** how similar is each key to the desired query?

# Understanding Attention with Search




Query (Q)

Key ($K_1$)

Key ($K_2$)

Value (V)

Key ($K_3$)

2. **Extract values based on attention:**
Return the values highest attention

# Learning Self-Attention with Neural Networks

**Goal: identify and attend to most important features in input.**

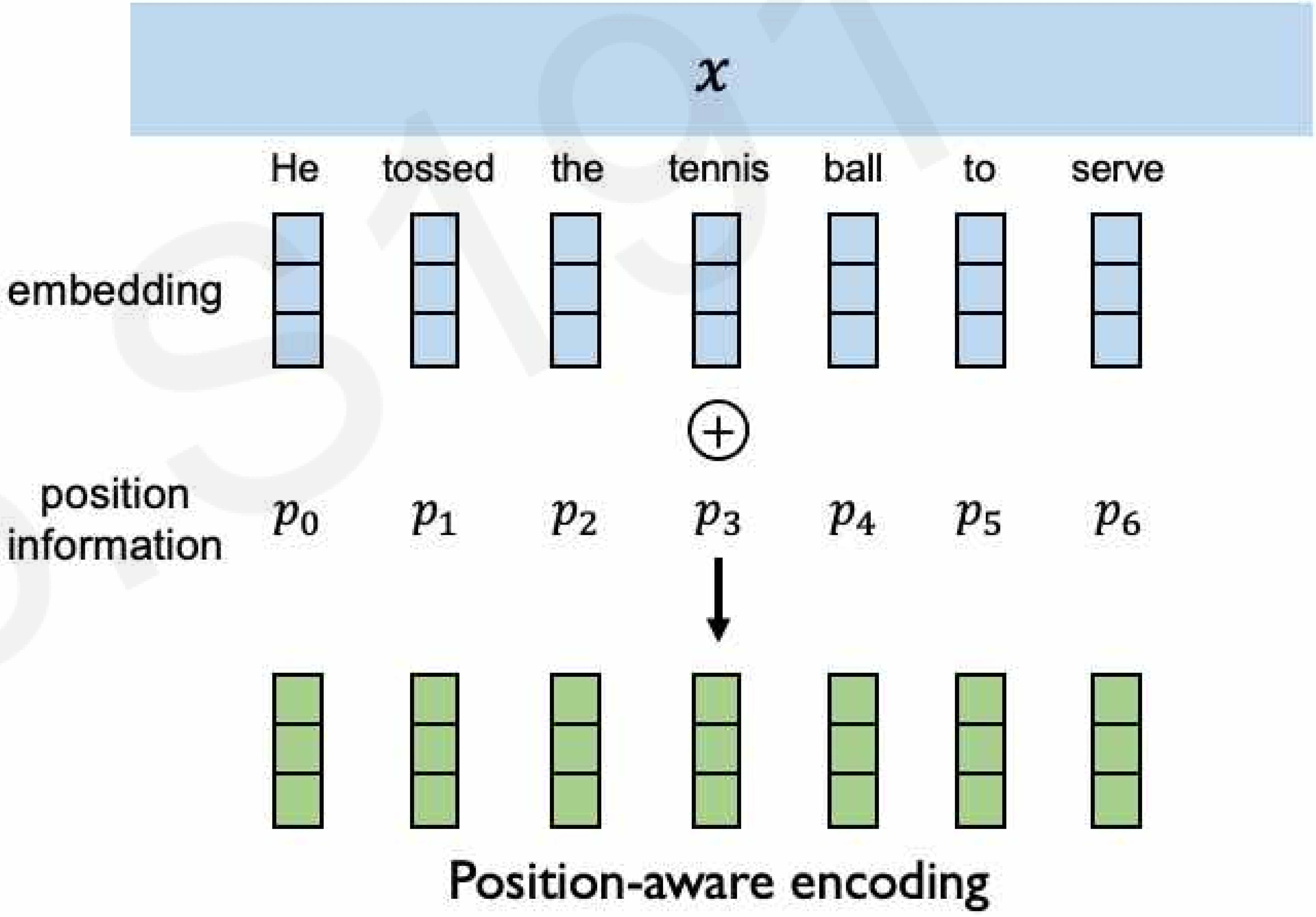
1. Encode **position** information
2. Extract **query, key, value** for search
3. Compute **attention weighting**
4. Extract **features with high attention**


Data is fed in all at once! Need to encode position information to understand order.

# Learning Self-Attention with Neural Networks

**Goal: identify and attend to most important features in input.**

1. Encode **position** information
2. Extract **query, key, value** for search
3. Compute **attention weighting**
4. Extract **features** with high attention

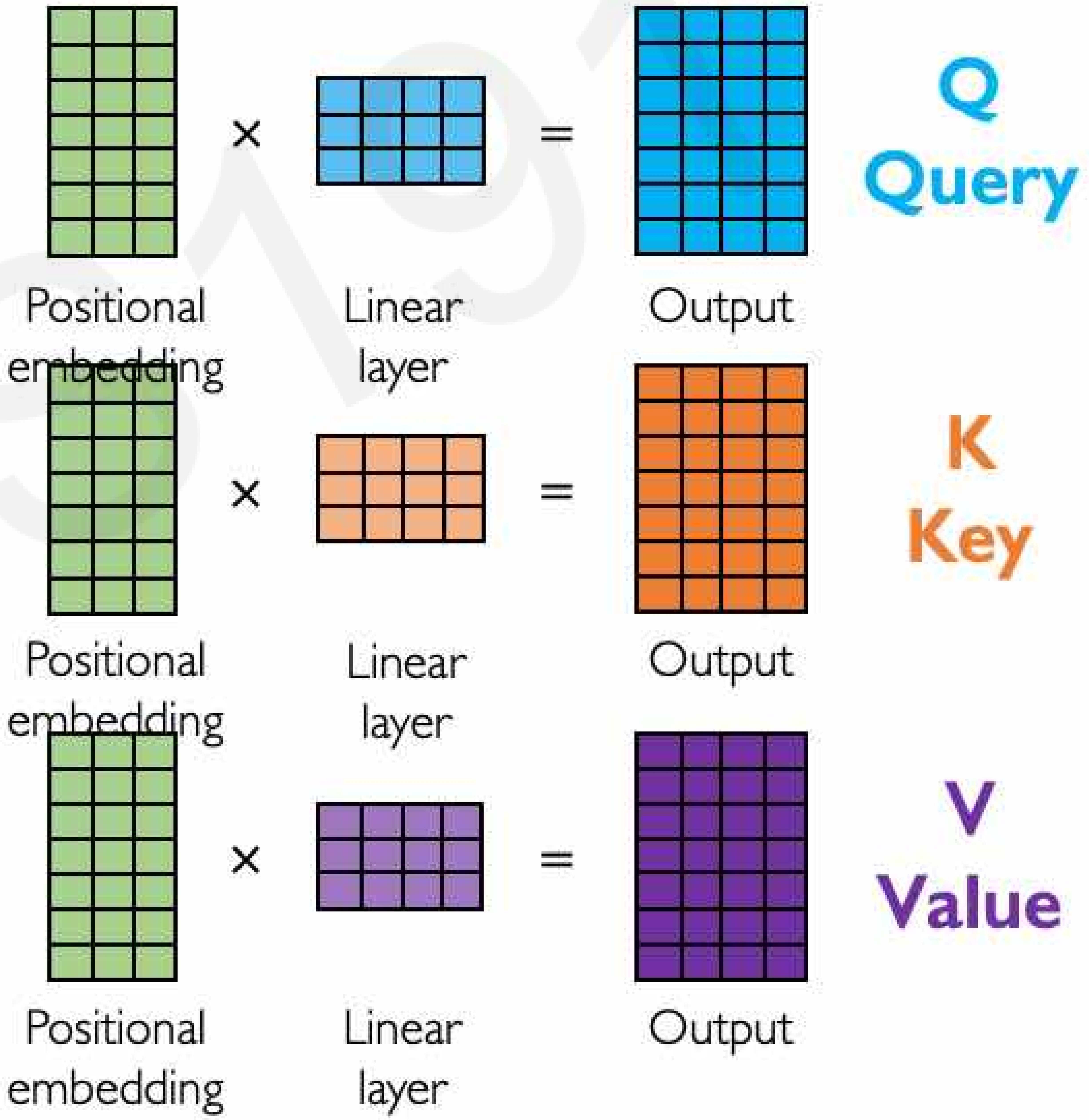



**Data is fed in all at once! Need to encode position information to understand order.**

# Learning Self-Attention with Neural Networks

**Goal: identify and attend to most important features in input.**

1. Encode **position** information
2. Extract **query, key, value** for search
3. Compute **attention weighting**
4. Extract **features** with high attention



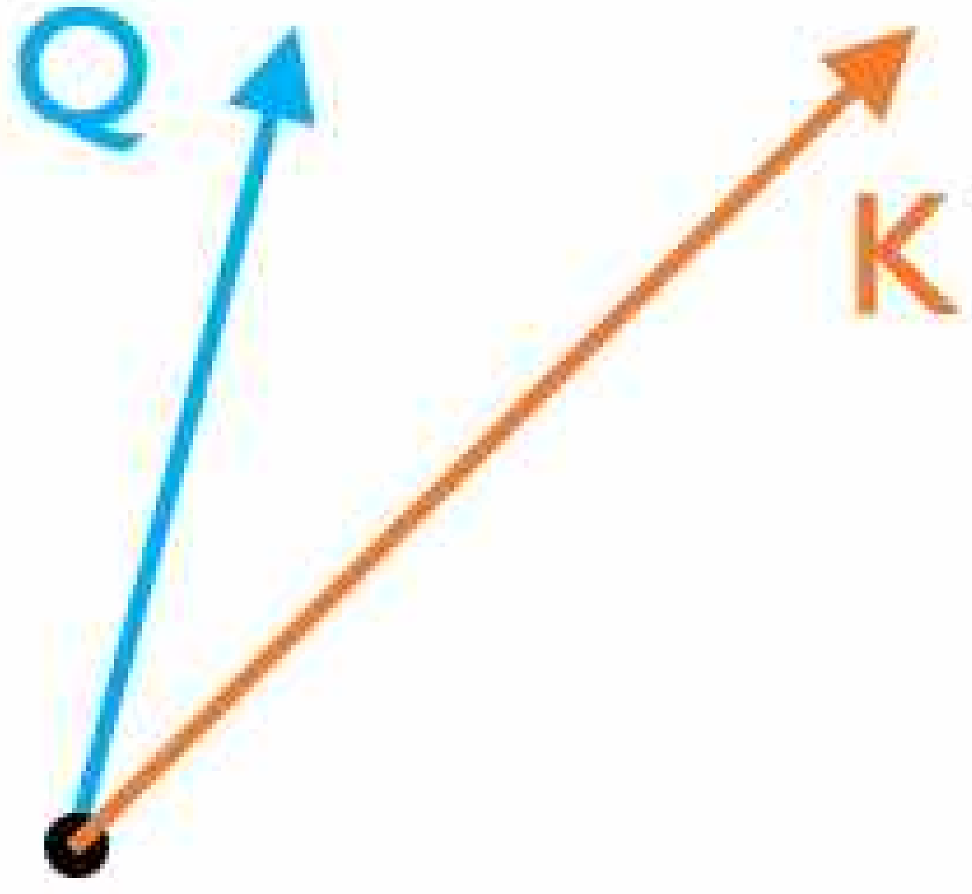
# Learning Self-Attention with Neural Networks

**Goal: identify and attend to most important features in input.**

1. Encode **position** information
2. Extract **query, key, value** for search
3. Compute **attention weighting**
4. Extract features with high attention

**Attention score:** compute pairwise similarity between each query and key

**How to compute similarity between two sets of features?**




Dot product

Scale

$$\frac{Q \cdot K^T}{\text{scaling}}$$

Similarity metric

Also known as the "cosine similarity"

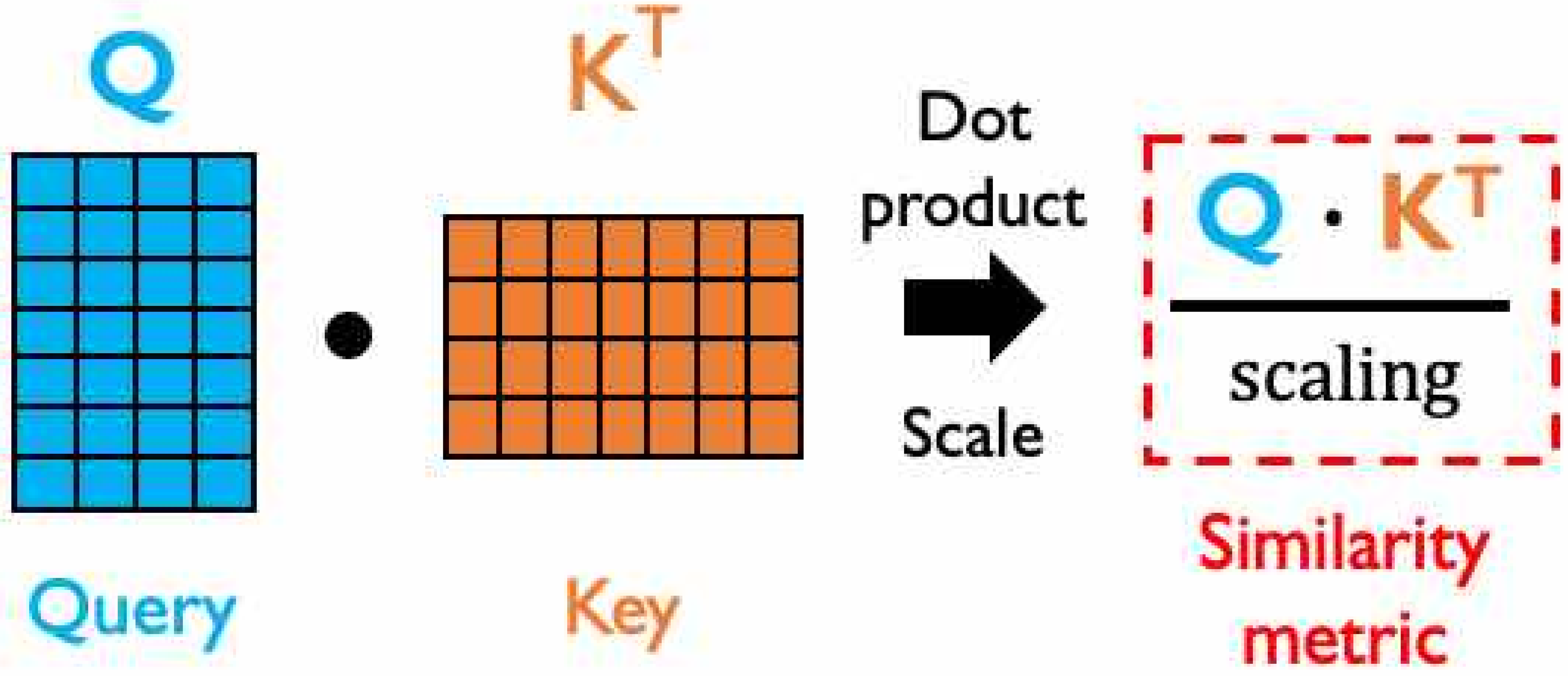
# Learning Self-Attention with Neural Networks

**Goal:** identify and attend to most important features in input.

1. Encode **position** information
2. Extract **query, key, value** for search
3. Compute **attention weighting**
4. Extract features with high attention

**Attention score:** compute pairwise similarity between each query and key

**How to compute similarity between two sets of features?**



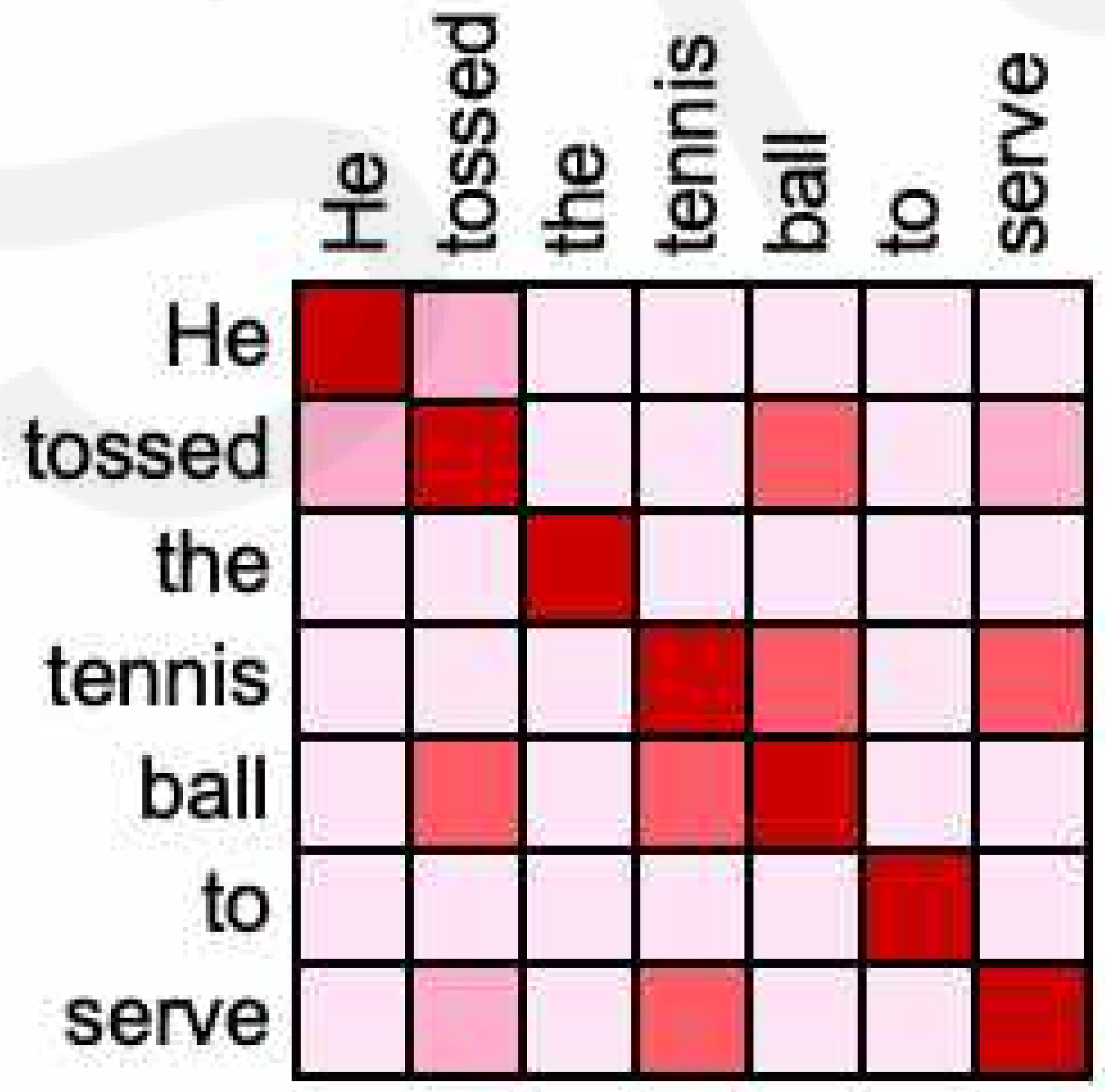

Also known as the "cosine similarity"

# Learning Self-Attention with Neural Networks

Goal: identify and attend to most important features in input.

1. Encode **position** information
2. Extract **query, key, value** for search
3. Compute **attention weighting**
4. Extract features with high attention

Attention weighting: where to attend to!
How similar is the key to the query?




$$softmax\left(\frac{Q \cdot K^T}{scaling}\right)$$

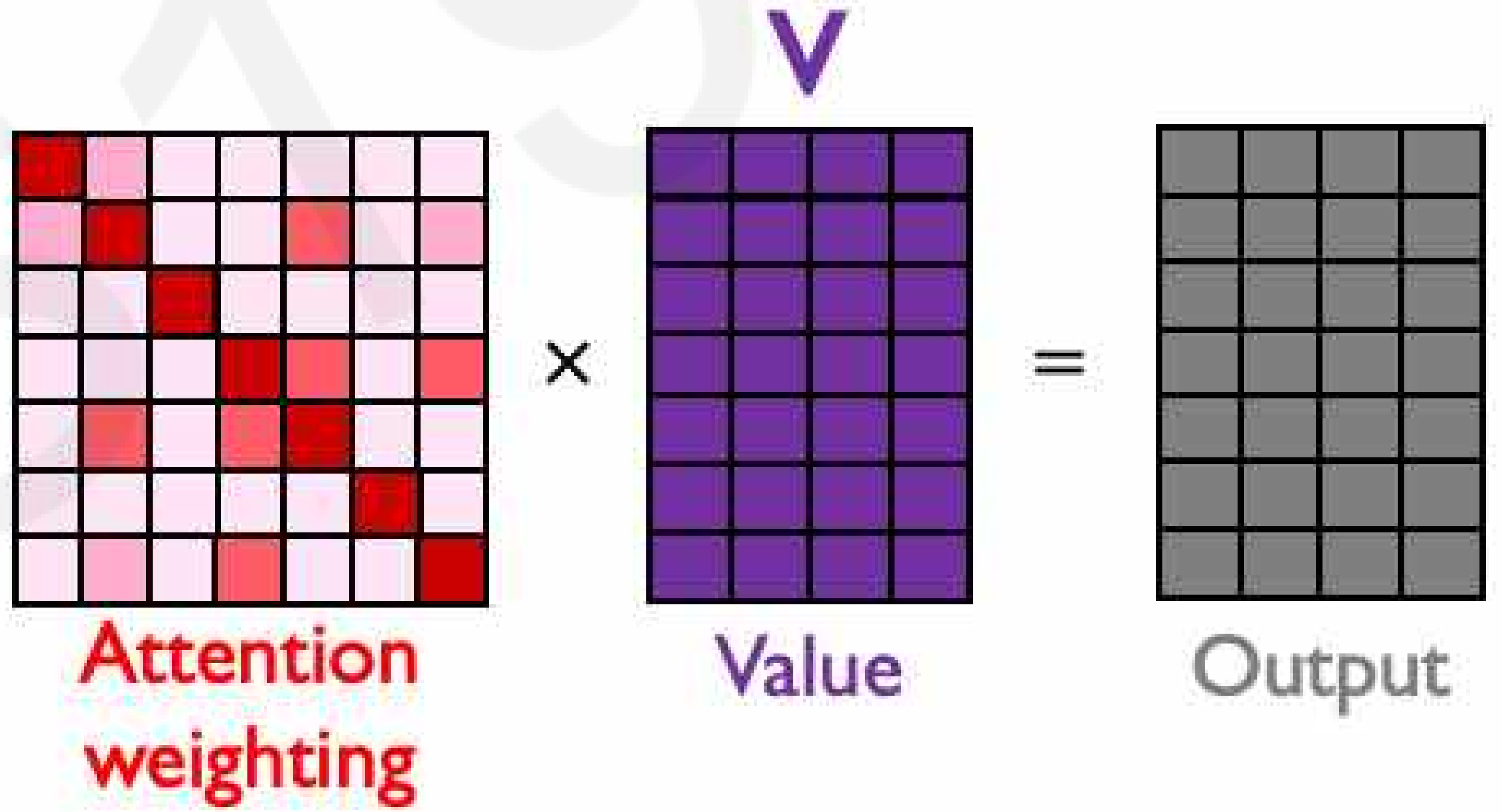
Attention weighting

# Learning Self-Attention with Neural Networks

Goal: identify and attend to most important features in input.

1. Encode **position** information
2. Extract **query, key, value** for search
3. Compute **attention weighting**
4. Extract **features with high attention**

Last step: self-attend to extract features



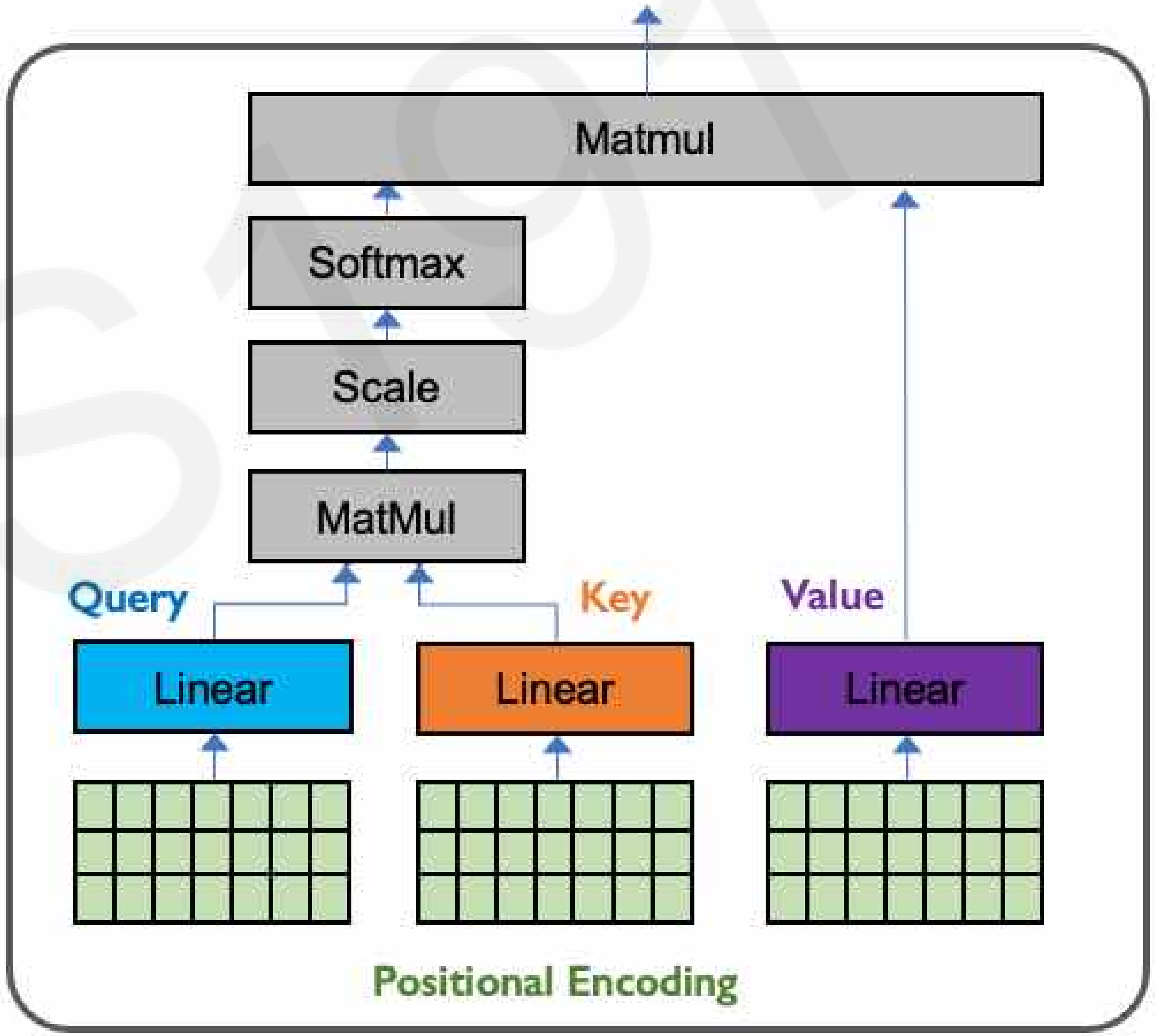

$$\underline{softmax\left(\frac{Q \cdot K^T}{scaling}\right)} \cdot \underline{V} = \underline{A(Q,K,V)}$$

# Learning Self-Attention with Neural Networks

**Goal: identify and attend to most important features in input.**

1. Encode **position** information
2. Extract **query, key, value** for search
3. Compute **attention weighting**
4. Extract **features with high attention**

**These operations form a self-attention head that can plug into a larger network. Each head attends to a different part of input.**




$$softmax\left(\frac{\boxed{Q \cdot K^T}}{scaling}\right) \cdot V$$

# Applying Multiple Self-Attention Heads

Attention weighting

×

Value

=

Output

Output of attention head 1

Output of attention head 2

Output of attention head 3

# Self-Attention Applied

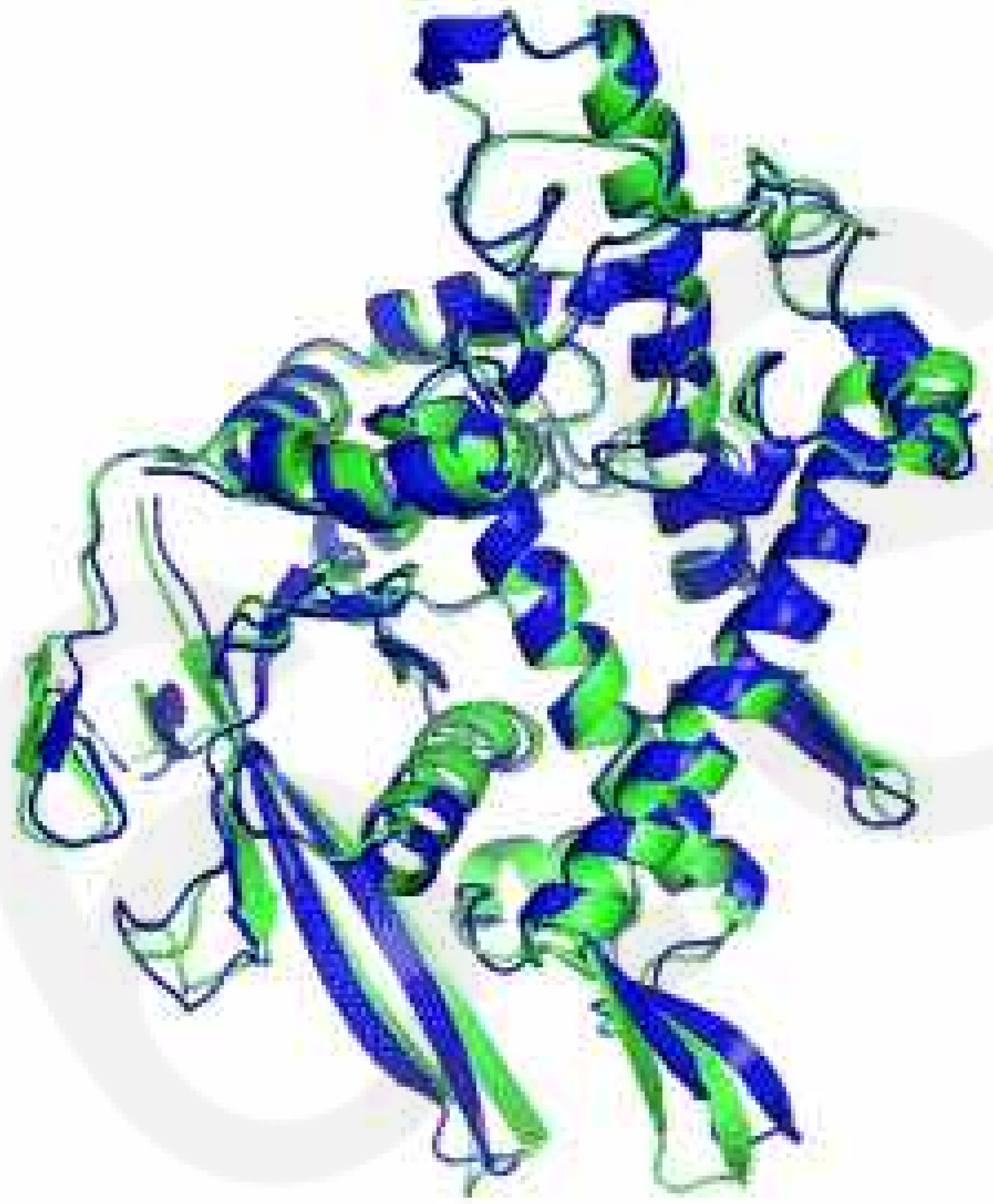
## Language Processing

An armchair in the shape of an avocado

BERT, GPT-3

Devlin et al., *NAACL* 2019
Brown et al., *NeurIPS* 2020

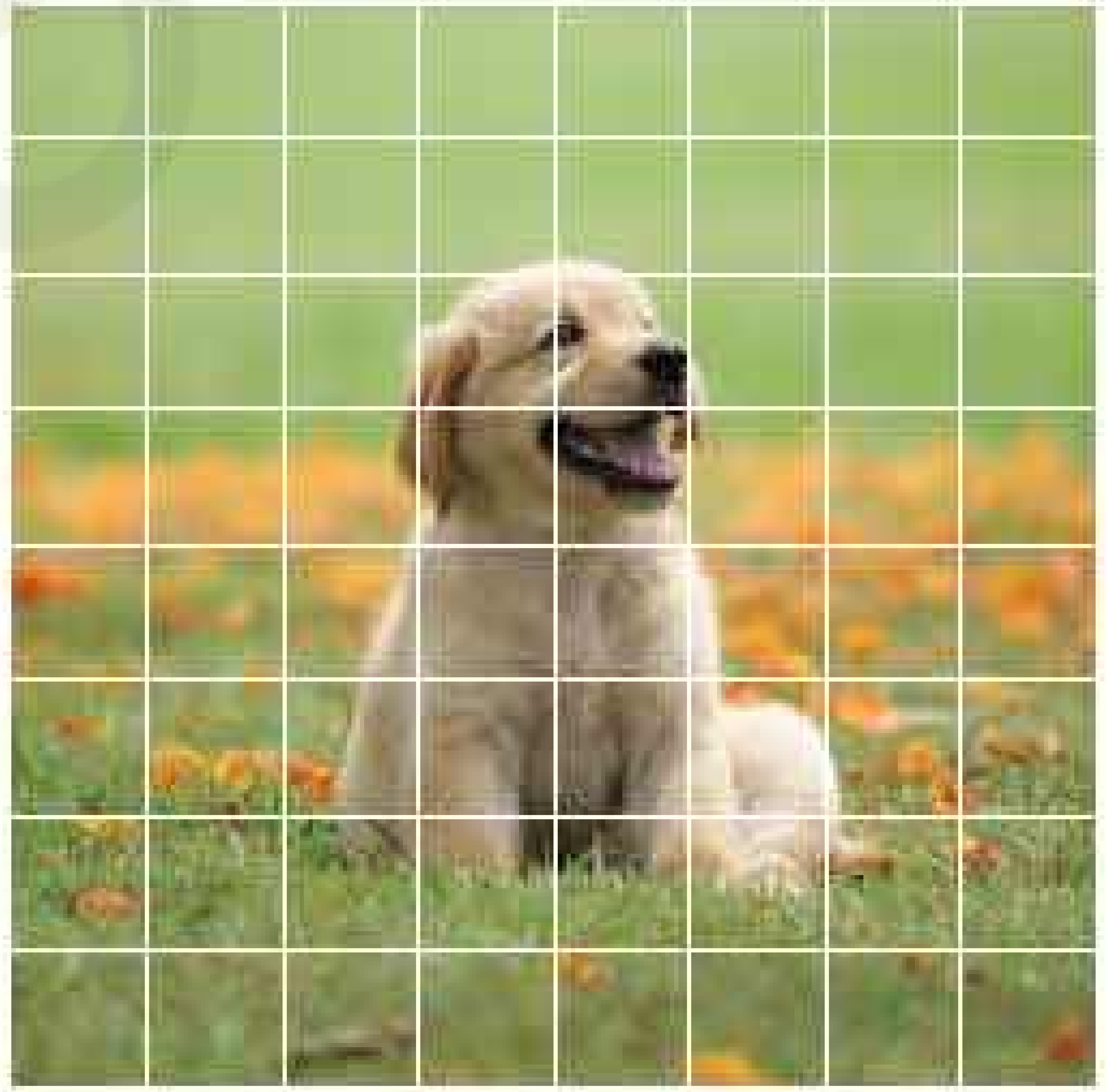
## Biological Sequences



AlphaFold2

Jumper et al., *Nature* 2021

## Computer Vision



Vision Transformers

Dosovitskiy et al., *ICLR* 2020

# Deep Learning for Sequence Modeling: Summary

1. RNNs are well suited for **sequence modeling** tasks
2. Model sequences via a **recurrence relation**
3. Training RNNs with **backpropagation through time**
4. Models for **music generation**, classification, machine translation, and more
5. Self-attention to model **sequences without recurrence**

# 6.S191: Introduction to Deep Learning

## Lab 1: Introduction to TensorFlow and Music Generation with RNNs

Link to download labs:
http://introtodeeplearning.com#schedule

1. Open the lab in Google Colab
2. Start executing code blocks and filling in the #TODOs
3. Need help? Come to the class Gather.Town!