# CAP 781
# MACHINE LEARNING

Tanzeela Javid Kaloo (32638)
Assistant Professor
System And Architecture
Lovely Professional University

# UNIT – V
# Neural Network and Deep Learning

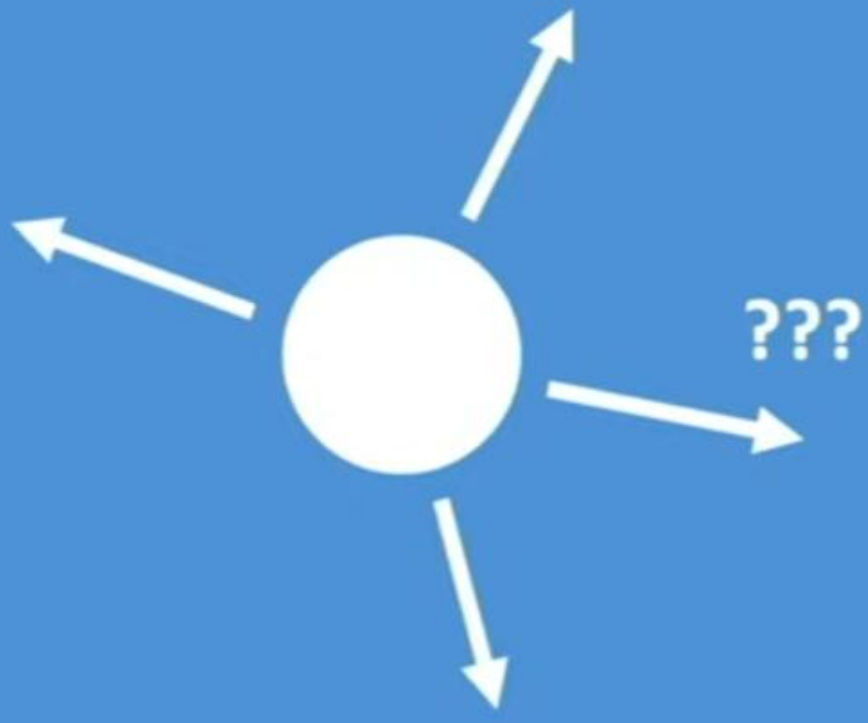
Tanzeela Javid Kaloo (32638)
Assistant Professor
System And Architecture
Lovely Professional University

# Content

- Introduction to Neural Networks,
- Perceptron and Multilayer Perceptron (MLPs),
- Activation Functions: Sigmoid, ReLU, and others,
- Convolutional Neural Networks (CNNs) for Image Recognition,
- Recurrent Neural Networks (RNNs) for Sequence Data,
- Transfer Learning and Pre-trained Models

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

Audio

# Sequences in the Wild

6.S191 Introduction to Deep Learning

Text

**character:** 6 . S I 9 I

**word:** Introduction to Deep Learning

Text

**character:** 6 . S I 9 I

**word:** Introduct  p Learning

6 . S I 9 I

**word:** Introduct p Learning


p 08, 2019

# Sequence Modeling Applications


One to One
**Binary Classification**

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

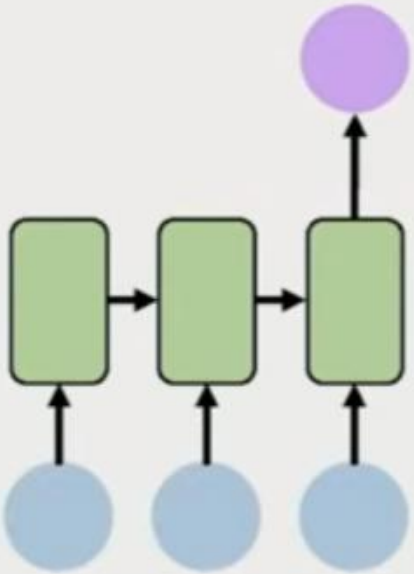
# Sequence Modeling Applications


One to One

**Binary Classification**

"Will I pass this class?"

Student → Pass?



Many to One

**Sentiment Classification**

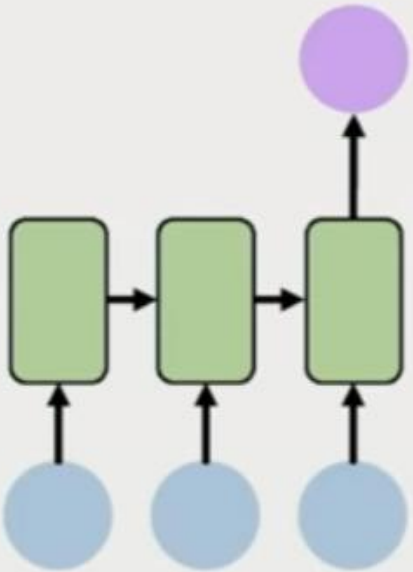
# Sequence Modeling Applications


One to One

**Binary Classification**

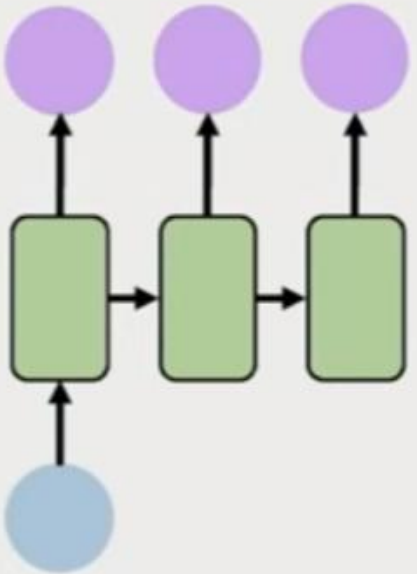
"Will I pass this class?"

Student → Pass?



Many to One

**Sentiment Classification**

One to Many

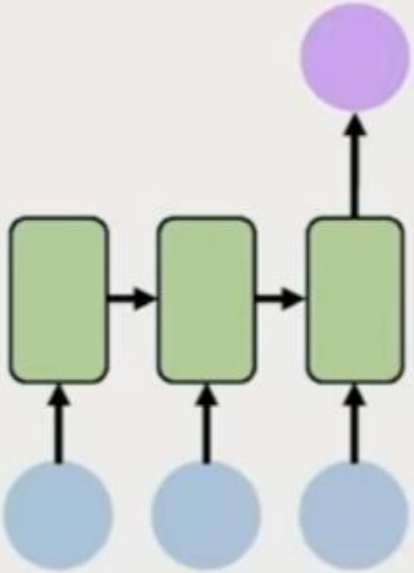
**Image Captioning**

"A baseball player throws a ball."

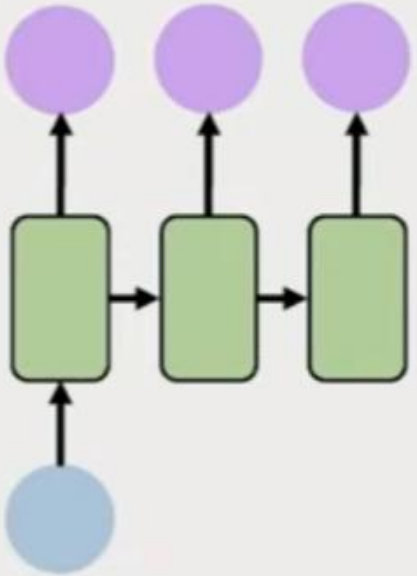
# Sequence Modeling Applications


One to One
**Binary Classification**

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

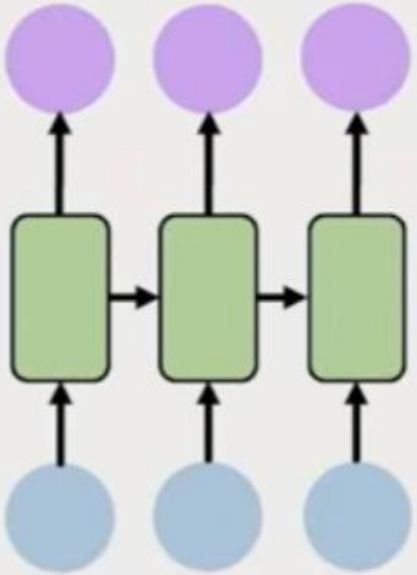


Many to One
**Sentiment Classification**

One to Many
**Image Captioning**

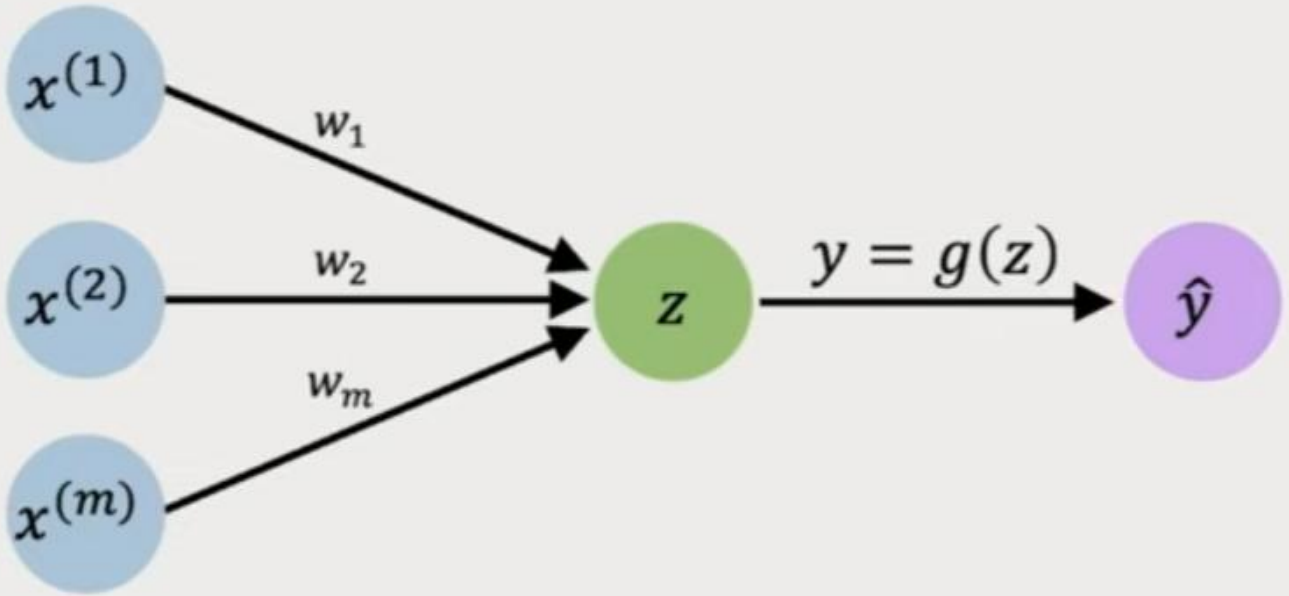
"A baseball player throws a ball."



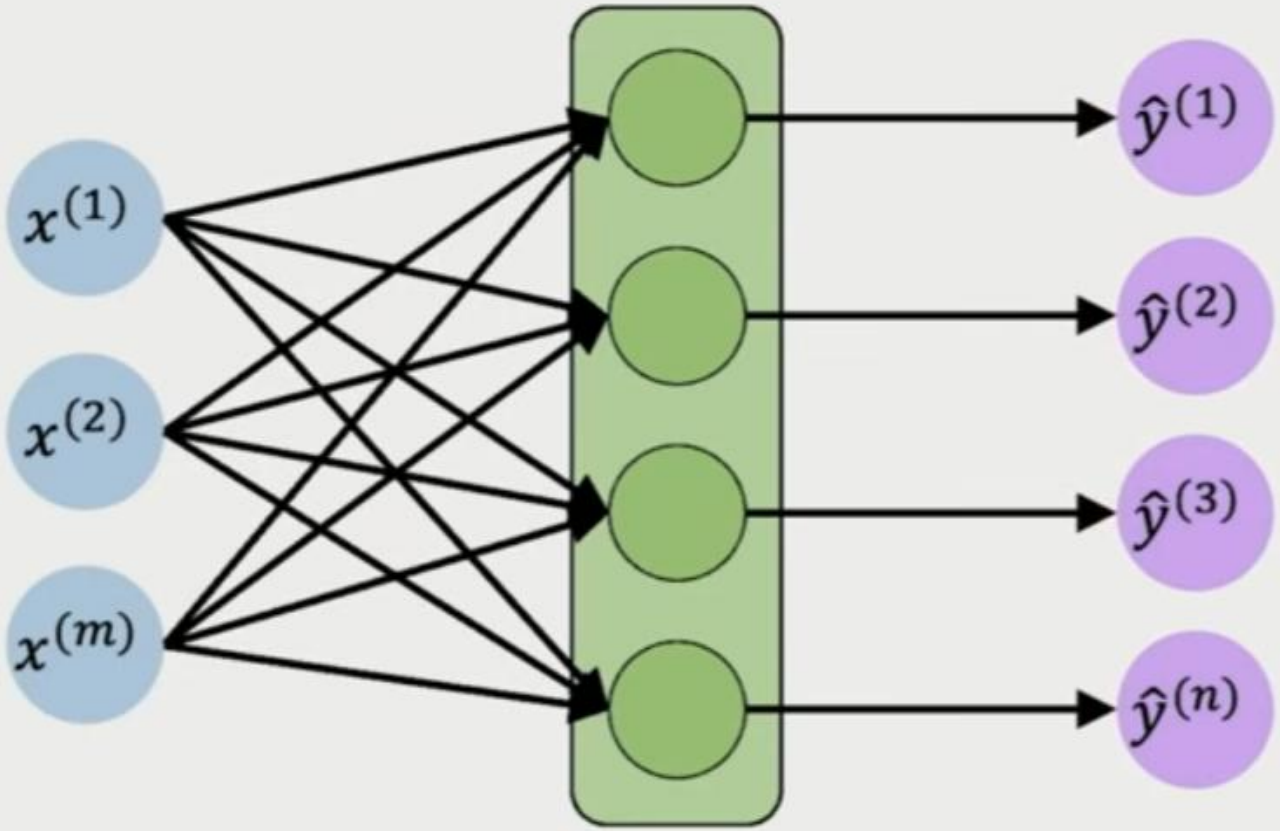
Many to Many
**Machine Translation**

# Neurons with Recurrence

# The Perceptron Revisited



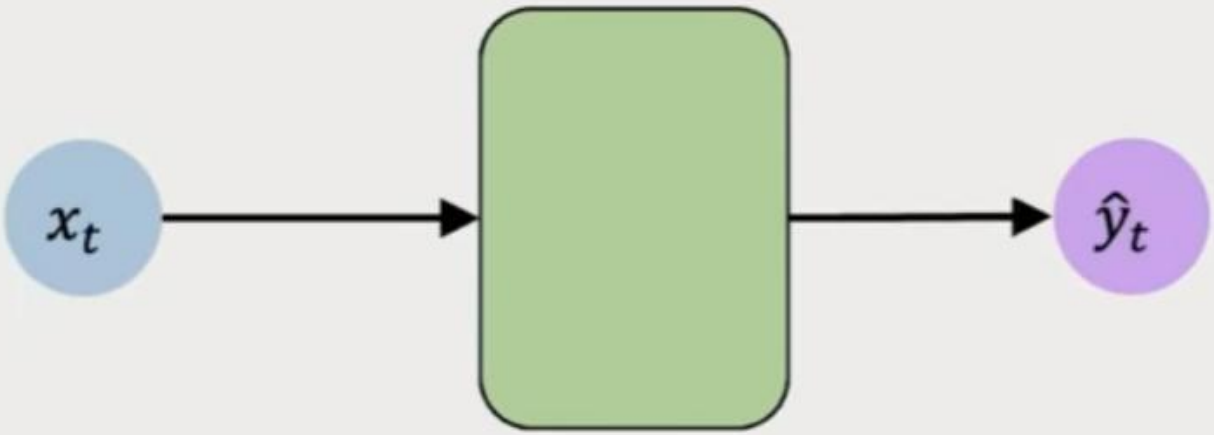
# Feed-Forward Networks Revisited




$\boldsymbol{x} \in \mathbb{R}^m$

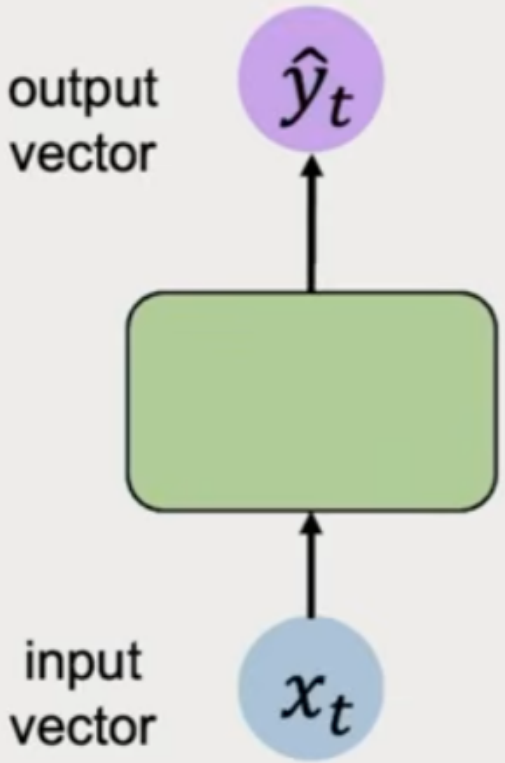
$\hat{\boldsymbol{y}} \in \mathbb{R}^n$

# Feed-Forward Networks Revisited

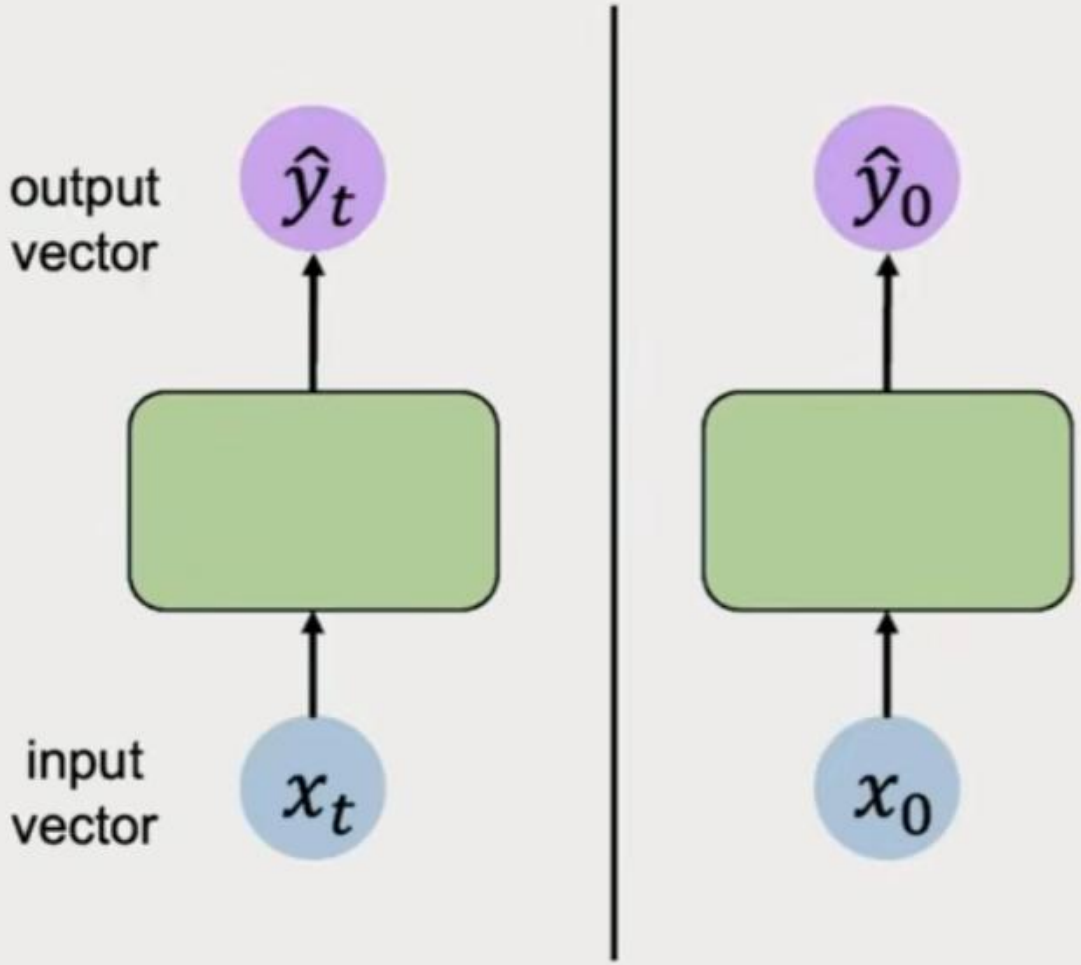



$$\boldsymbol{x_t} \in \mathbb{R}^m \qquad \hat{\boldsymbol{y}}_t \in \mathbb{R}^n$$

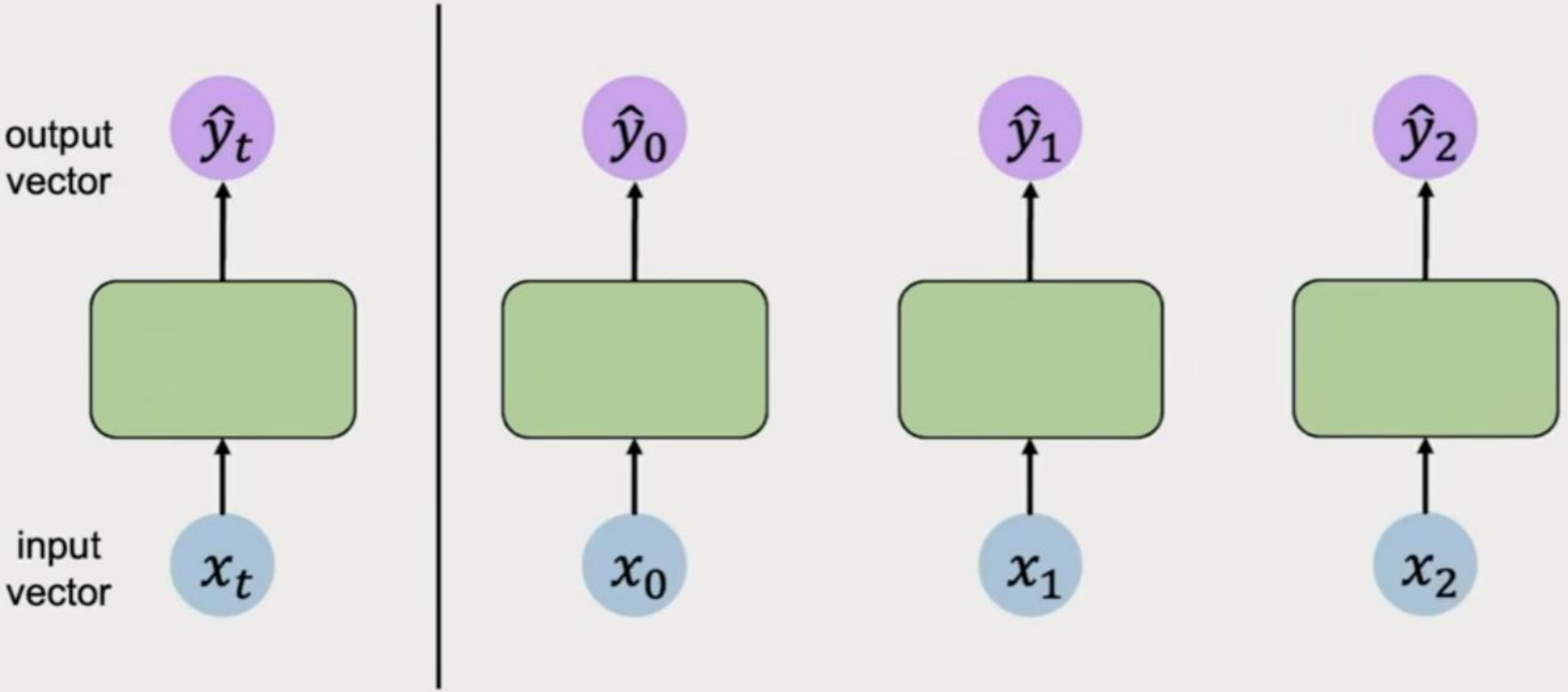
# Handling Individual Time Steps



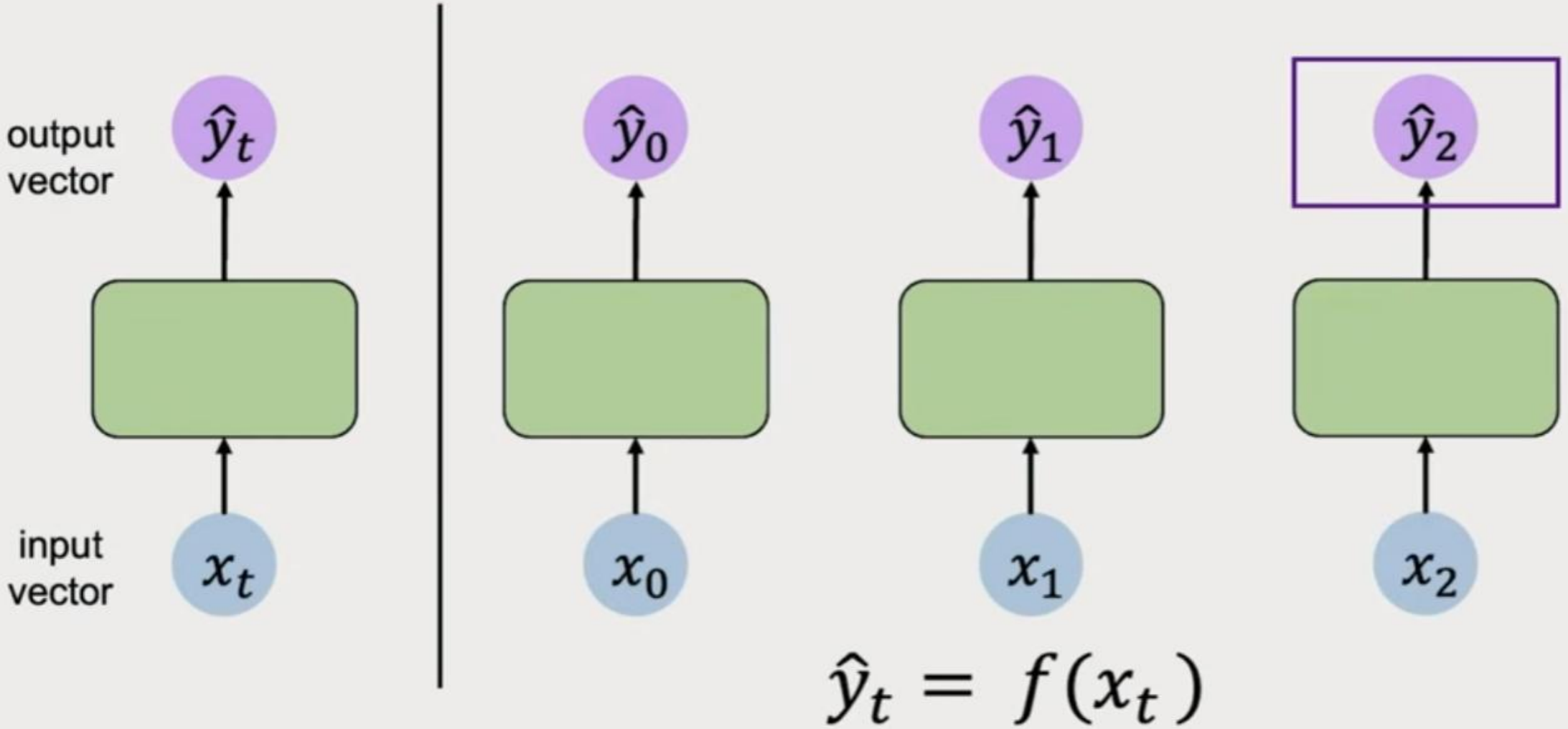
# Handling Individual Time Steps



# Handling Individual Time Steps

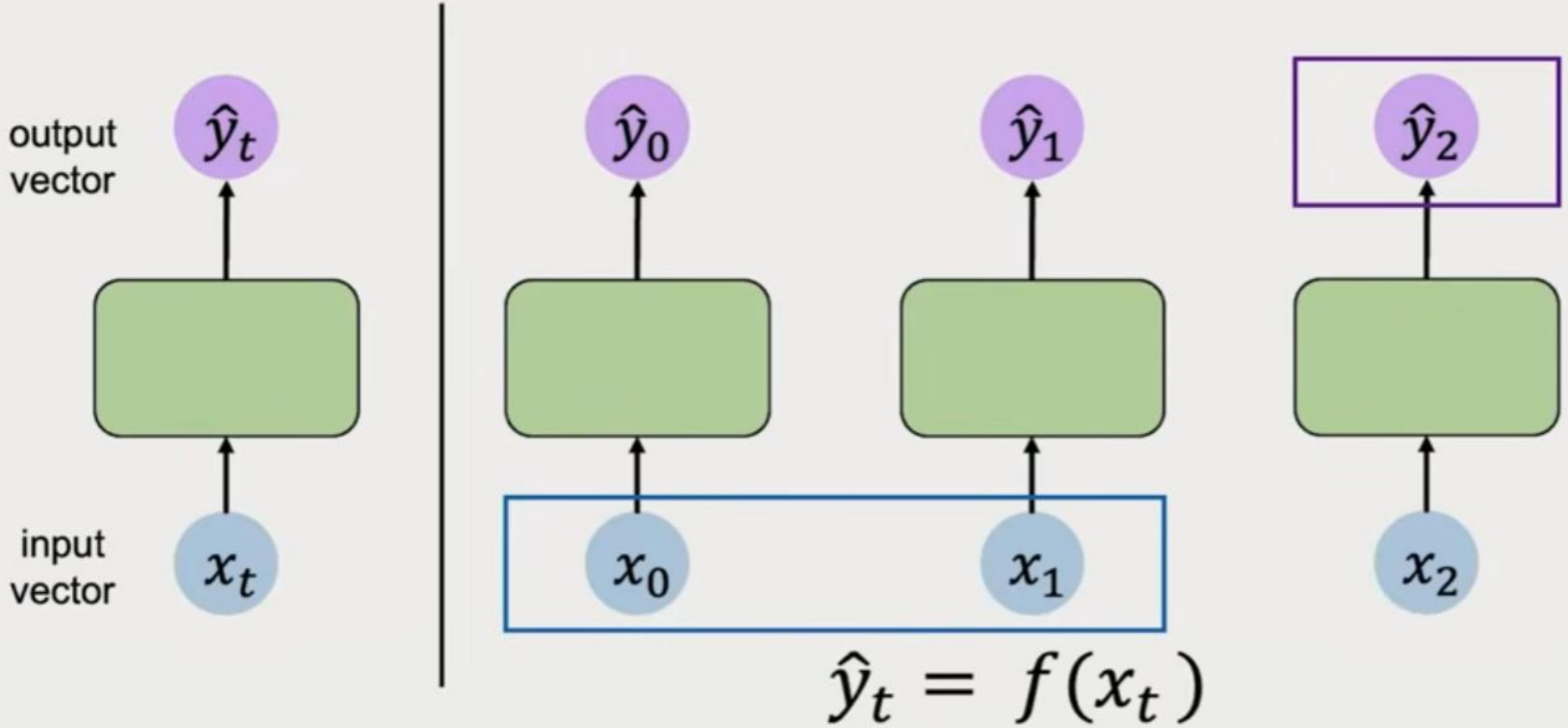


# Handling Individual Time Steps

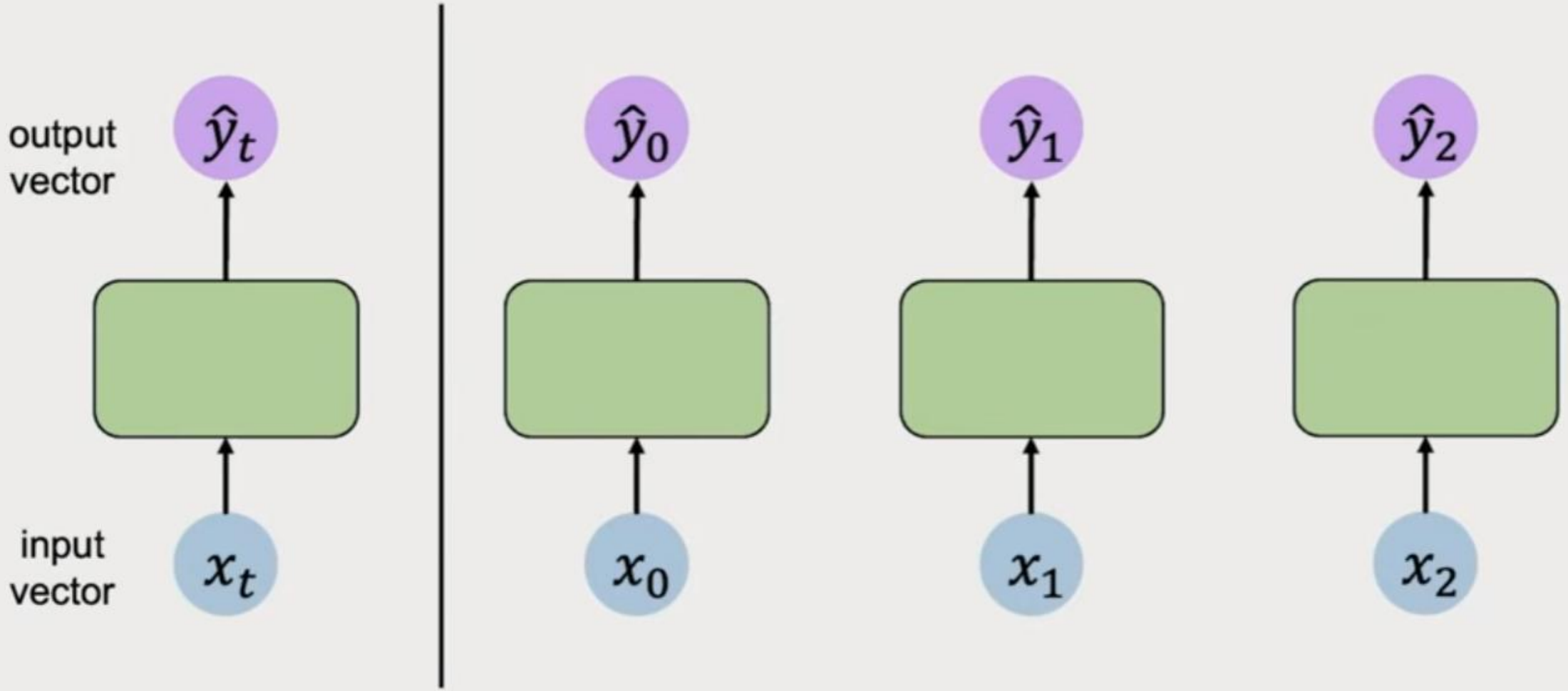



$$\hat{y}_t = f(x_t)$$

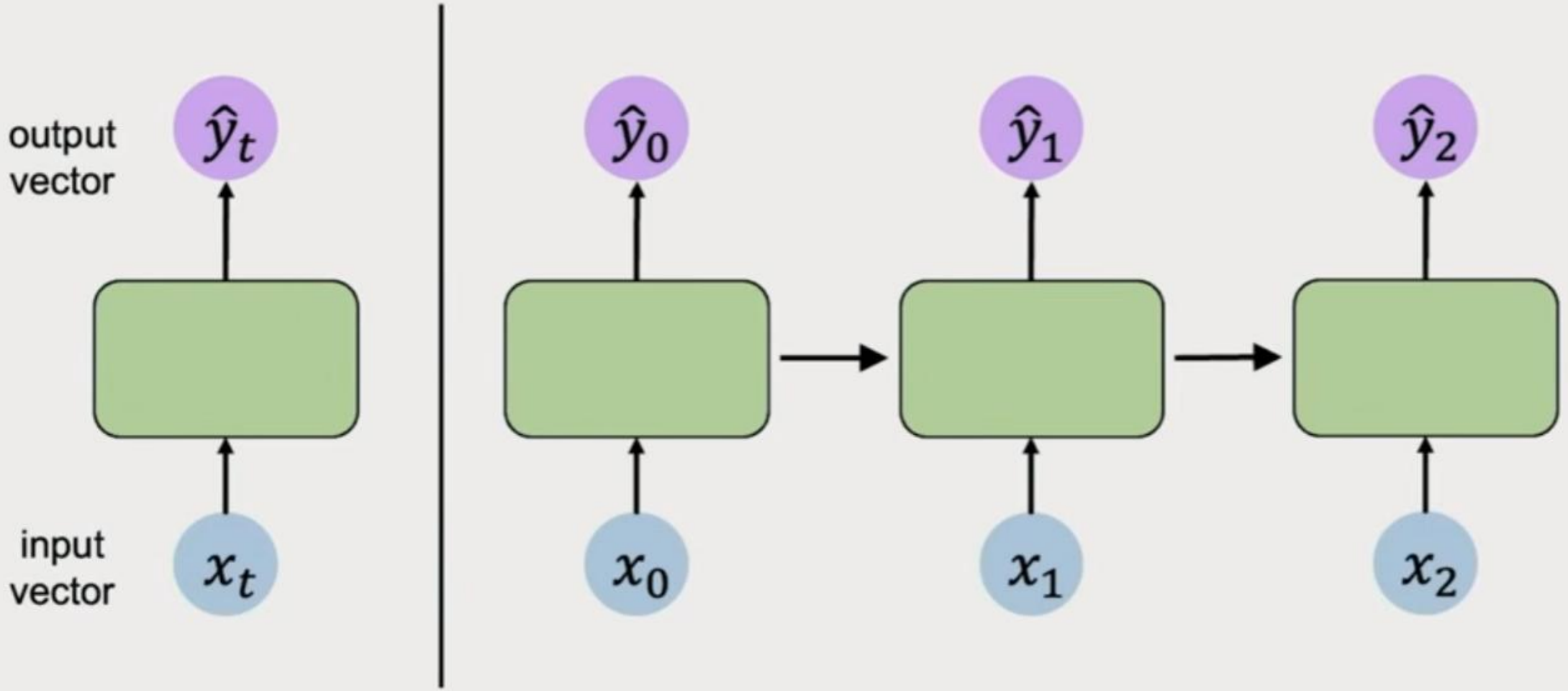
# Handling Individual Time Steps



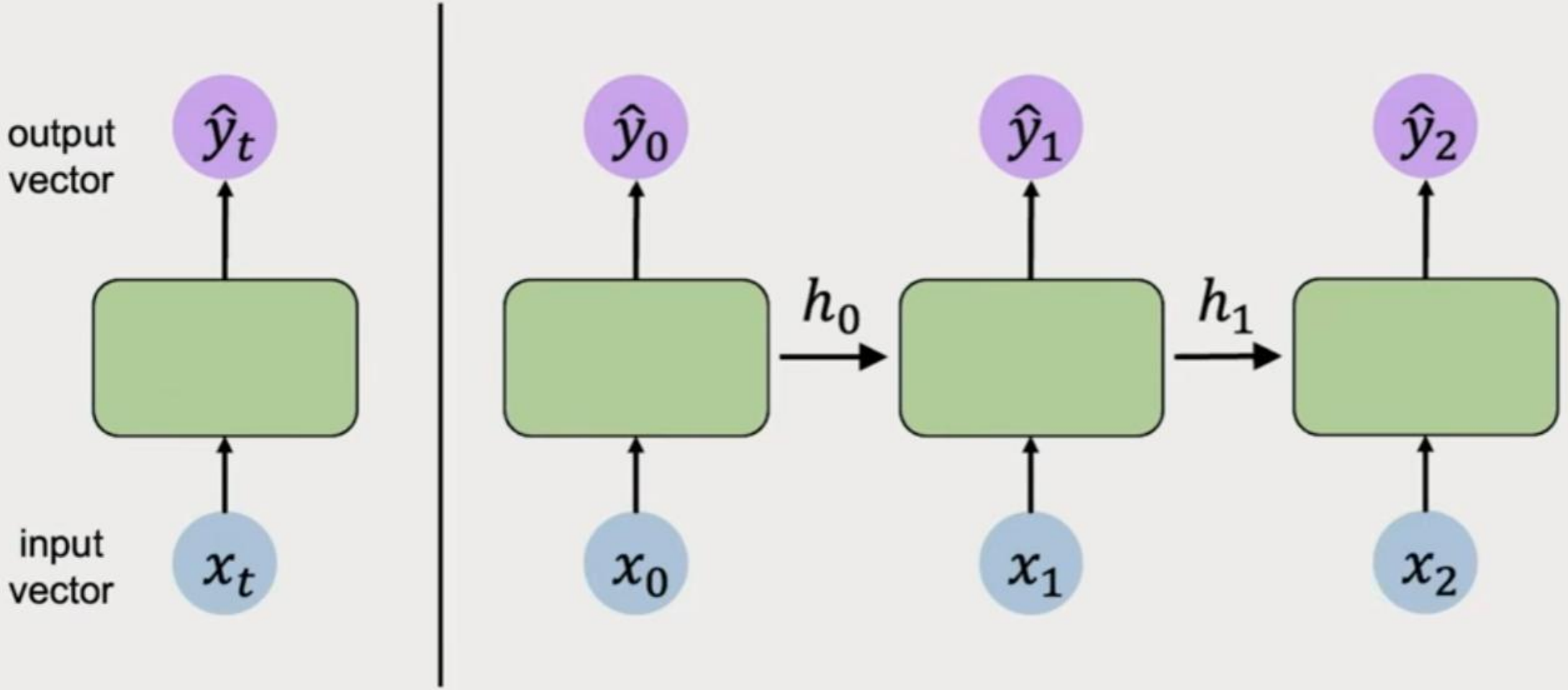
# Neurons with Recurrence



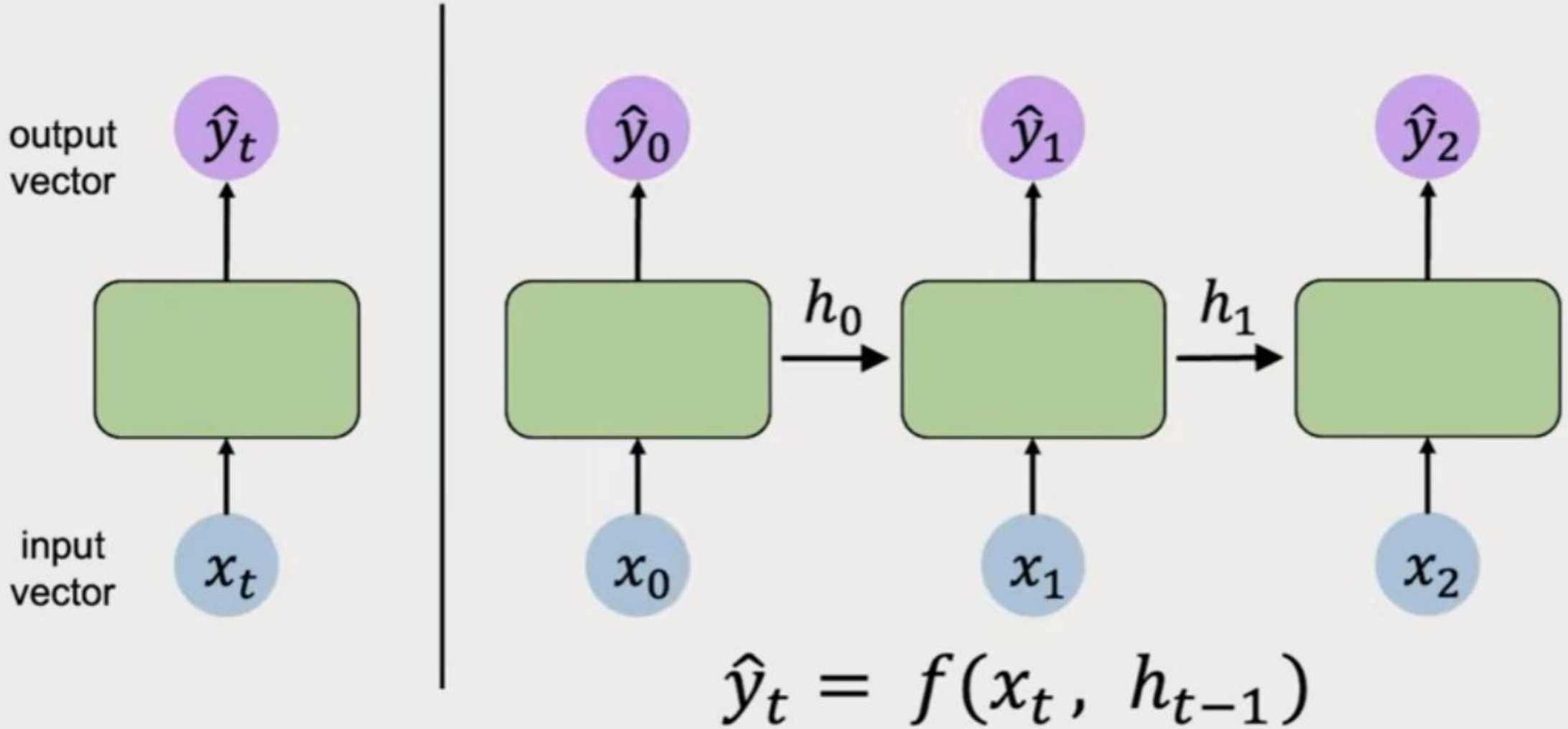
# Neurons with Recurrence



# Neurons with Recurrence

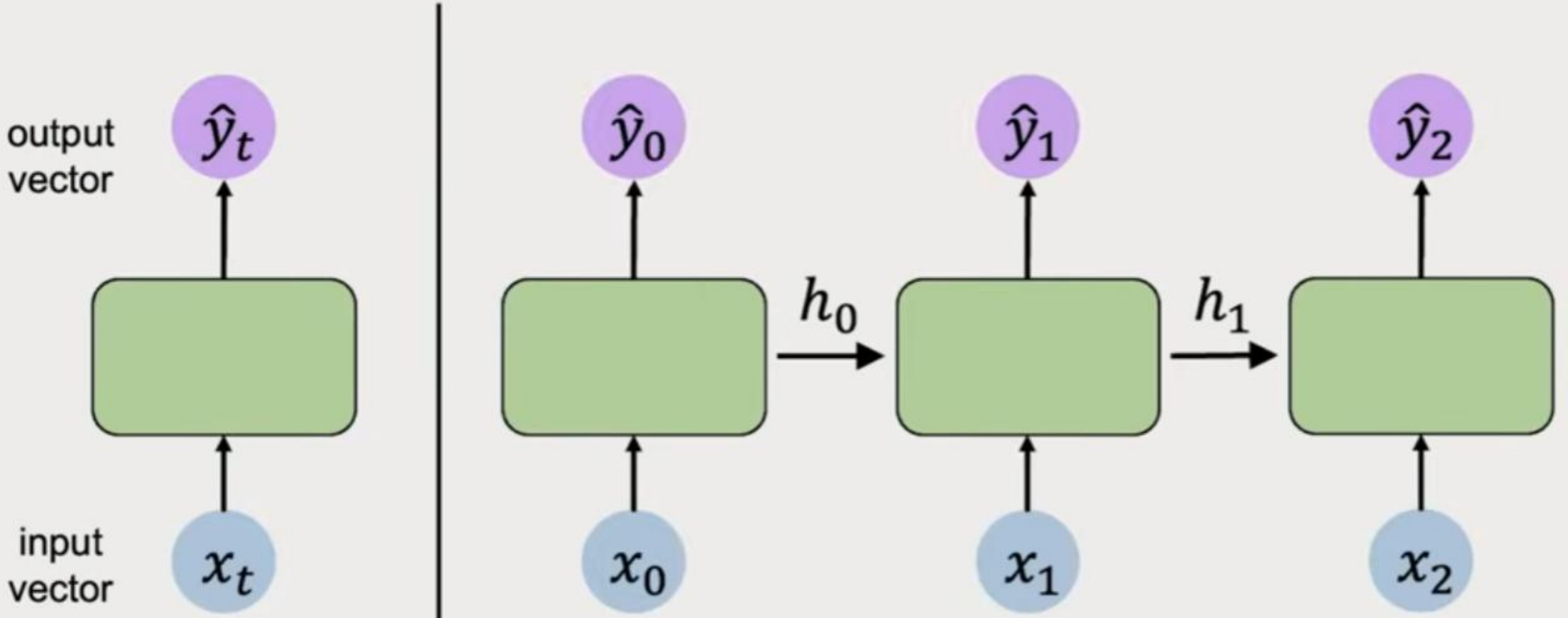


# Neurons with Recurrence




$$\hat{y}_t = f(x_t, h_{t-1})$$

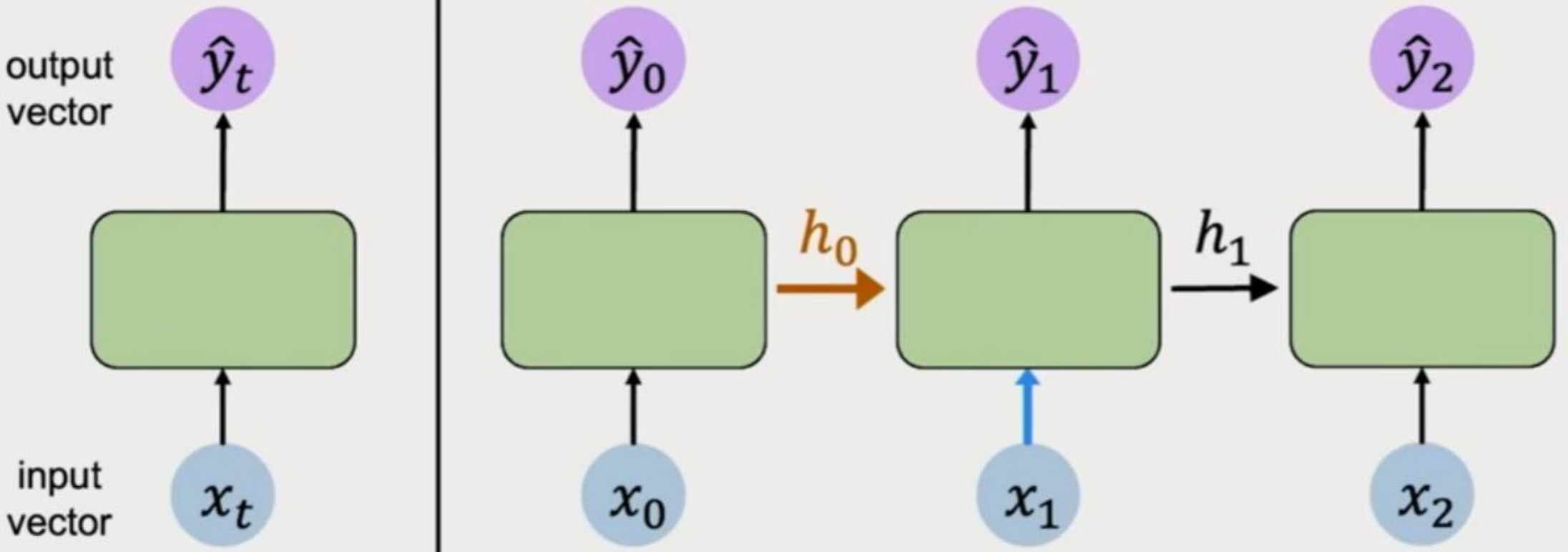
# Neurons with Recurrence




$$\underline{\hat{y}_t} = f(x_t, h_{t-1})$$

output

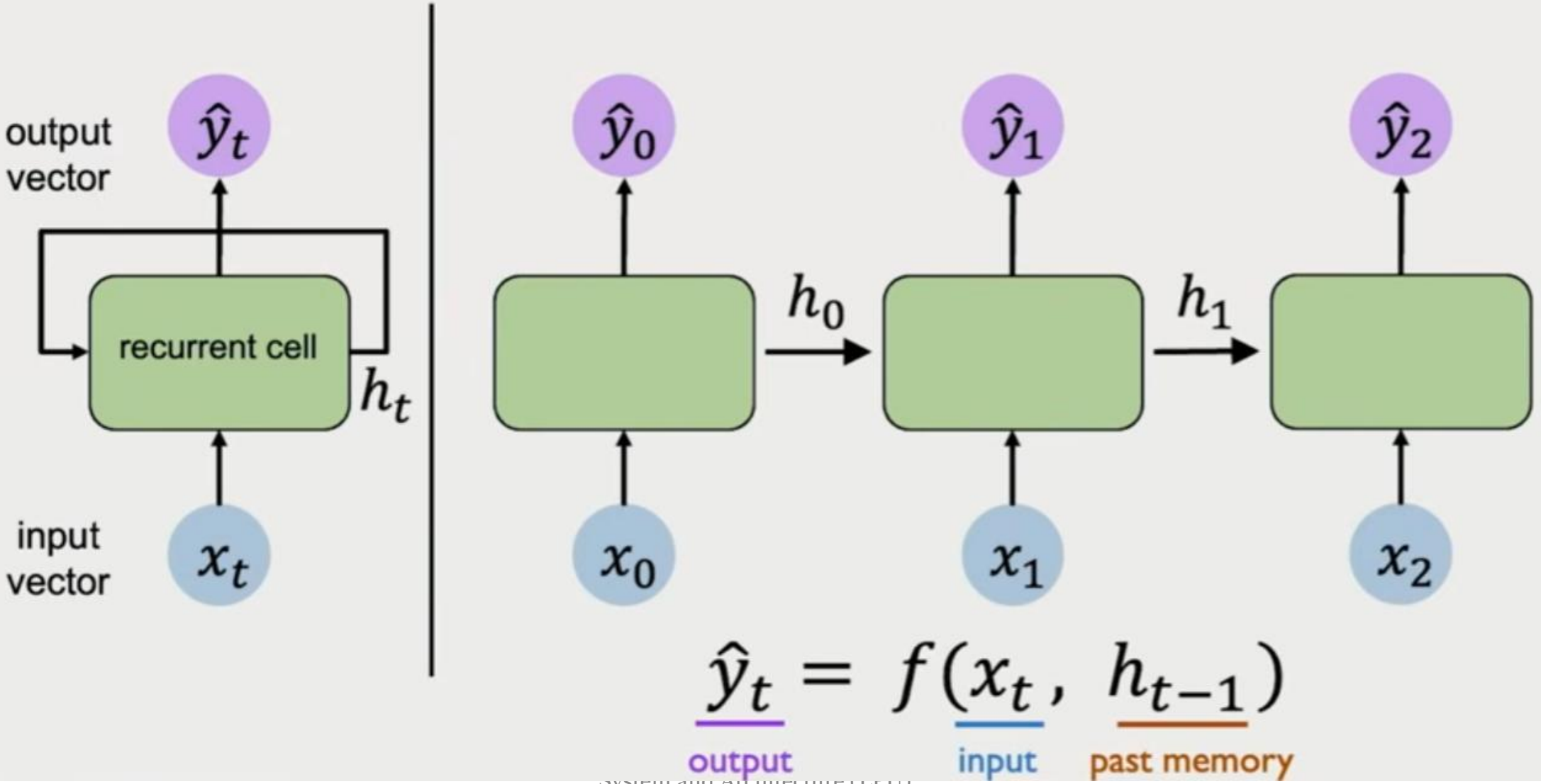
# Neurons with Recurrence




$$\underline{\hat{y}_t} = f(\underline{x_t}, \underline{h_{t-1}})$$

output input past memory

# Neurons with Recurrence

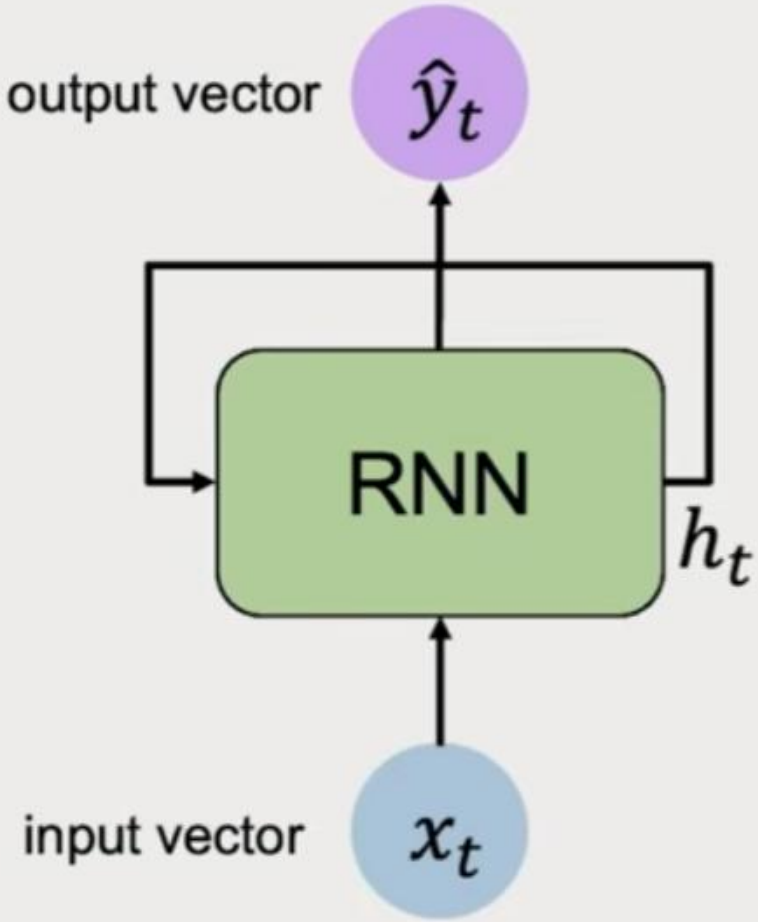



$$\hat{y}_t = f(x_t, h_{t-1})$$

output input past memory

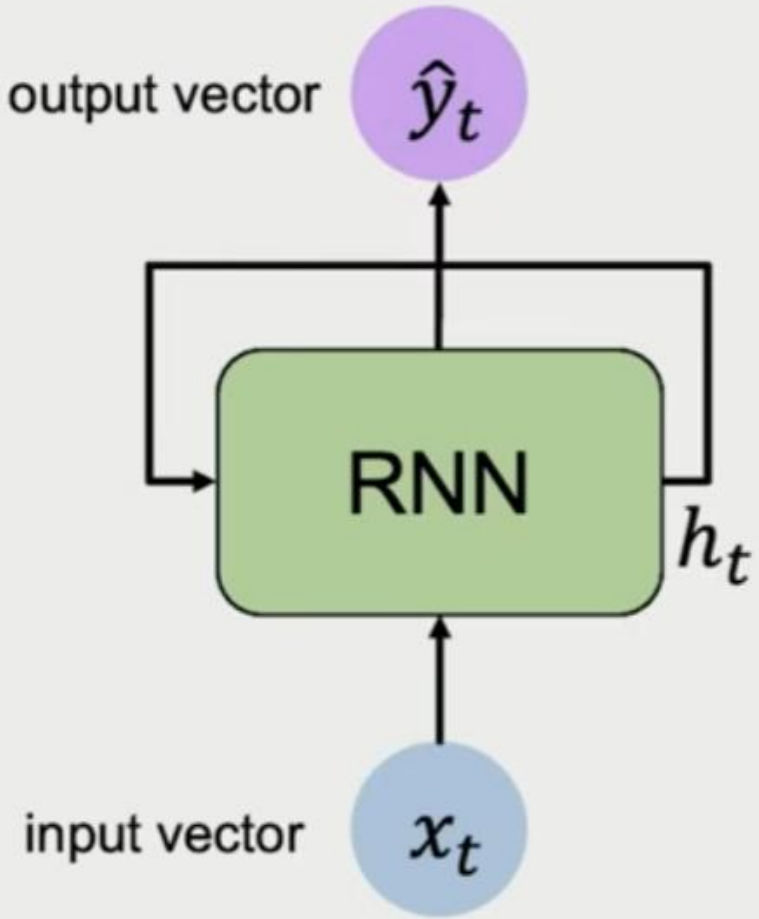
# Recurrent Neural Networks (RNNs)

# Recurrent Neural Networks (RNNs)




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

# Recurrent Neural Networks (RNNs)

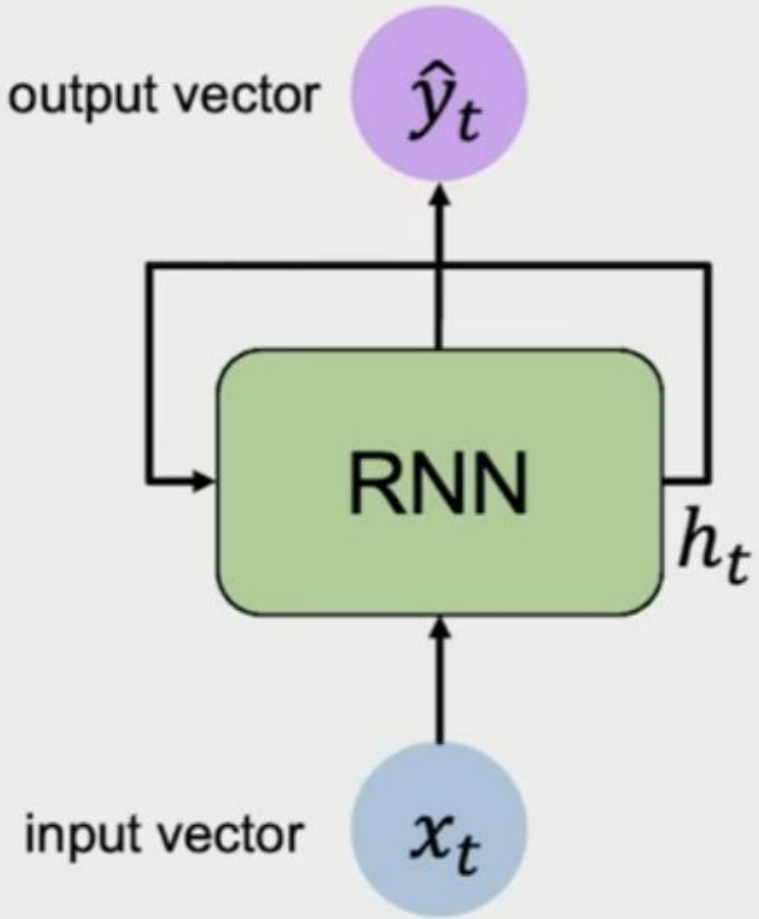



Apply a **recurrence relation** at every time step to process a sequence:

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

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

# Recurrent Neural Networks (RNNs)

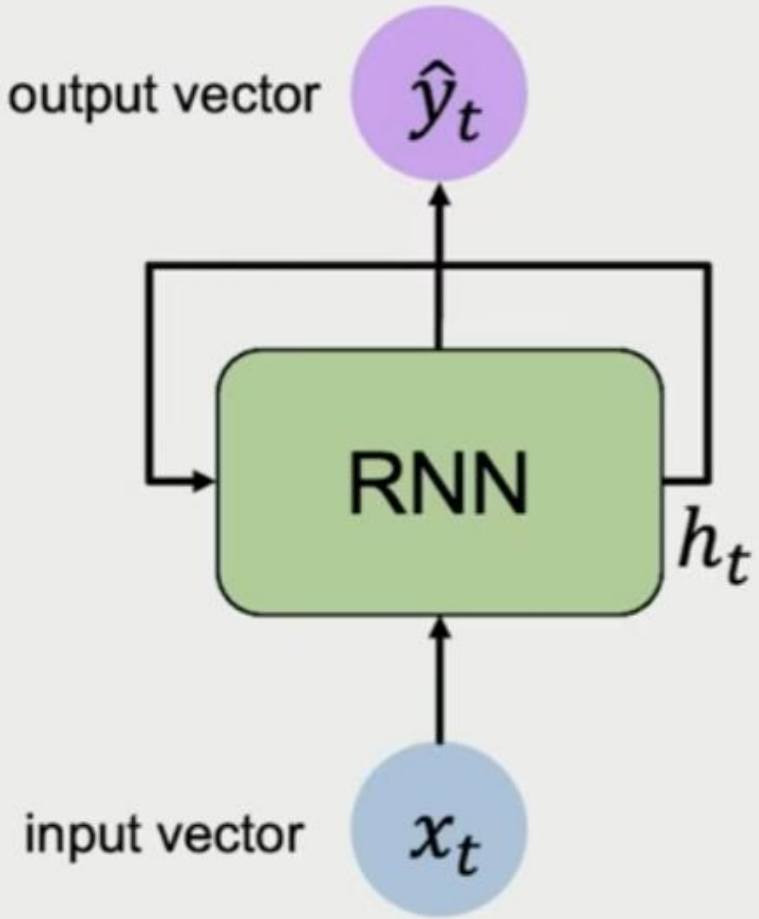



Apply a **recurrence relation** at every time step to process a sequence:

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

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

# Recurrent Neural Networks (RNNs)

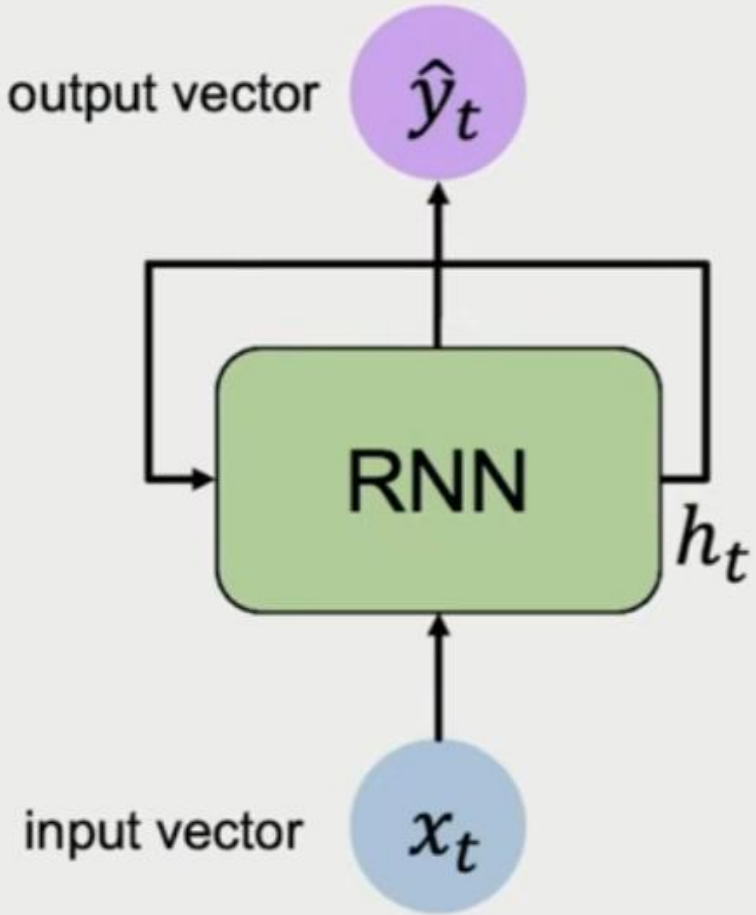



Apply a **recurrence relation** at every time step to process a sequence:

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

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

# Recurrent Neural Networks (RNNs)



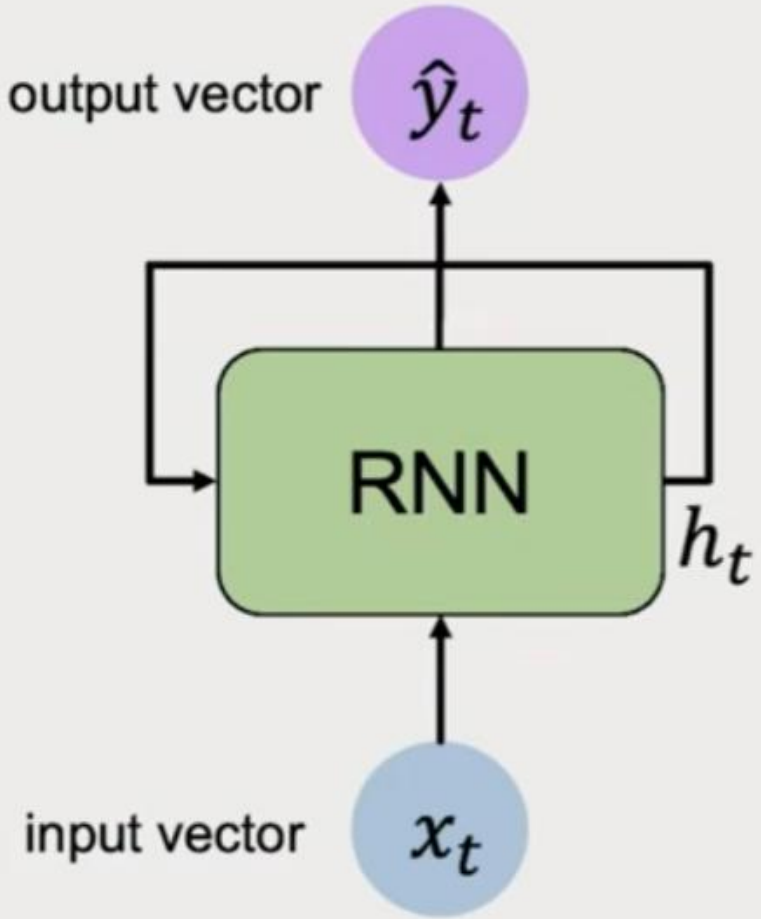

Apply a **recurrence relation** at every time step to process a sequence:

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

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

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

# Recurrent Neural Networks (RNNs)




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

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

cell state

function with weights W

input

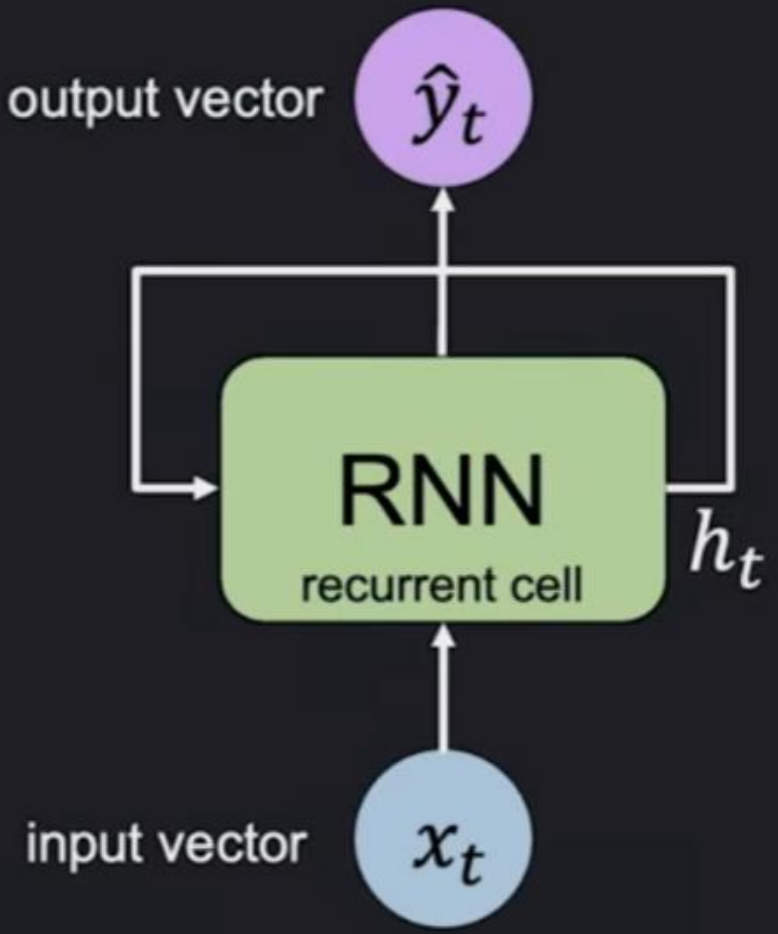
old state

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

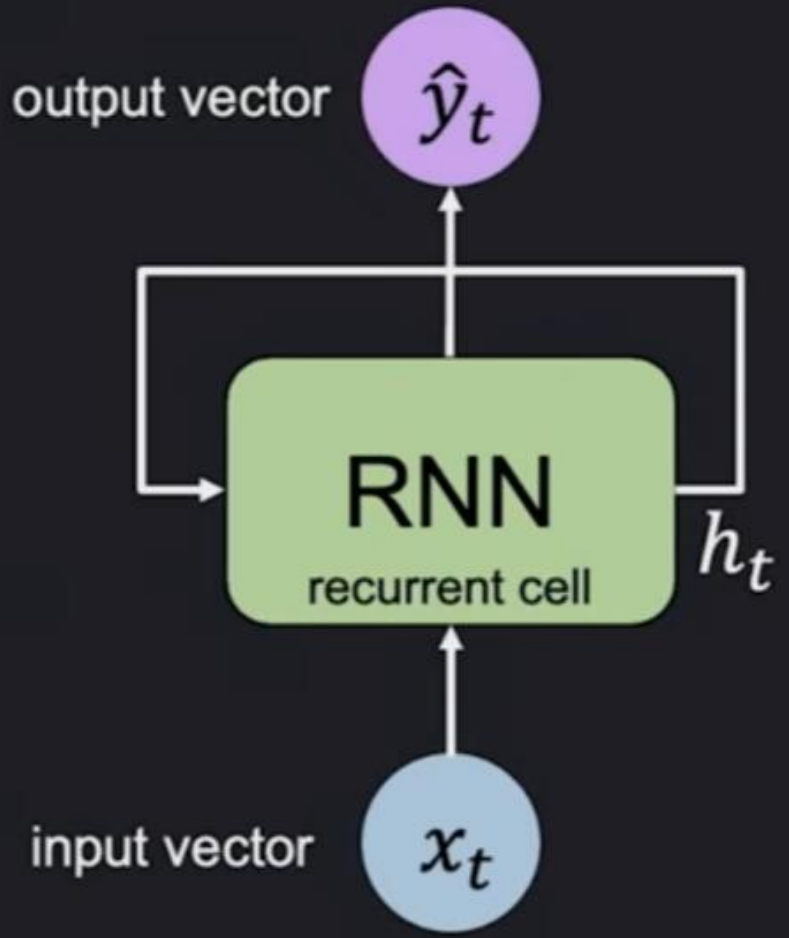
# RNN Intuition

```
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!"
```

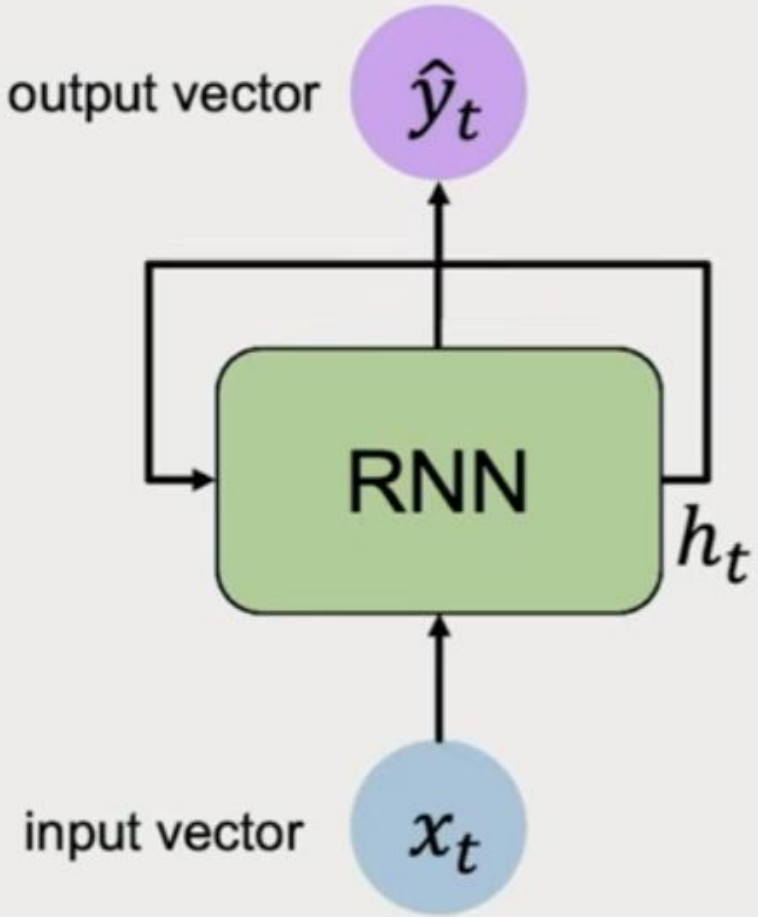


# RNN Intuition

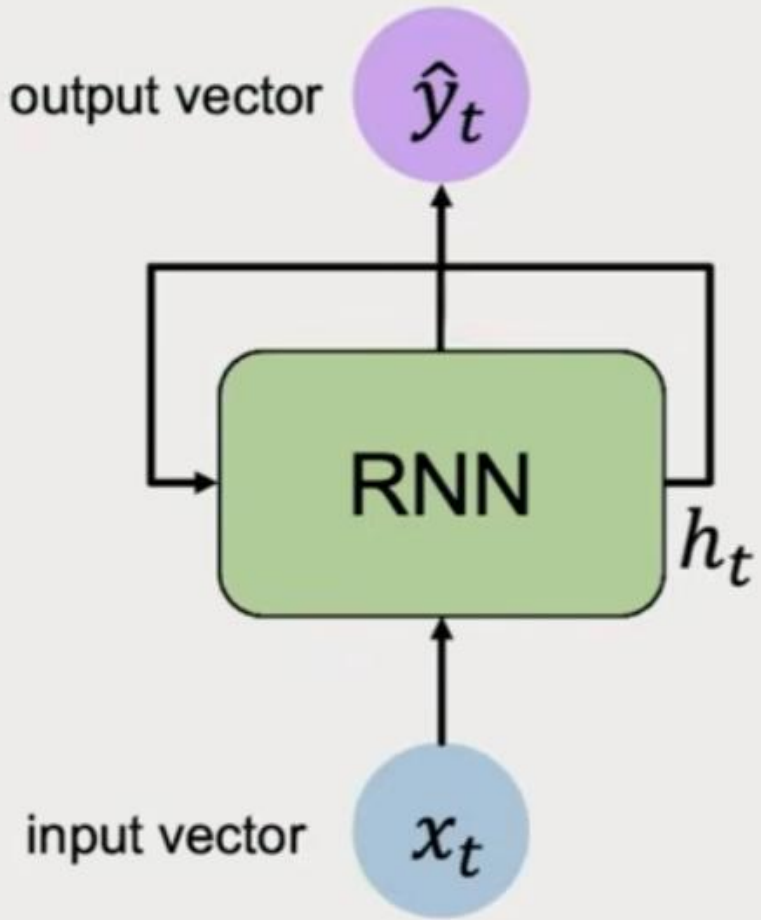
```
my_rnn = RNN()
hidden_state = [0, 0, 0, 0]

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

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

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



# RNN State Update and Output



# RNN State Update and Output



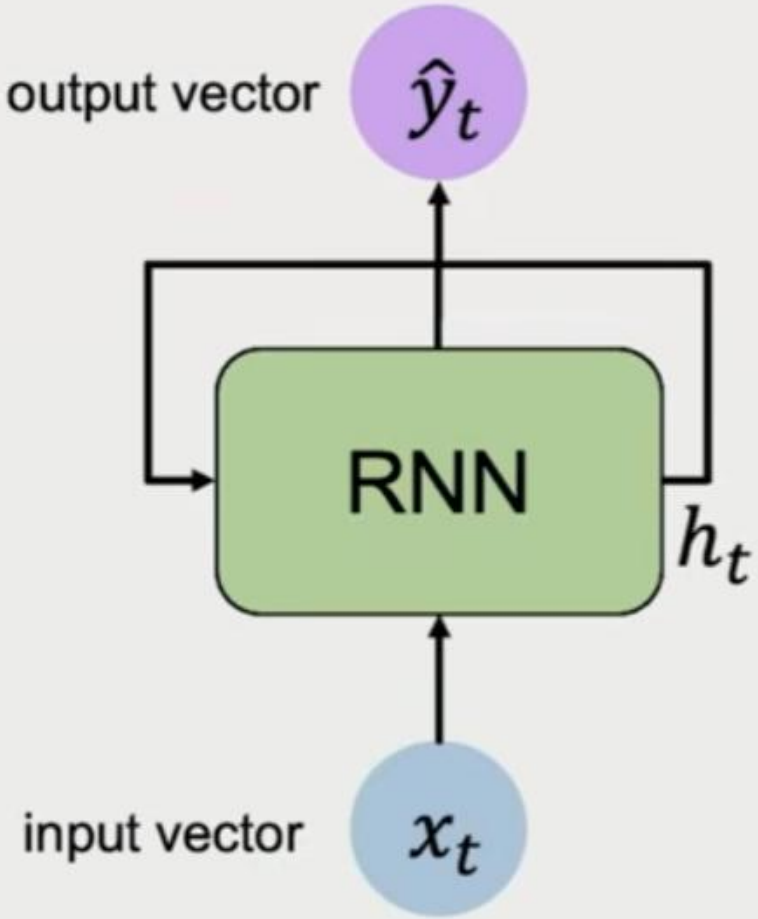

Update Hidden State

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

Input Vector

$x_t$

# RNN State Update and Output




## Output Vector

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

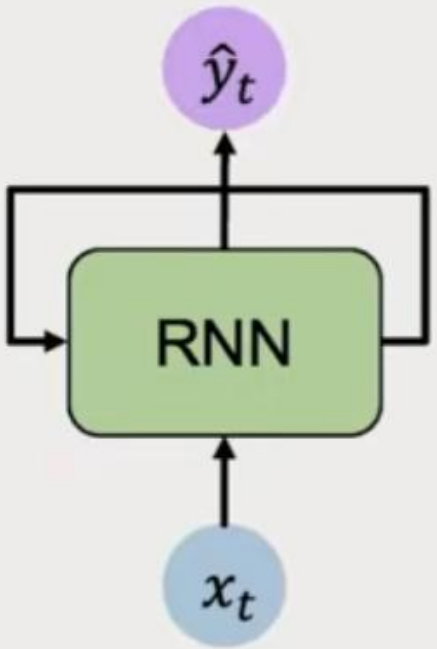
## Update Hidden State

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

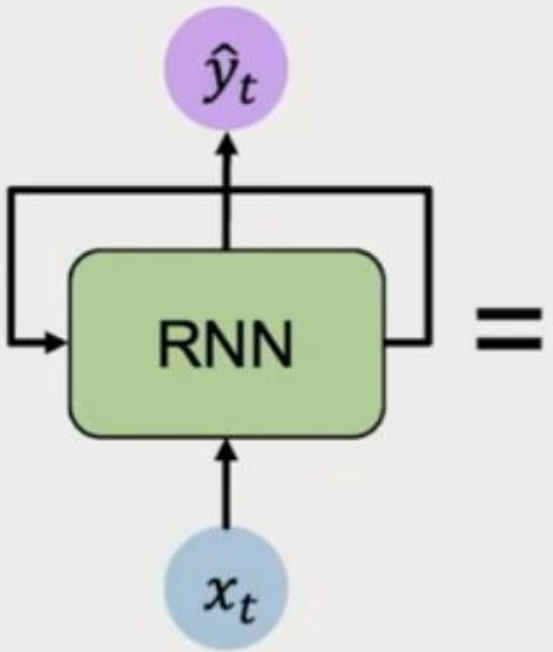
## Input Vector

$$x_t$$

# RNNs: Computational Graph Across Time



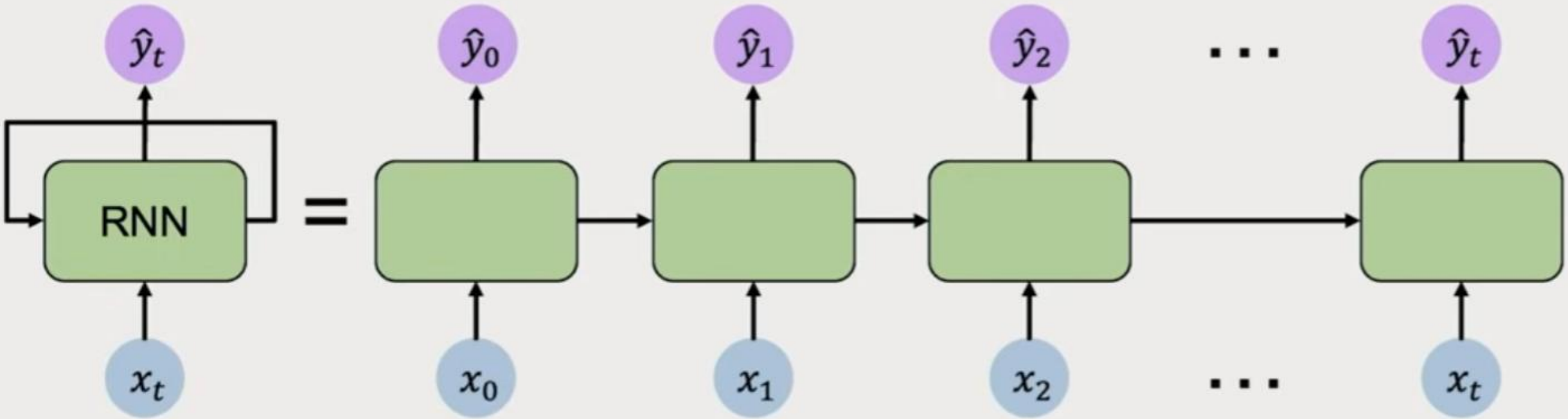
# RNNs: Computational Graph Across Time



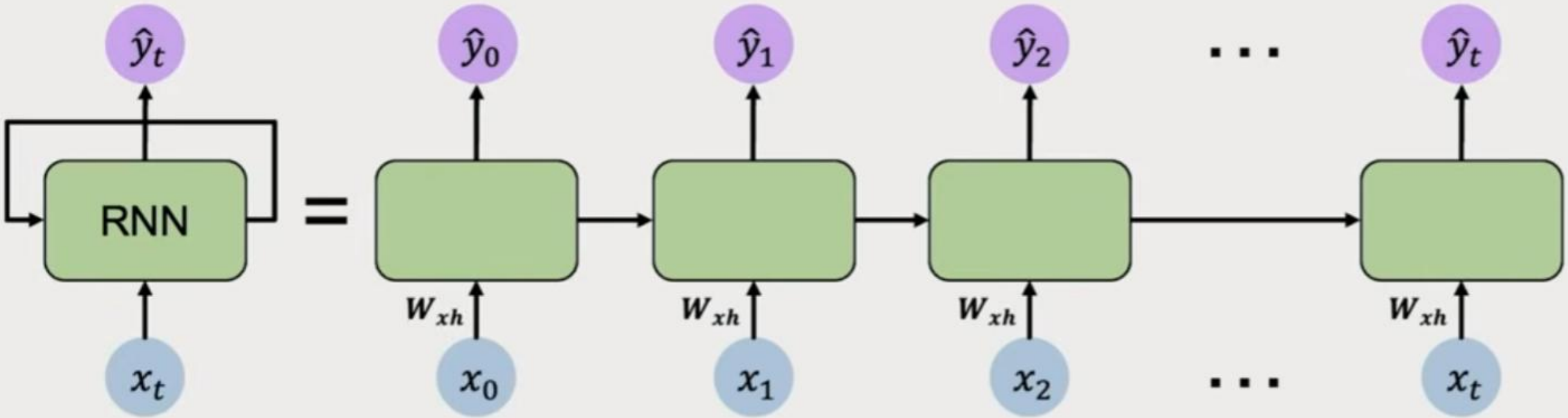

=

Represent as computational graph unrolled across time

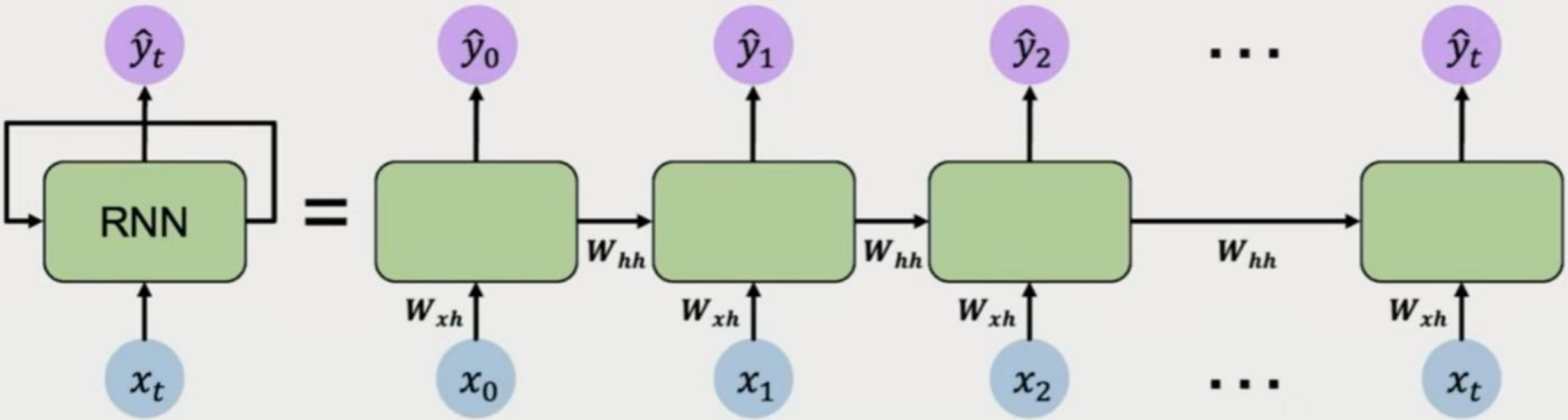
# RNNs: Computational Graph Across Time



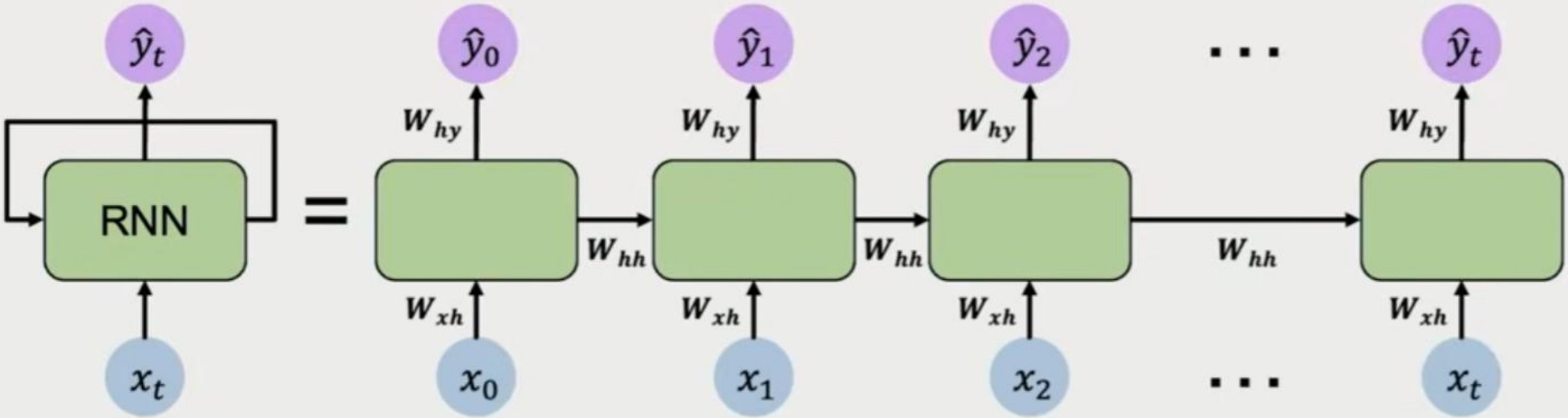
# RNNs: Computational Graph Across Time



# RNNs: Computational Graph Across Time

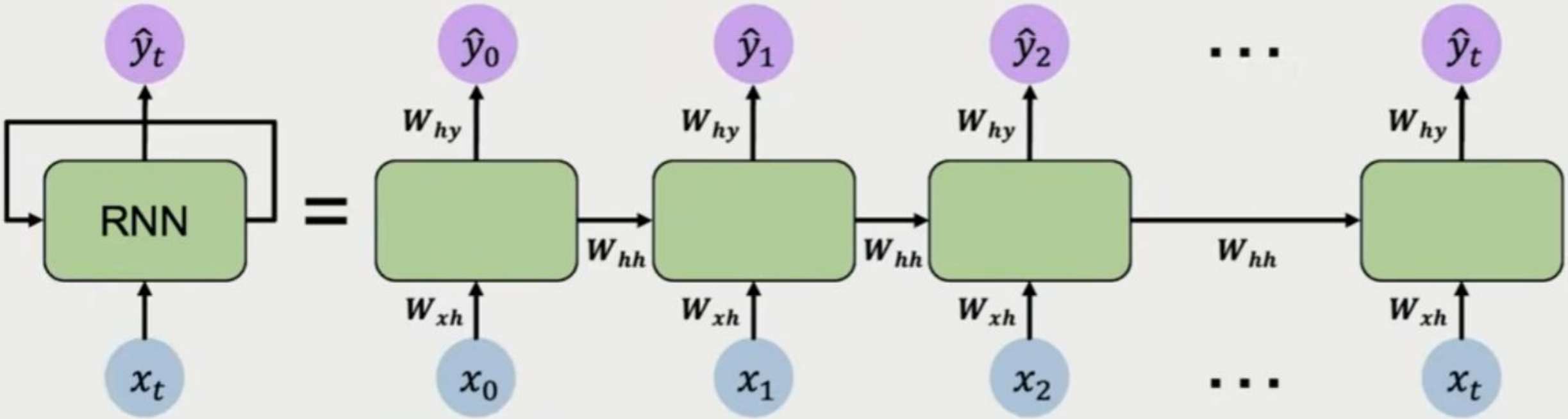


# RNNs: Computational Graph Across Time

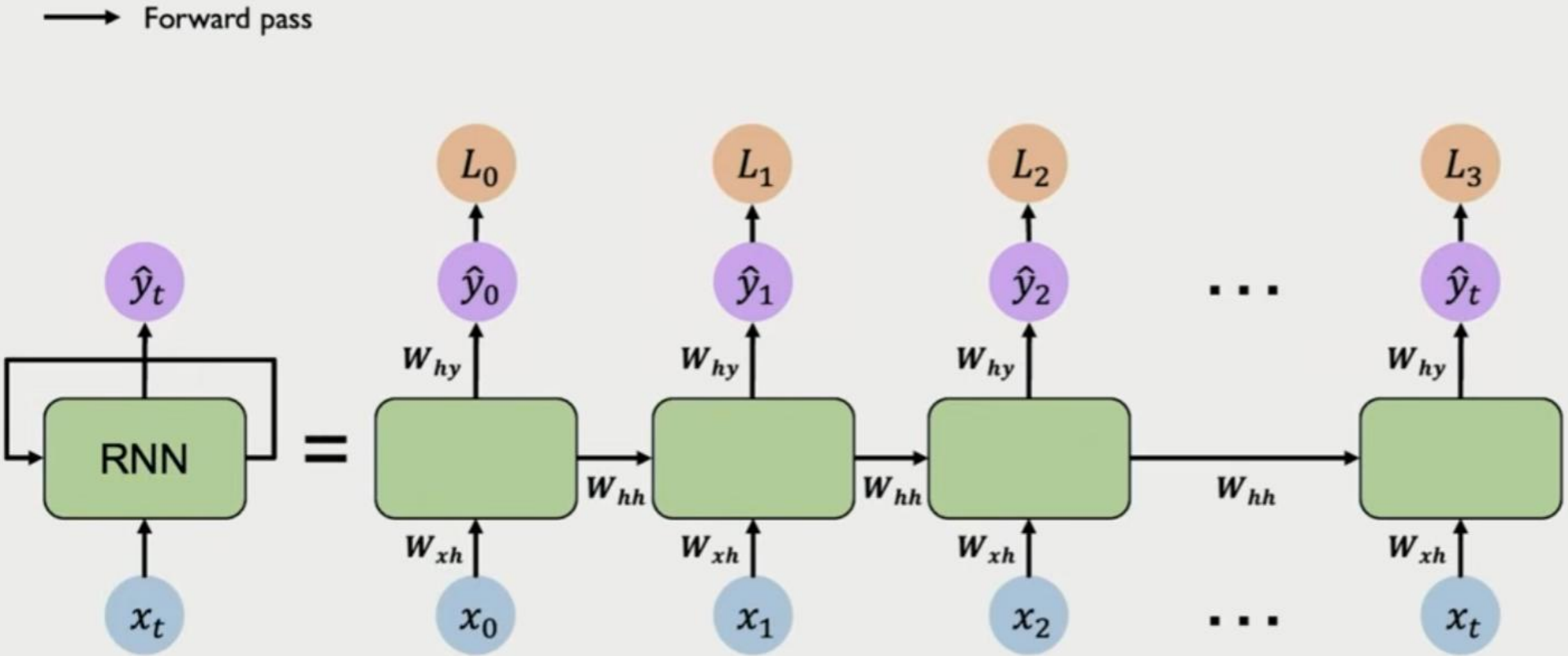


# RNNs: Computational Graph Across Time

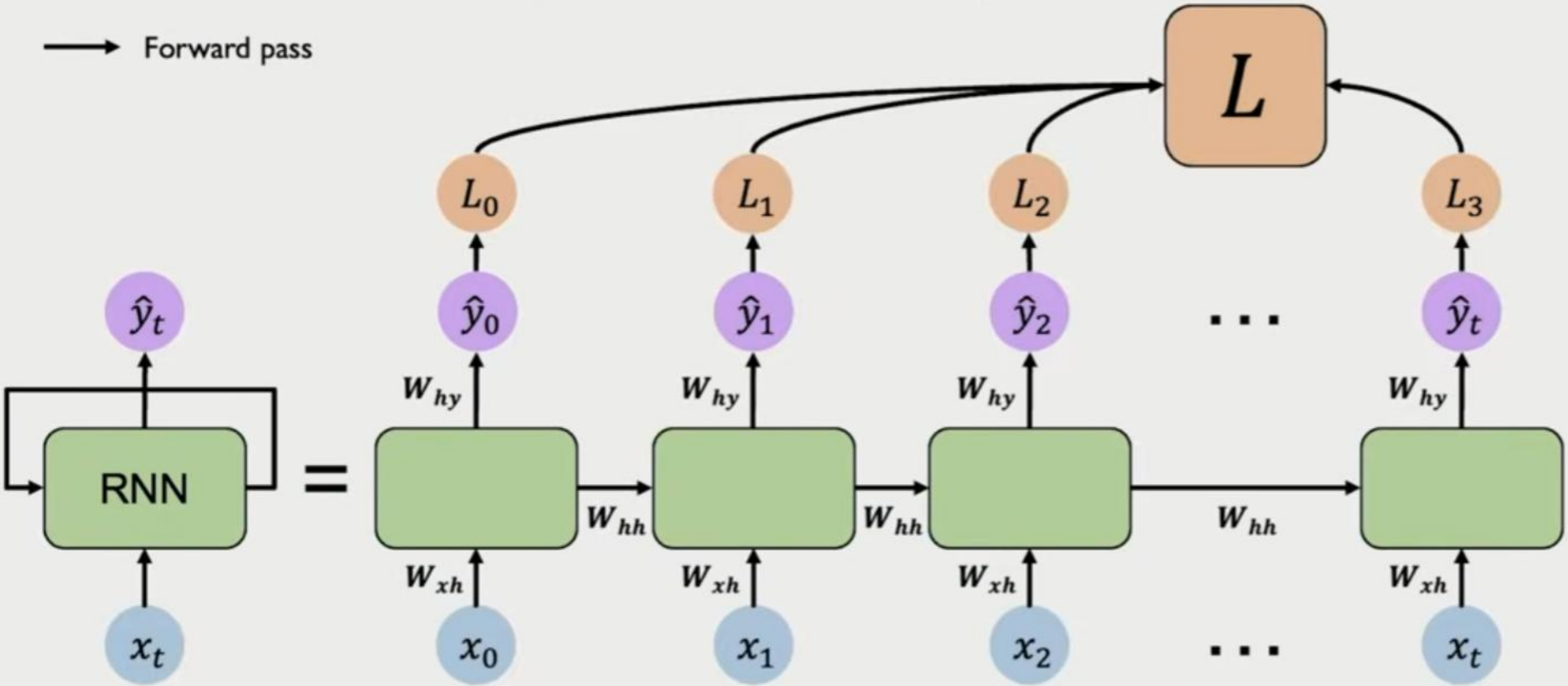
Re-use the **same weight matrices** at every time step



# RNNs: Computational Graph Across Time

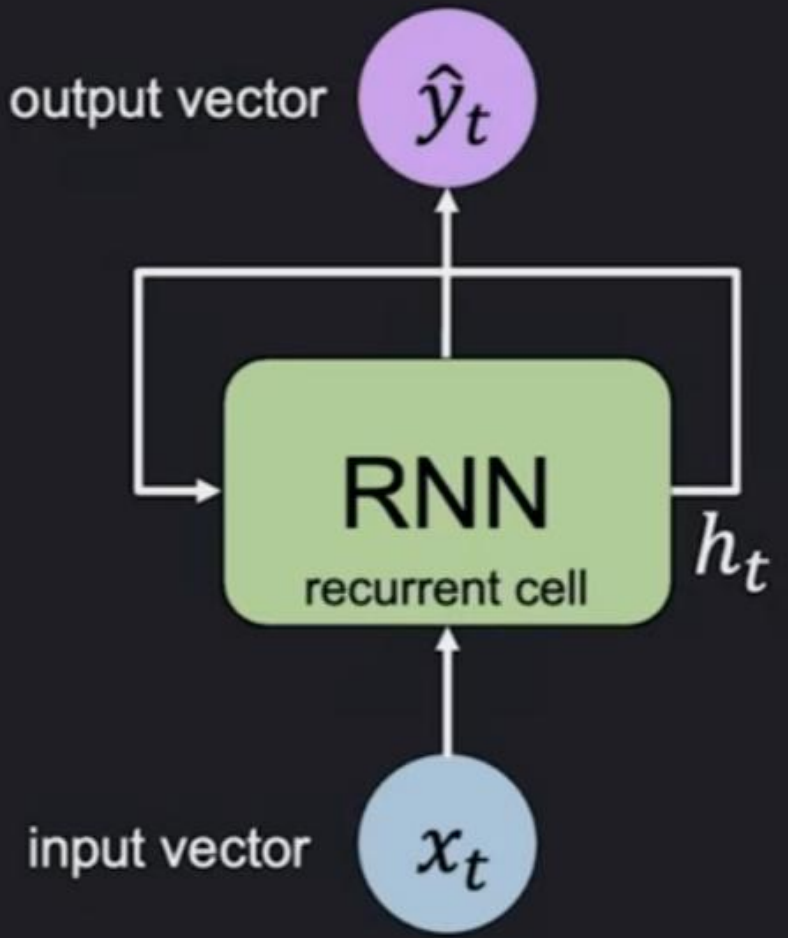


# RNNs: Computational Graph Across Time



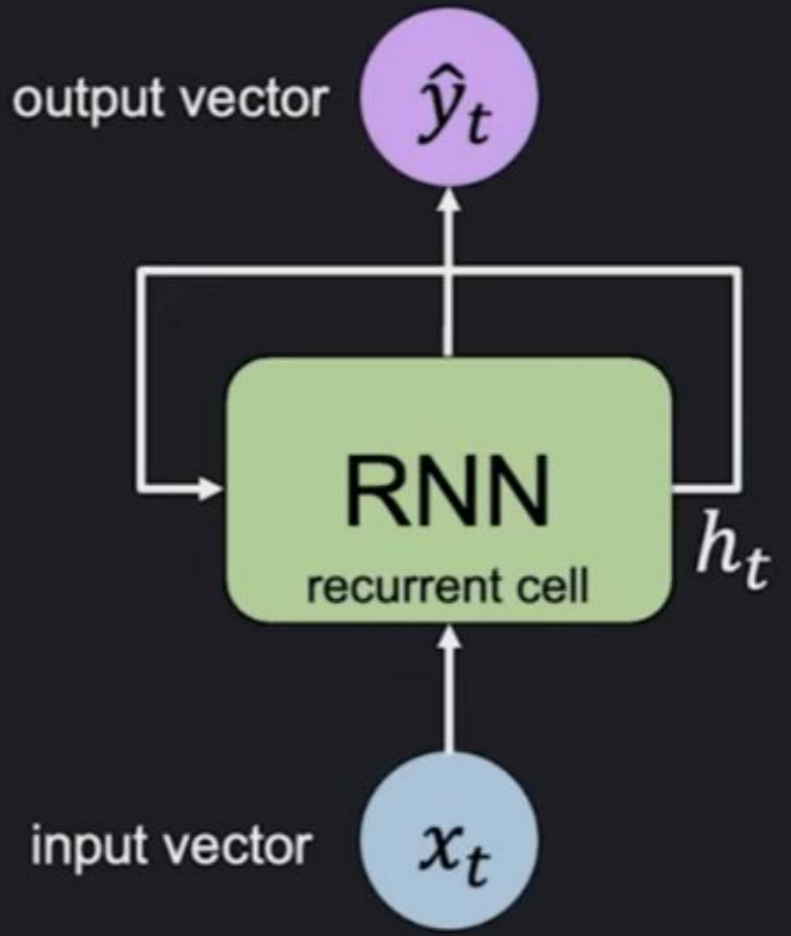
# RNNs from Scratch in TensorFlow

```
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])
```



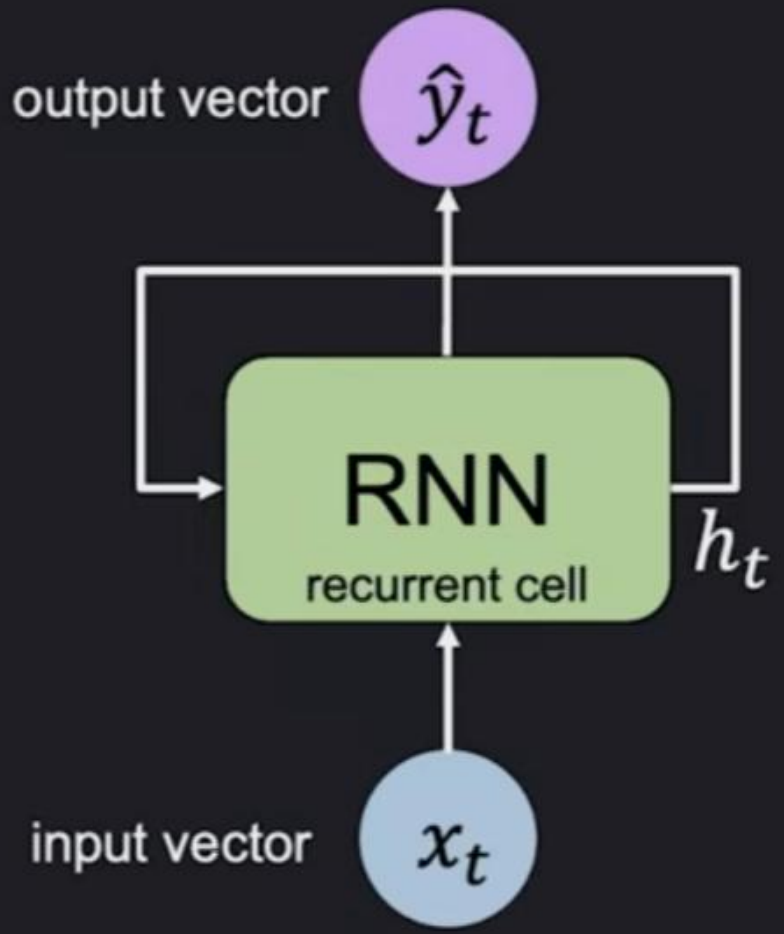
# RNNs from Scratch in TensorFlow

```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
```



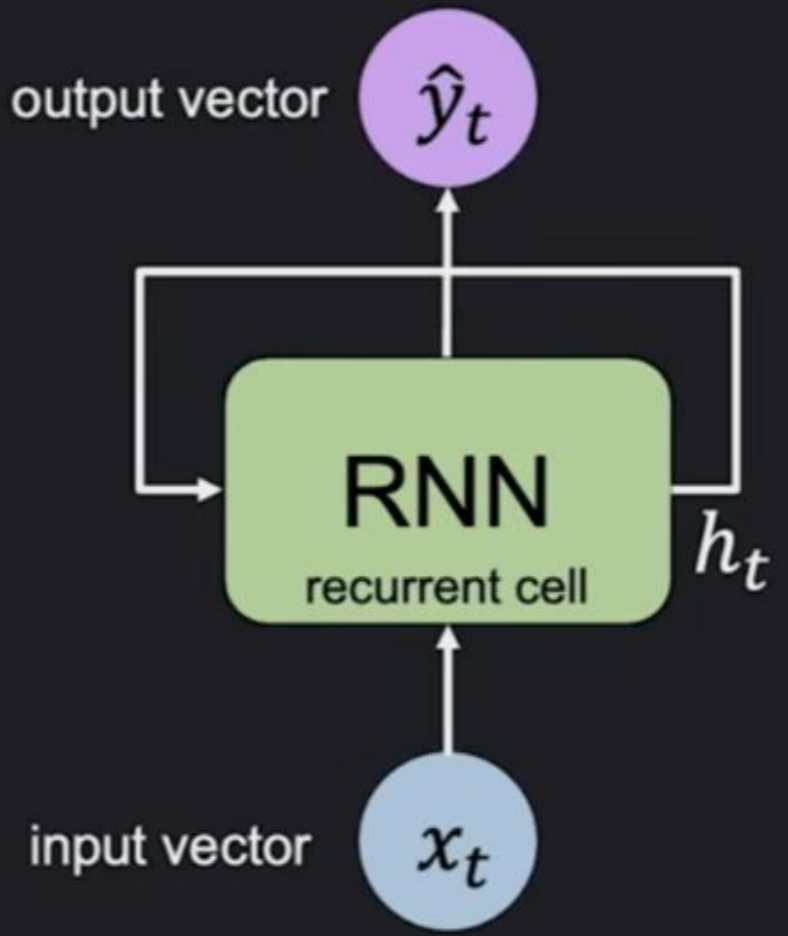
# RNNs from Scratch in TensorFlow

```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
```



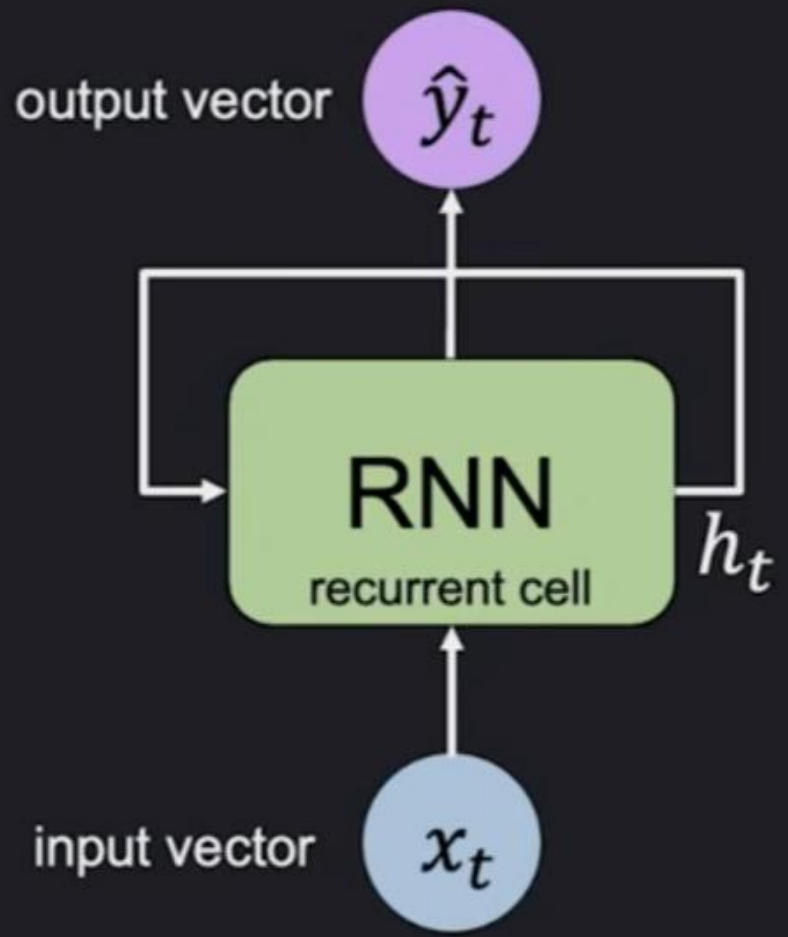
# RNNs from Scratch in TensorFlow

```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
```



# RNNs from Scratch in TensorFlow

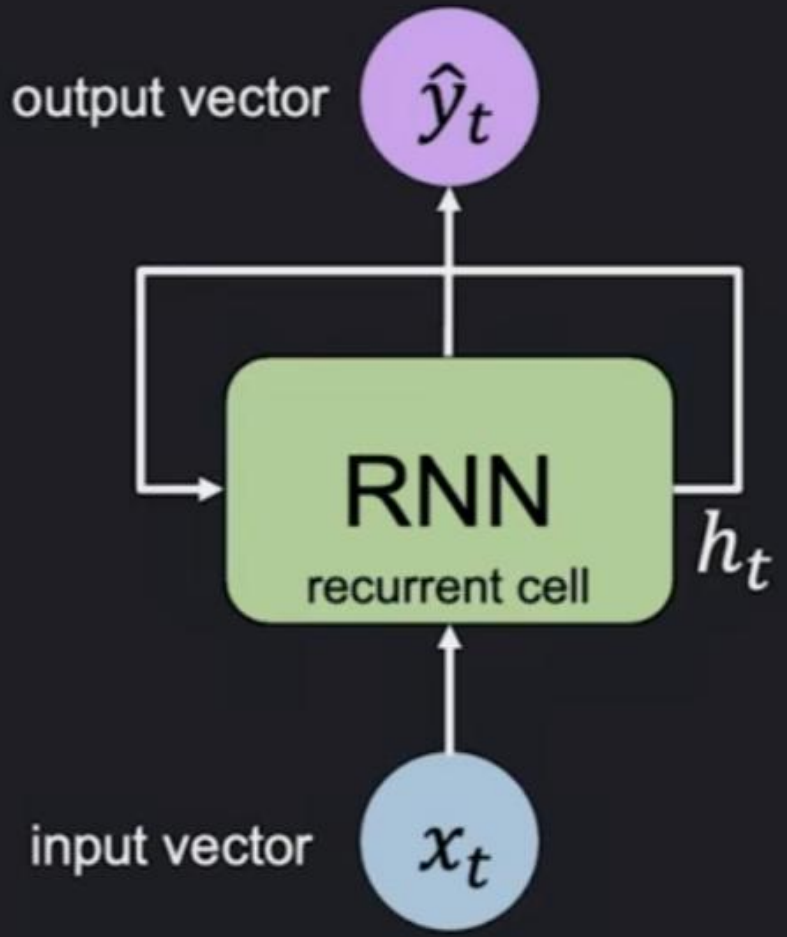
```python
class MyRNNCell(tf.keras.layers.Layer):
  def __init__(self, rnn_units, input_dim, output_dim):
    super(MyRNNCell, self).__init__()

    # 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: TensorFlow & PyTorch

```
from tf.keras.layers import SimpleRNN
    model = SimpleRNN(rnn_units)
```

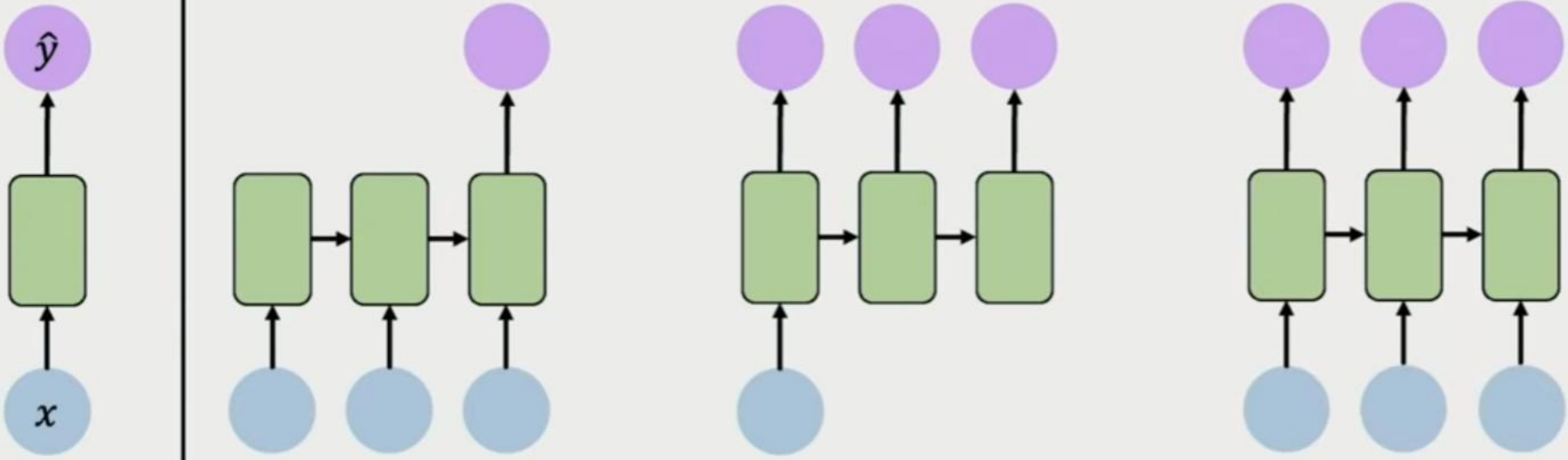
```
    from torch.nn import RNN
model = RNN(input_size, rnn_units)
```



# RNNs for Sequence Modeling


One to One
"Vanilla" NN
*Binary classification*

# 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*

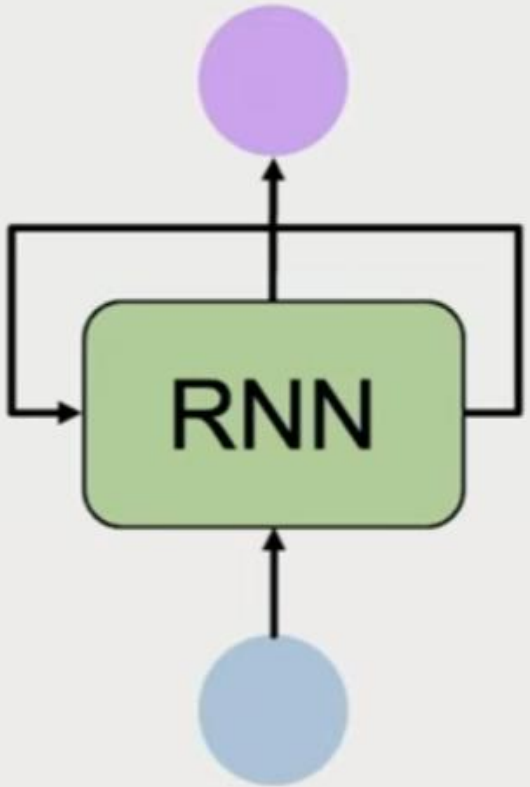
# Sequence Modeling: Design Criteria

To model sequences, we need to:

# 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

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

# 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

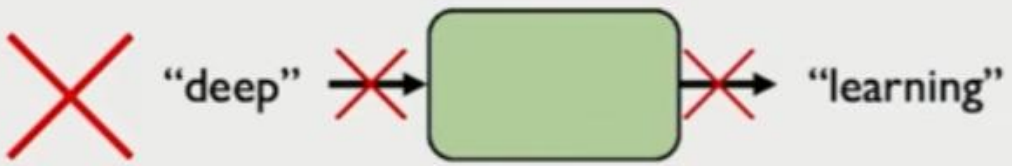
# A Sequence Modeling Problem: Predict the Next Word

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

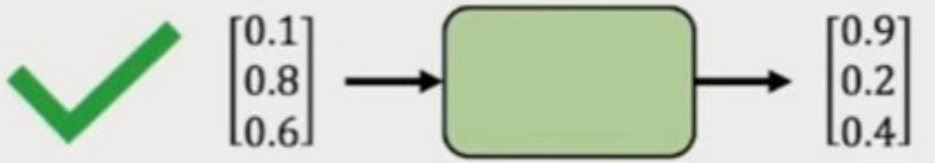
given these words

predict the next word

## Representing Language to a Neural Network



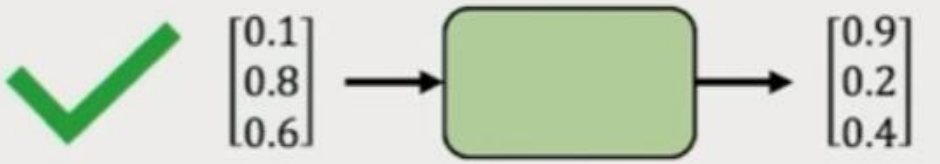

*Neural networks cannot interpret words*




*Neural networks require numerical inputs*

# Encoding Language for a Neural Network

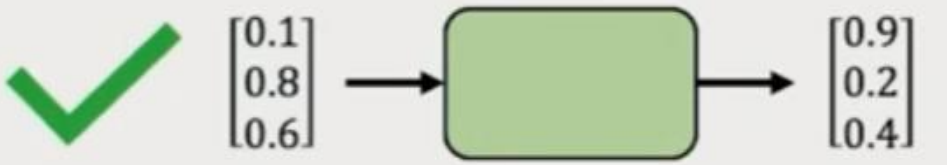

*Neural networks cannot interpret words*




*Neural networks require numerical inputs*

# Encoding Language for a Neural Network

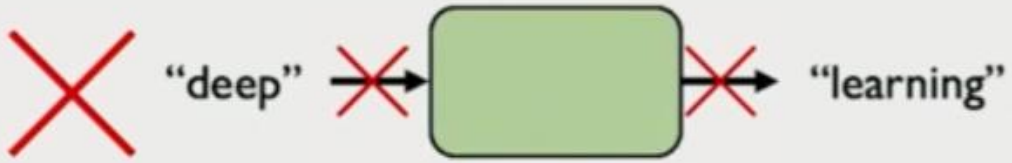

*Neural networks cannot interpret words*



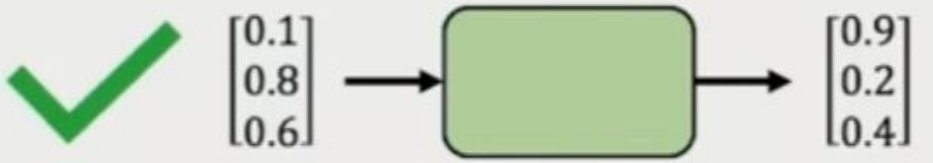

*Neural networks require numerical inputs*

Embedding: transform indexes into a vector of fixed size.

# Encoding Language for a Neural Network

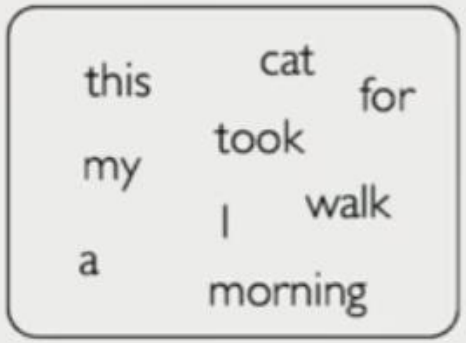



*Neural networks cannot interpret words*



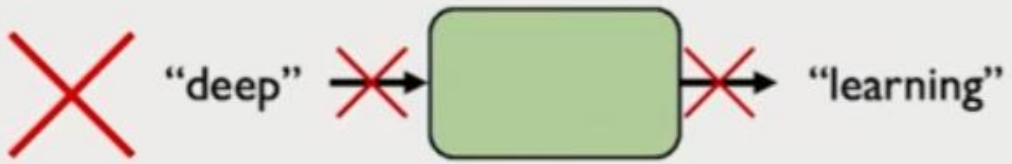

*Neural networks require numerical inputs*

Embedding: transform indexes into a vector of fixed size.

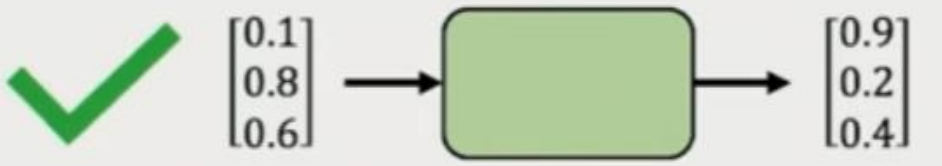



**1. Vocabulary:**
Corpus of words

# Encoding Language for a Neural Network

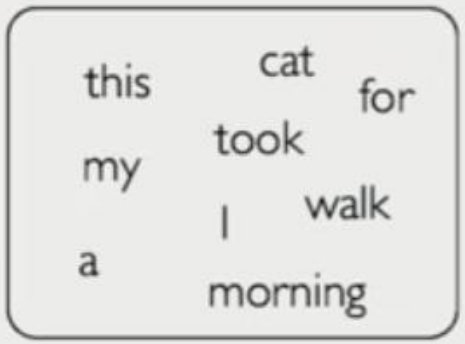



*Neural networks cannot interpret words*

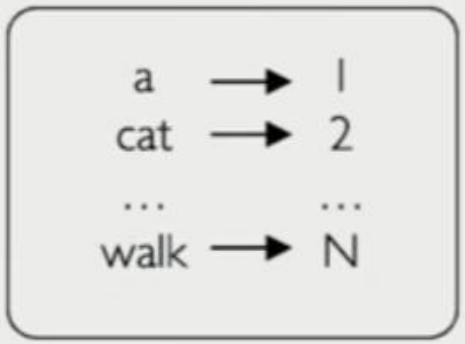



*Neural networks require numerical inputs*

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

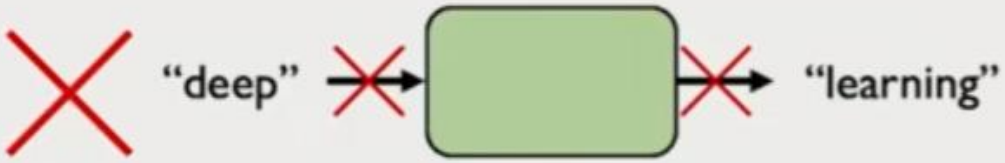



**1. Vocabulary:**
Corpus of words

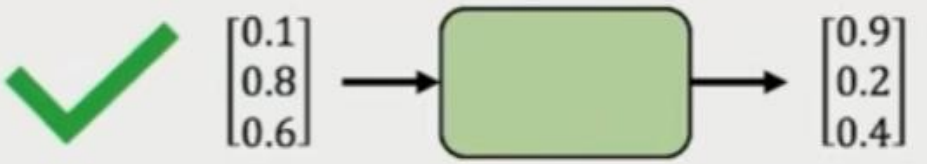



**2. Indexing:**
Word to index

# Encoding Language for a Neural Network

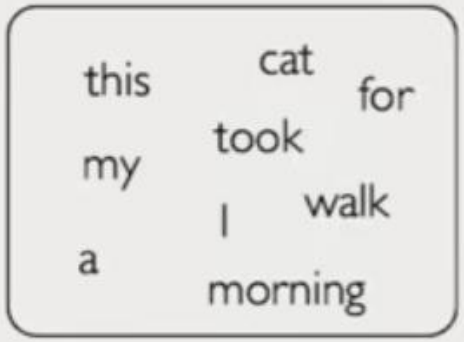



*Neural networks cannot interpret words*

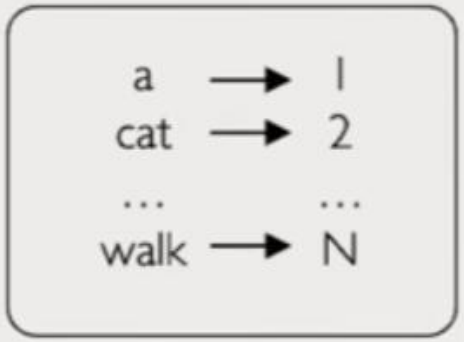



*Neural networks require numerical inputs*

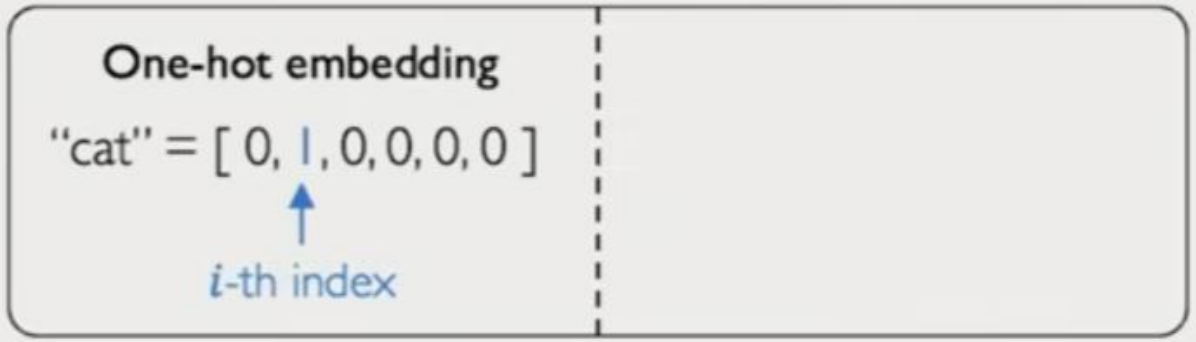
## Embedding: transform indexes into a vector of fixed size.




**1. Vocabulary:**
Corpus of words



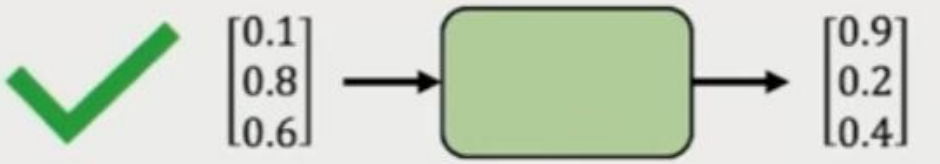

**2. Indexing:**
Word to index




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

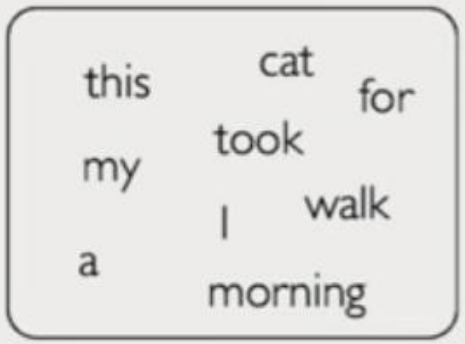
# Encoding Language for a Neural Network


*Neural networks cannot interpret words*

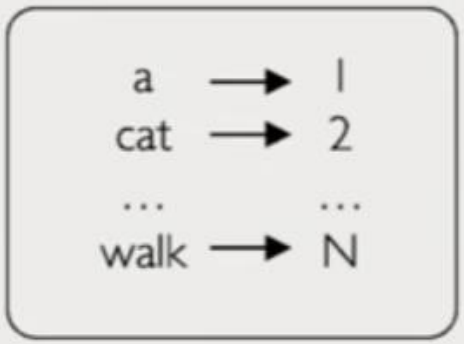



*Neural networks require numerical inputs*

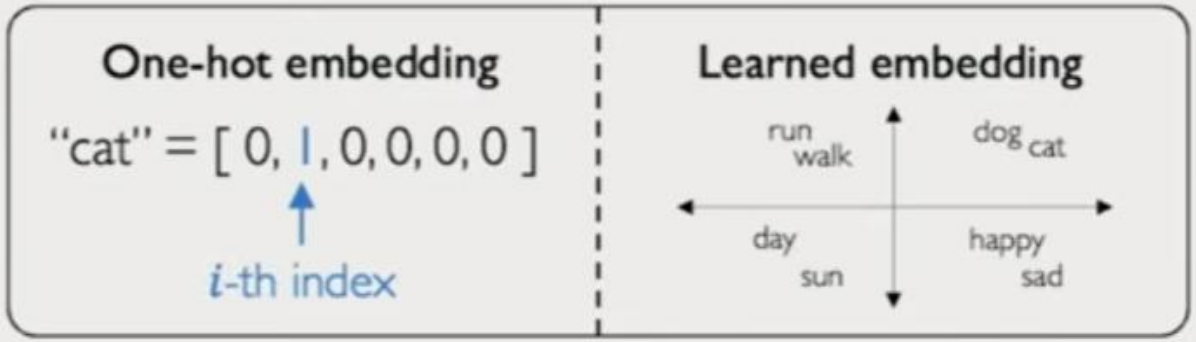
## Embedding: transform indexes into a vector of fixed size.




**1. Vocabulary:**
Corpus of words




**2. Indexing:**
Word to index




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

# Handle Variable Sequence Lengths

The food was great

# Handle Variable Sequence Lengths

The food was great

vs.

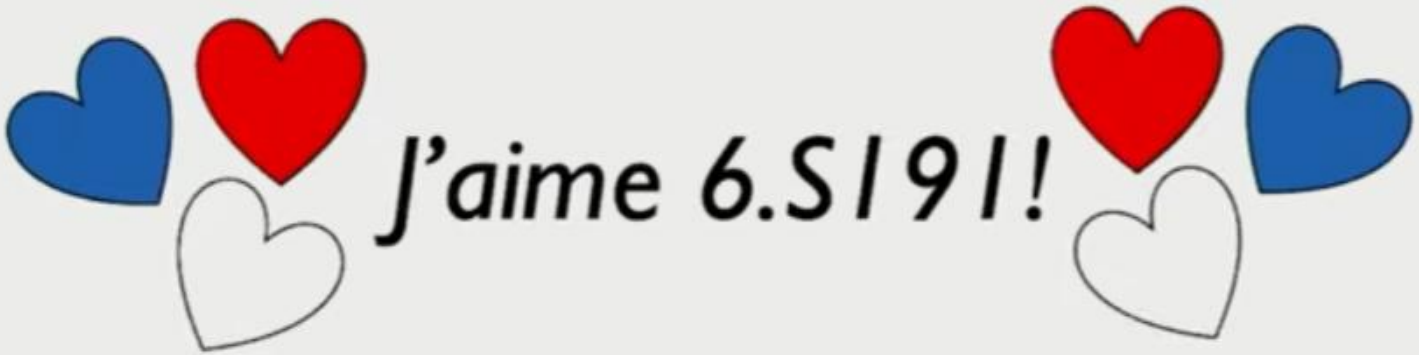
We visited a restaurant for lunch

vs.

We were hungry but cleaned the house before eating

# Model Long-Term Dependencies

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




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.

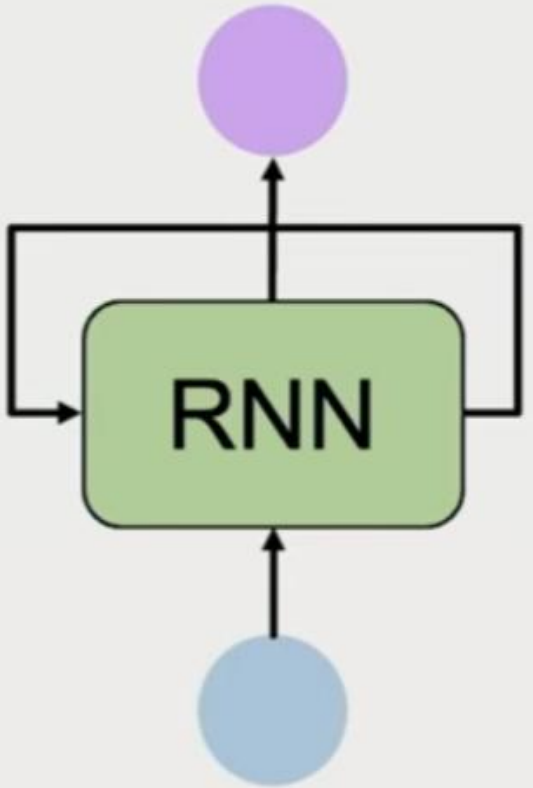
vs.

The food was bad, not good at all.

# Sequence Modeling: Design Criteria

To model sequences, we need to:

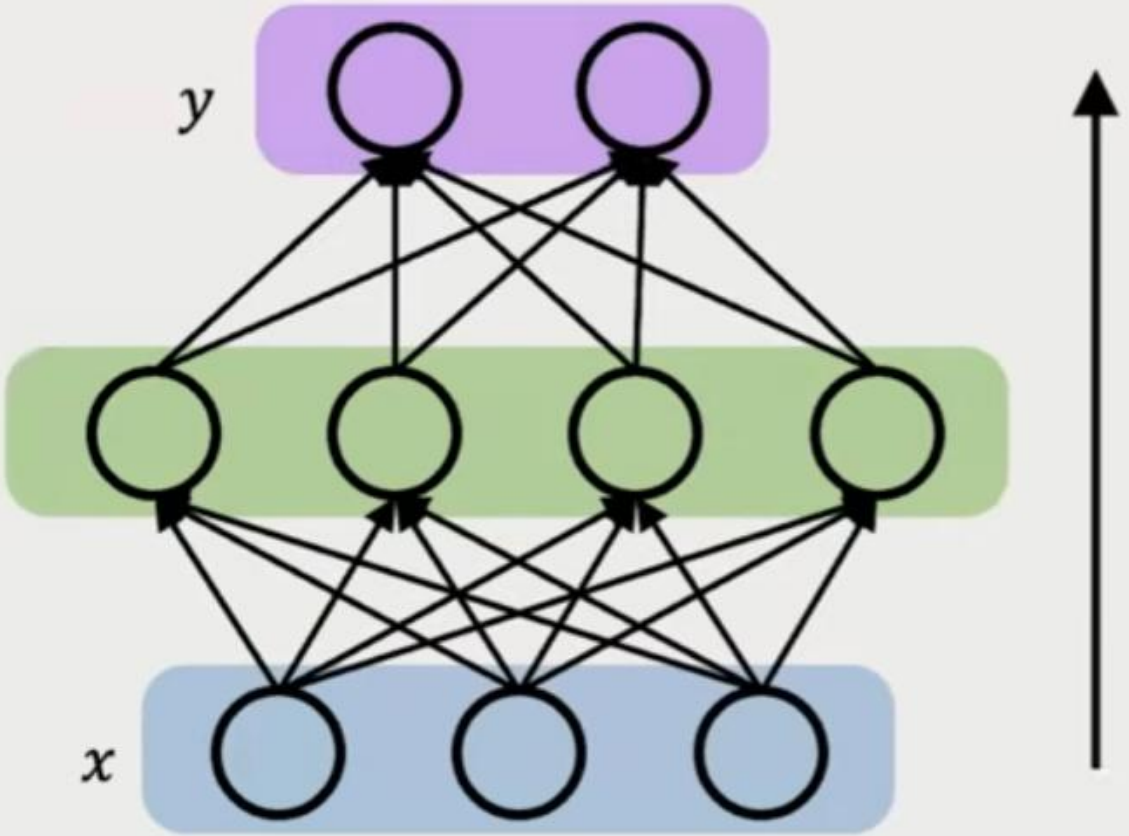
1. Handle **variable-length** sequences
2. Track **long-term** dependencies
3. Maintain information about **order**
4. **Share parameters** across the sequence



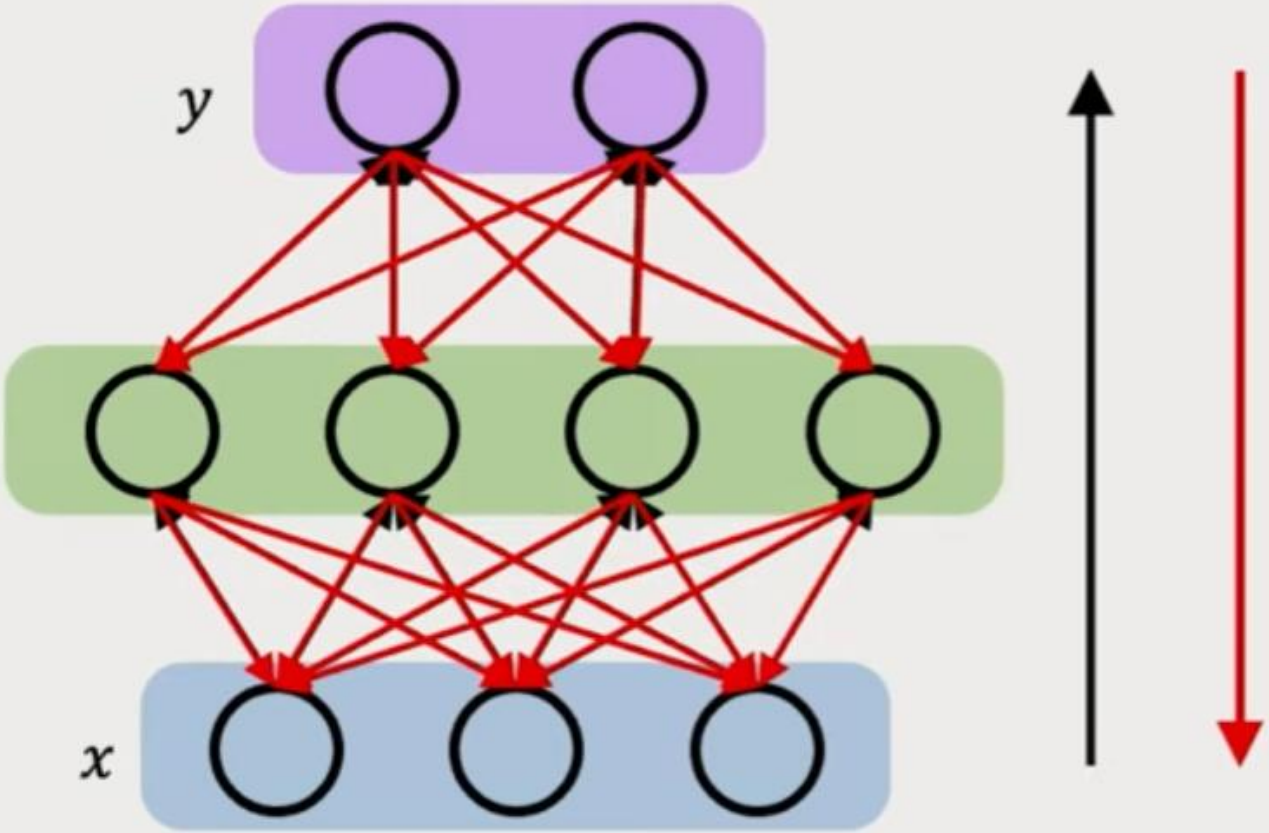

**Recurrent Neural Networks (RNNs)** meet these sequence modeling design criteria

# Backpropagation Through Time (BPTT)

# Recall: Backpropagation in Feed Forward Models



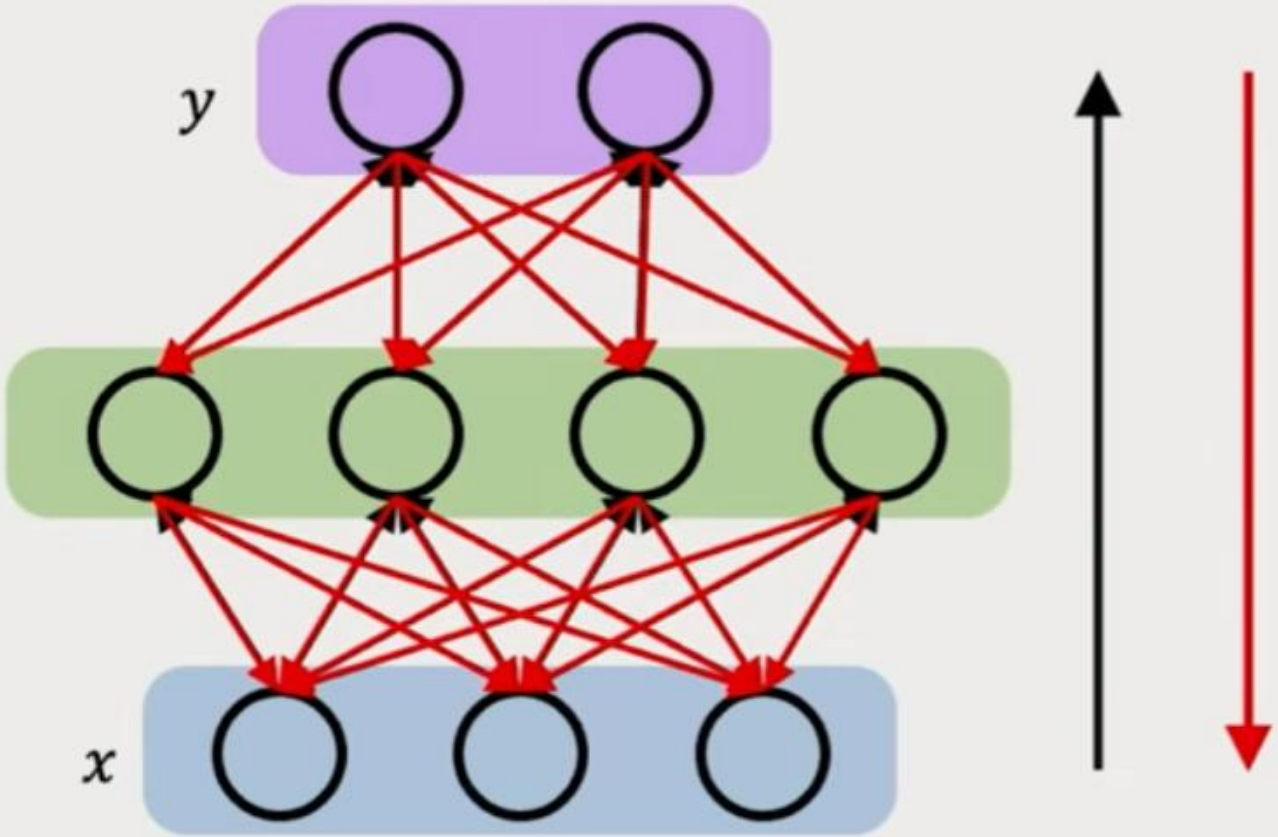
# Recall: Backpropagation in Feed Forward Models




**Backpropagation algorithm:**

1. Take the derivative (gradient) of the loss with respect to each parameter

# Recall: Backpropagation in Feed Forward Models

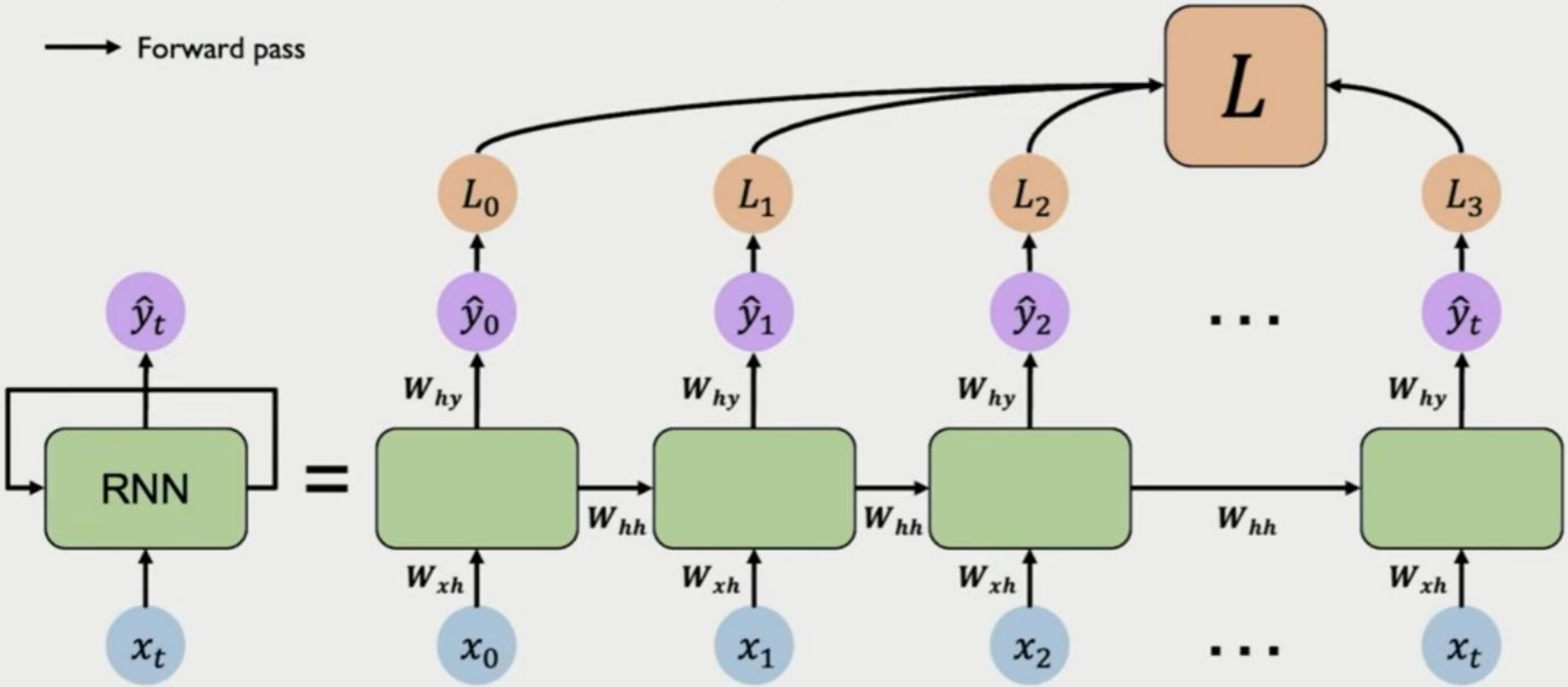




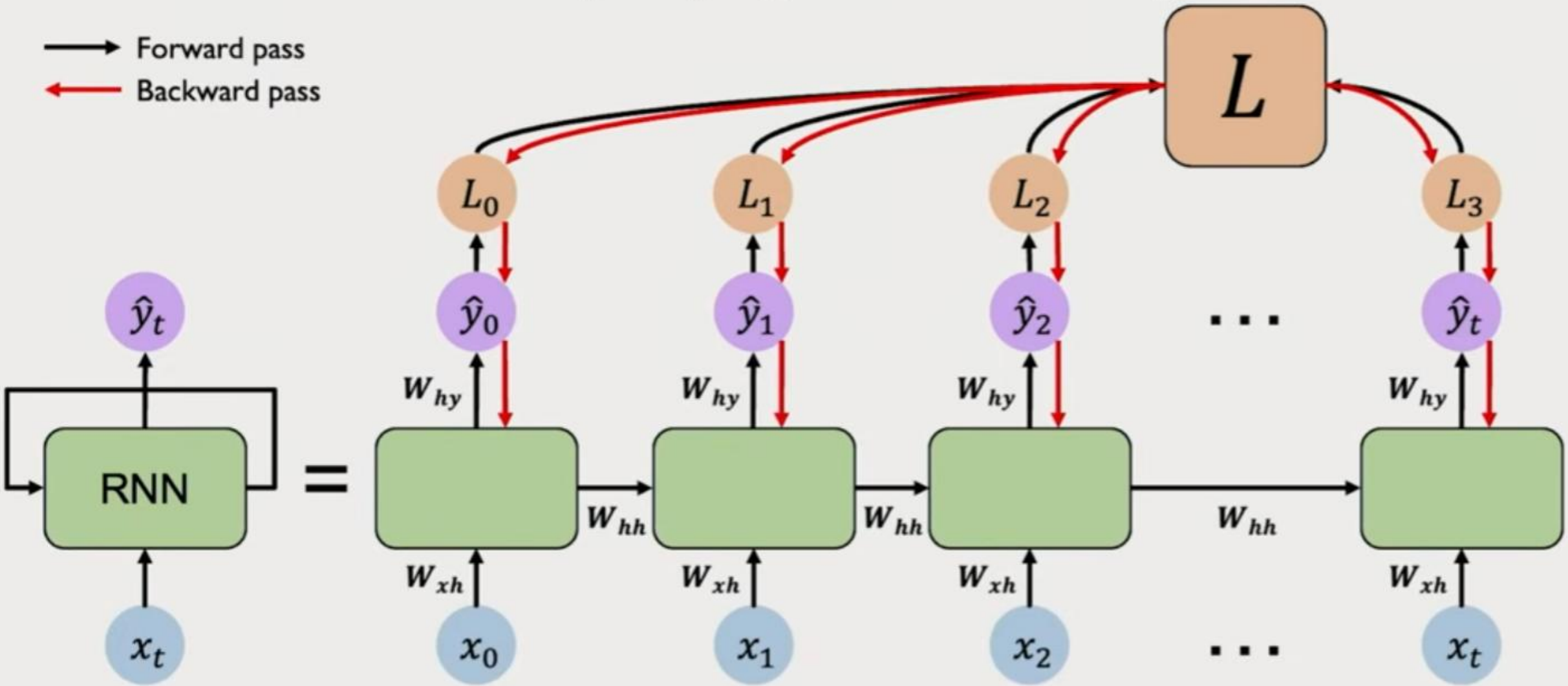

**Backpropagation algorithm:**

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

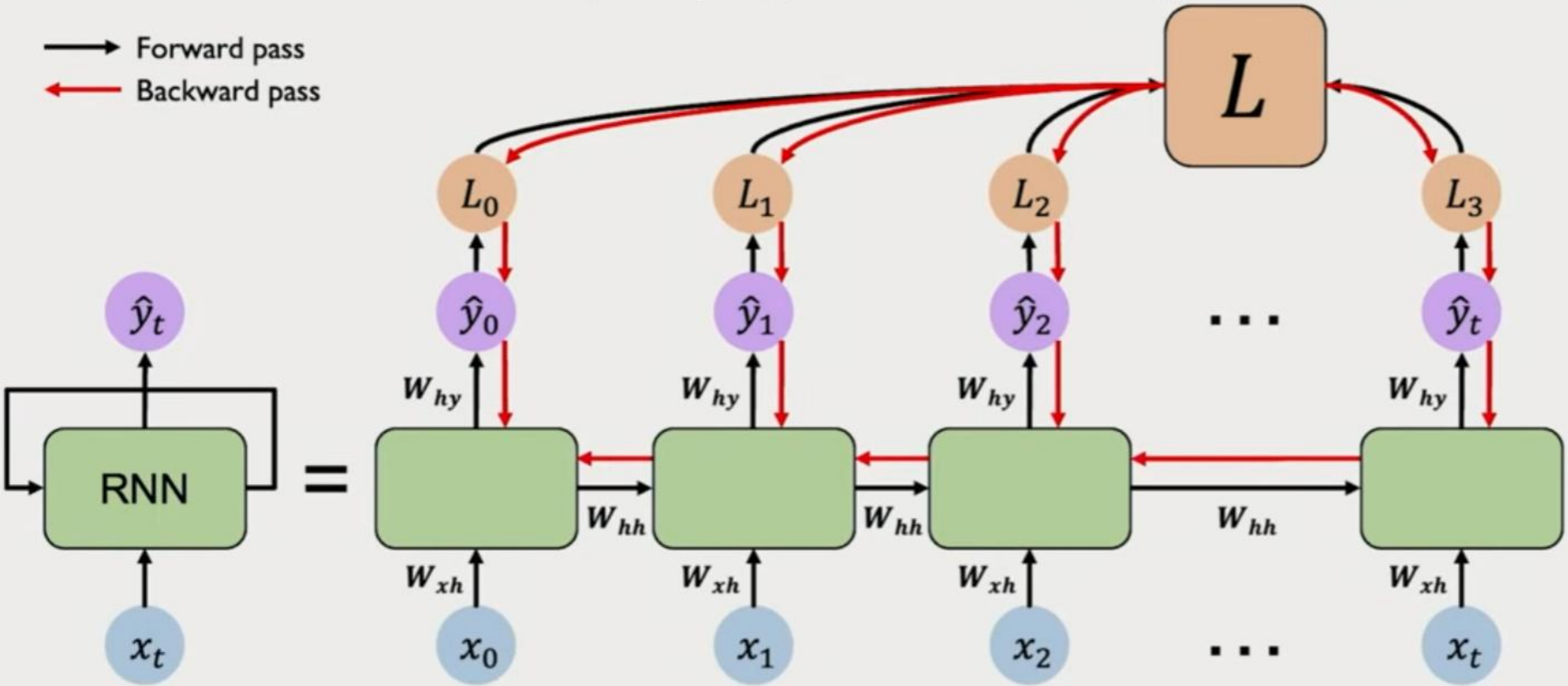
# RNNs: Backpropagation Through Time



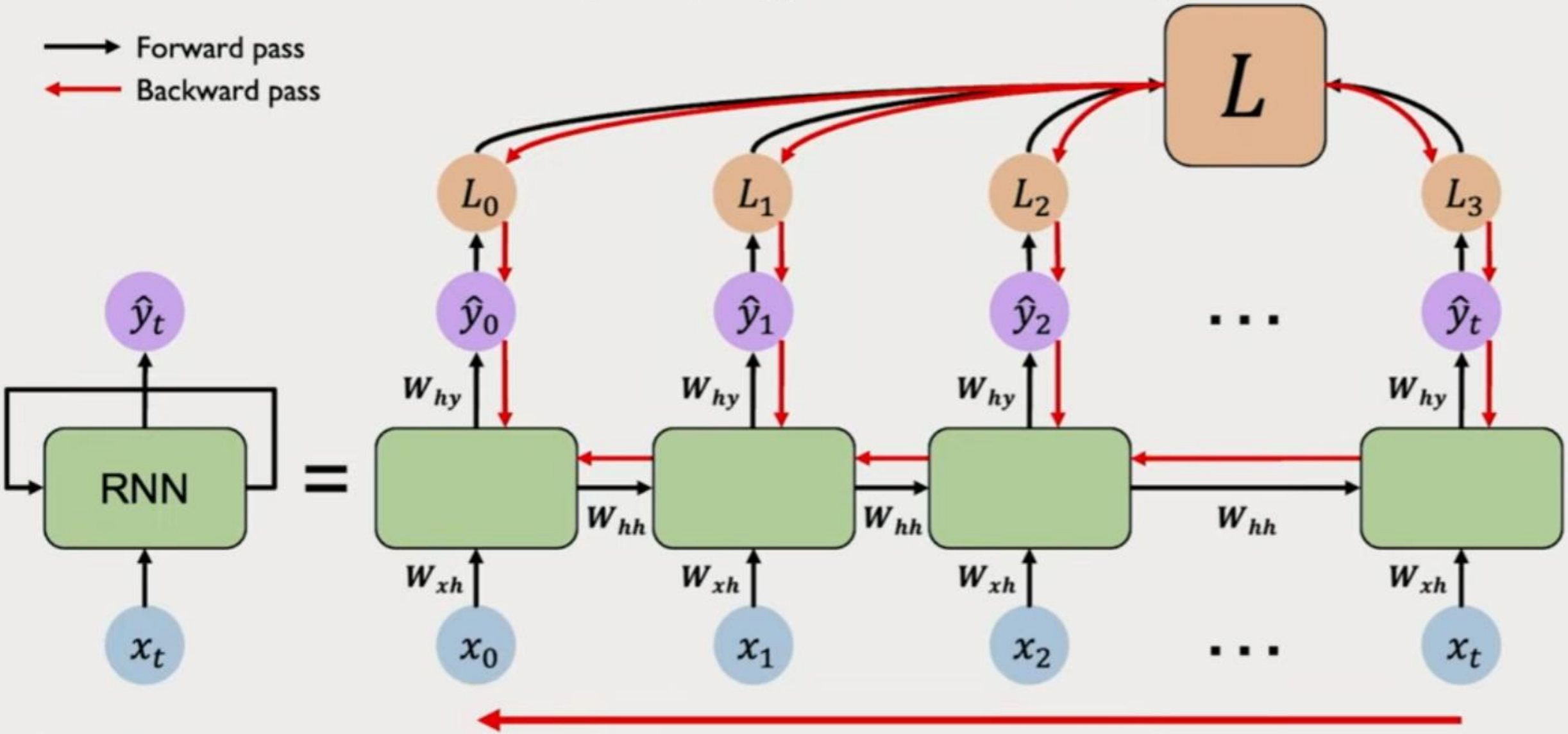
# RNNs: Backpropagation Through Time



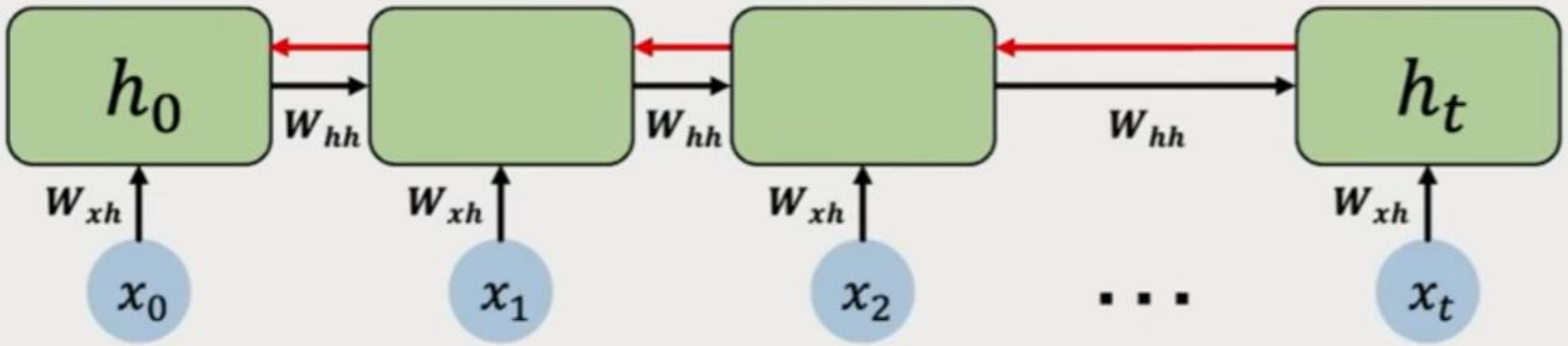
# RNNs: Backpropagation Through Time



# RNNs: Backpropagation Through Time

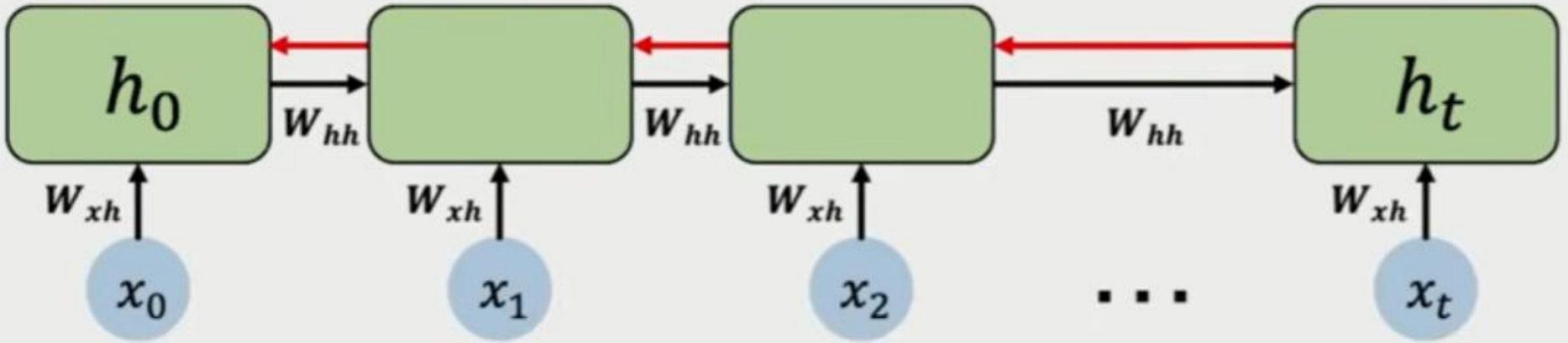


# Standard RNN Gradient Flow




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

# Standard RNN Gradient Flow: Exploding Gradients

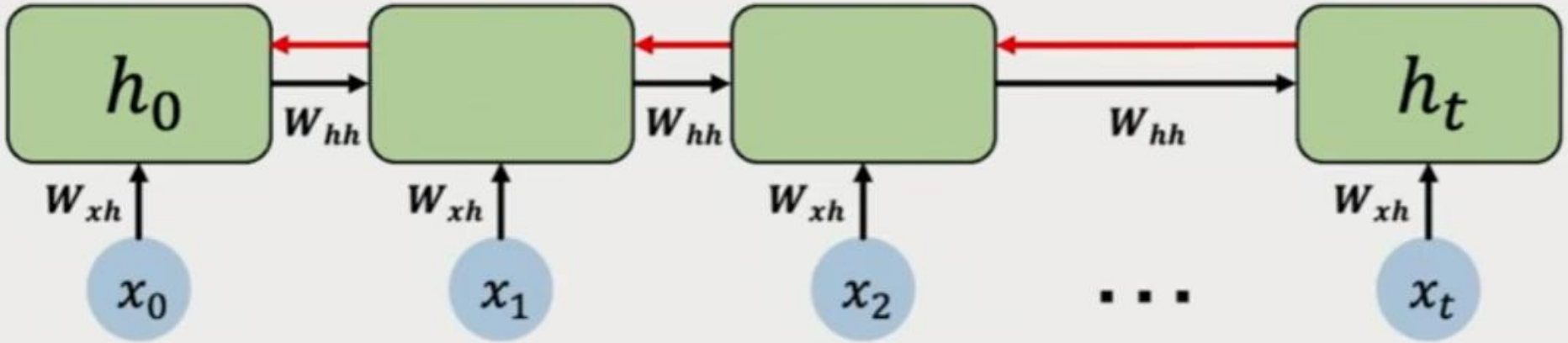



Computing the gradient wrt $h_0$ involves **many factors of $W_{hh}$ + repeated gradient computation!**

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

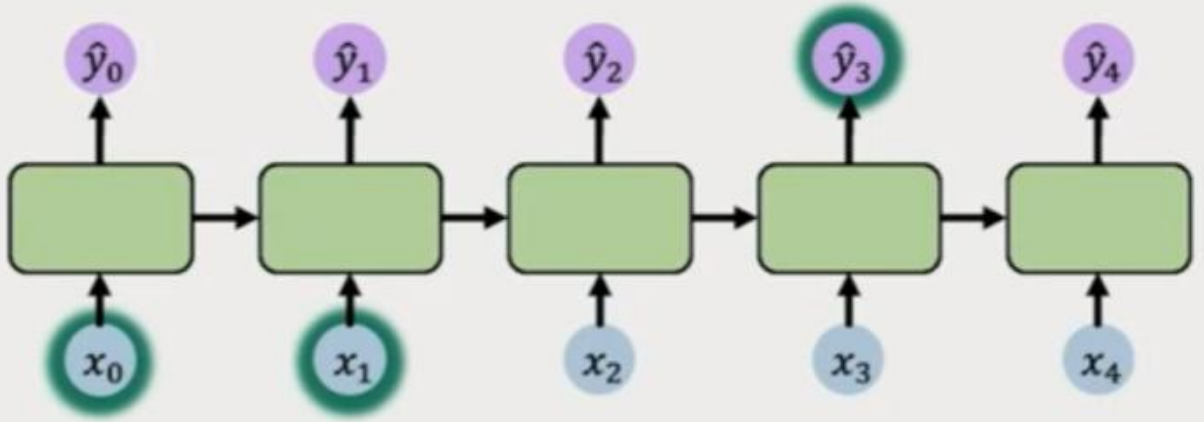
↓

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

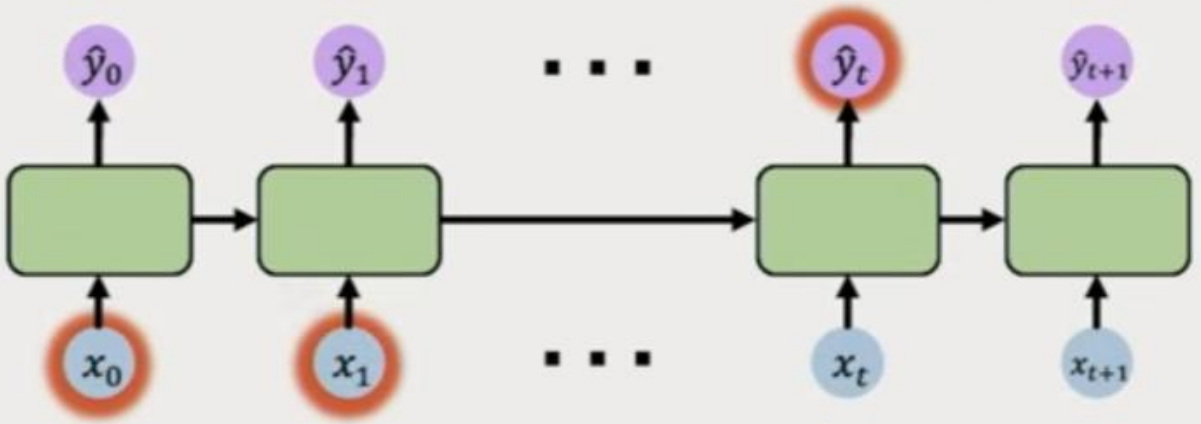
↓

Bias parameters to capture short-term dependencies

"The clouds are in the ___"




"I grew up in France, … and I speak fluent___ "

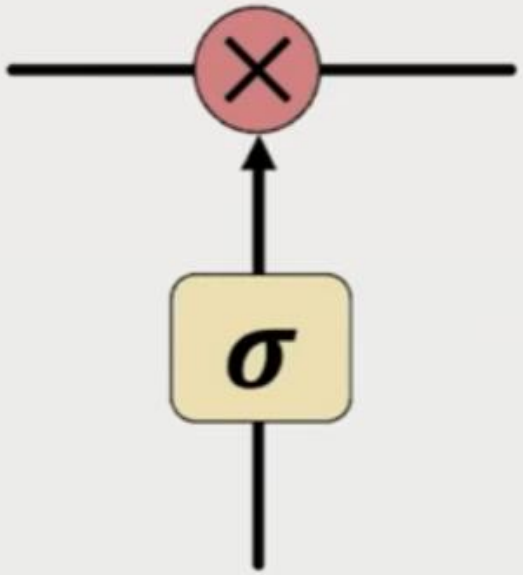


# Gating Mechanisms in Neurons

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

Pointwise multiplication

Sigmoid neural net layer



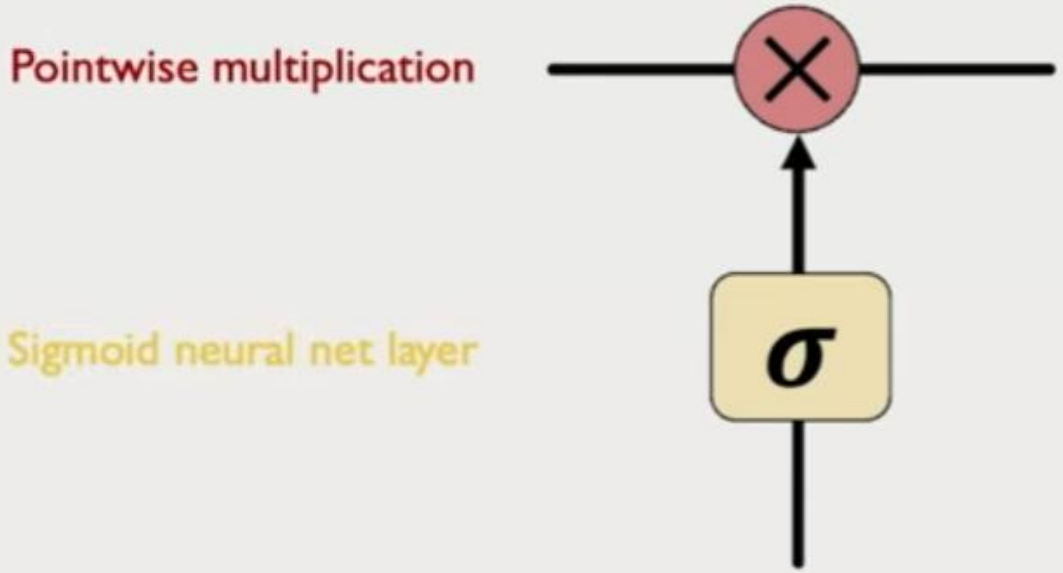

gated cell

LSTM, GRU, etc.

Gates optionally let information through the cell

# Gating Mechanisms in Neurons

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




gated cell

LSTM, GRU, etc.

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.