THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

Applied Machine Intelligence and Reinforcement Learning

Professor Hamza F. Alsarhan

SEAS 8505

Lecture 7

July 27, 2024

Welcome to SEAS Online at George Washington University

Class will begin shortly

Audio: To eliminate background noise, please be sure your audio is muted. To speak, please click the hand icon at the bottom of your screen (Raise Hand). When instructor calls on you, click microphone icon to unmute. When you've finished speaking, *be sure to mute yourself again*.

Chat: Please type your questions in Chat.

Recordings: As part of the educational support for students, we provide downloadable recordings of each class session to be used exclusively by registered students in that particular class for their own private use. Releasing these recordings is strictly prohibited.

Agenda

- Processing Sequences Using RNNs & LSTM
- Recurrent Neural Networks (RNNs) - Principles of Operation
- A Sequence Modeling Problem – Predict the Next Word
- Backpropagation Through Time (BPTT)
- Long Short-Term Memory (LSTM) Networks
- RNN Applications

THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

Processing Sequences Using RNNs & LSTM

Professor Hamza F. Alsarhan
SEAS 8505

Introduction to Recurrent Neural Networks

- New Paradigm - Temporal relationships
- Sequences of Instances

Example Domains Include:

- Stock Market prediction
- Music Generation
- Video Analysis
- etc.

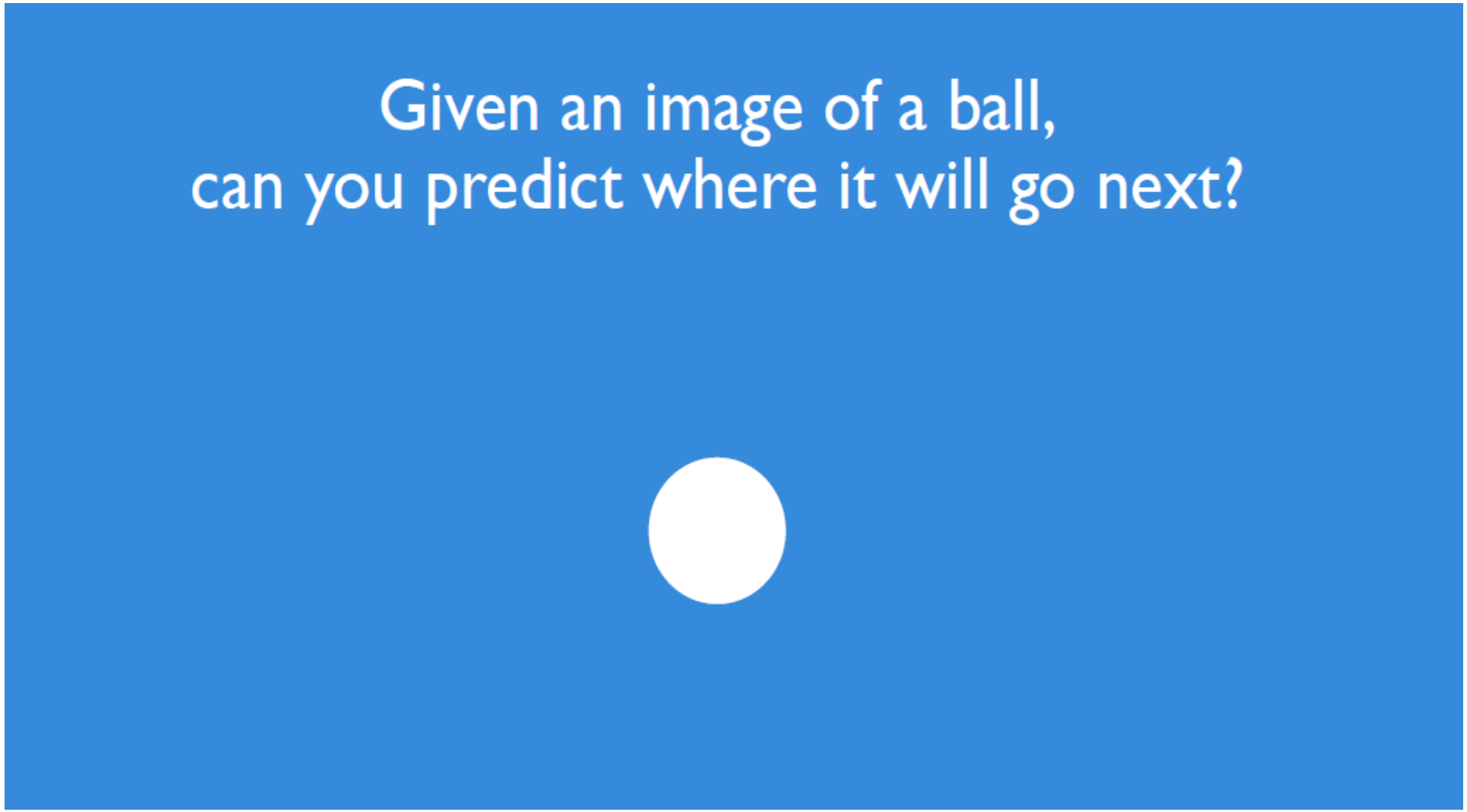
What are RNNs?

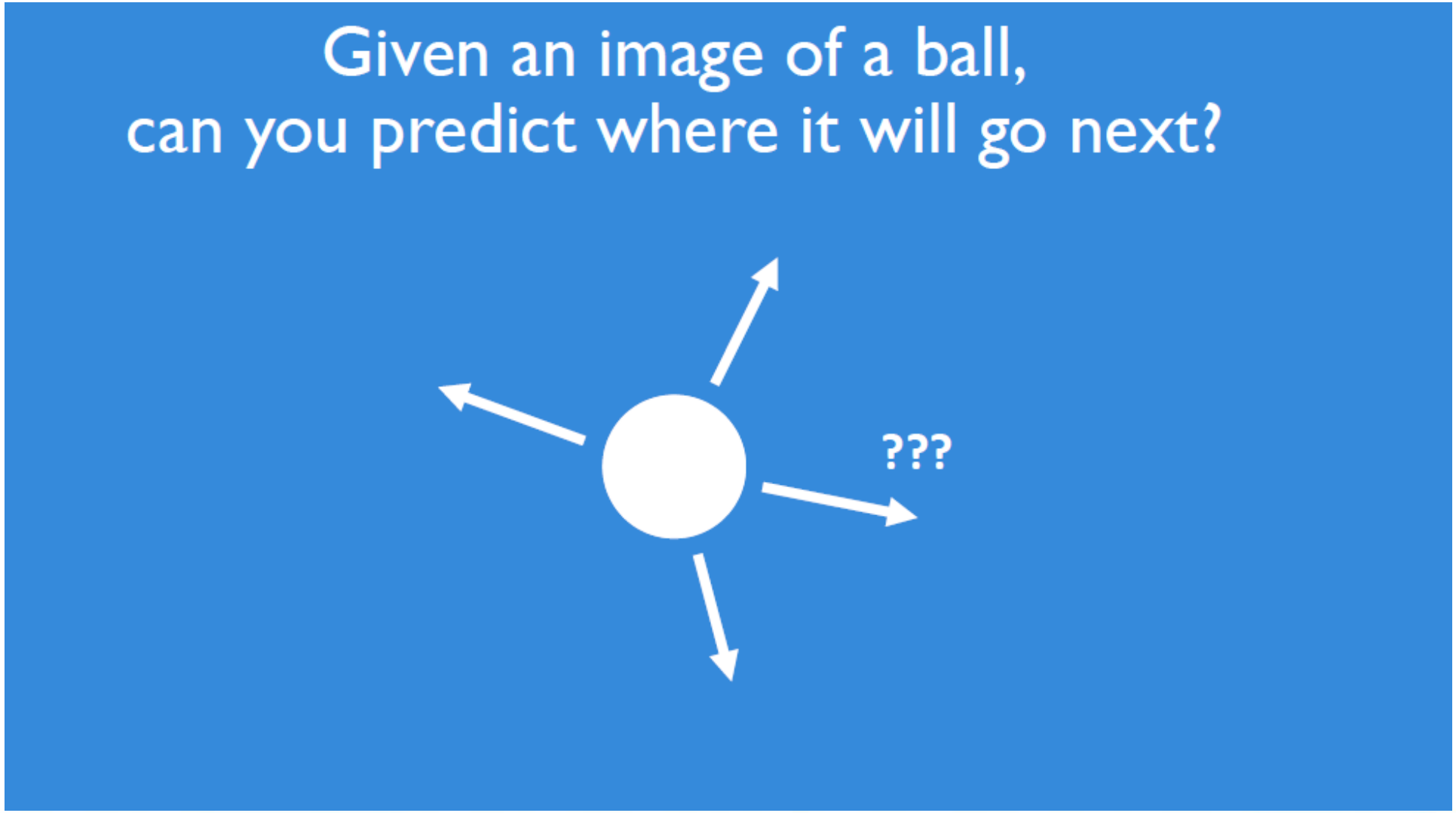
- Recurrent neural networks (RNNs) are a class of nets that can make predictions
- RNNs can work on sequences of arbitrary lengths, rather than on fixed-sized inputs like all the nets we have considered so far
- RNNs can:
  - Take sentences, documents, or audio samples as input, making them extremely useful for natural language processing applications such as automatic translation or speech-to-text
  - Analyze time series data such as stock prices, and tell you when to buy or sell
  - In autonomous driving systems, they can anticipate car trajectories and help avoid accidents
- RNNs are trained using backpropagation through time
- RNNs have a memory

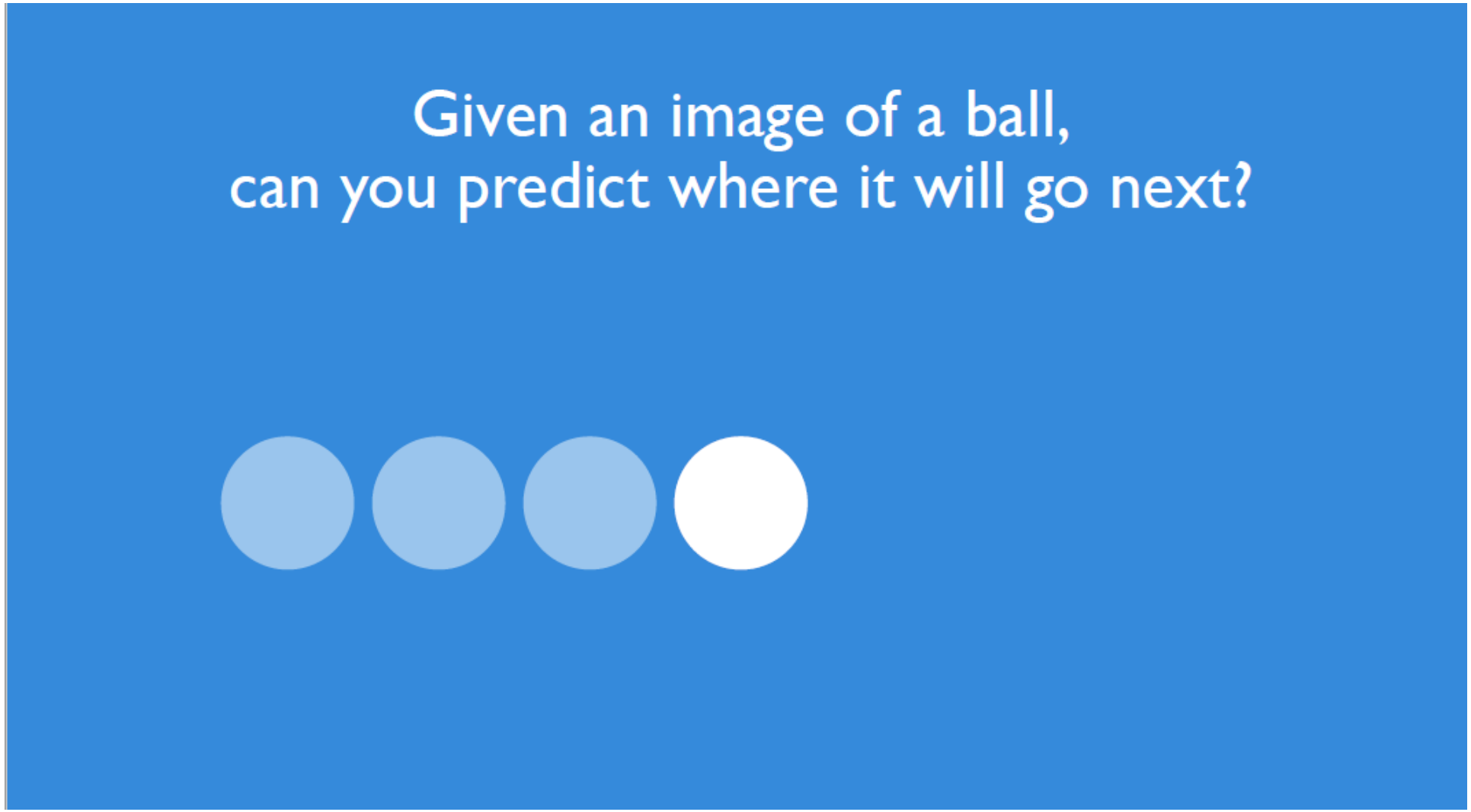
Learning Time

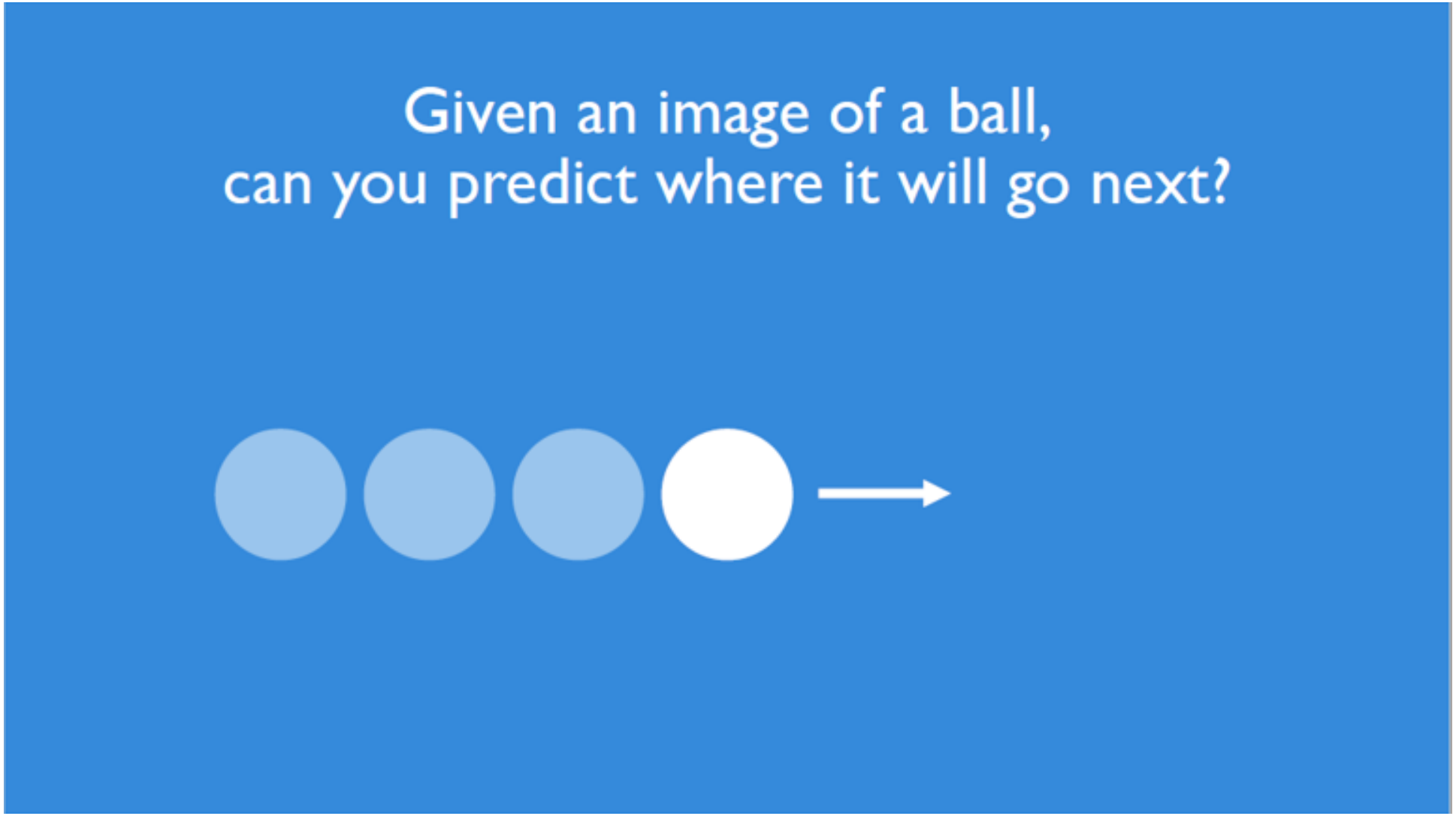
- Applications:
  - Sequence recognition: Speech recognition
  - Sequence reproduction: Time-series prediction
  - (Temporal) Sequence association: Sequence generation

- Network architectures
  - Time-delay networks (Waibel et al., 1989)
  - Recurrent networks (Rumelhart et al., 1986)

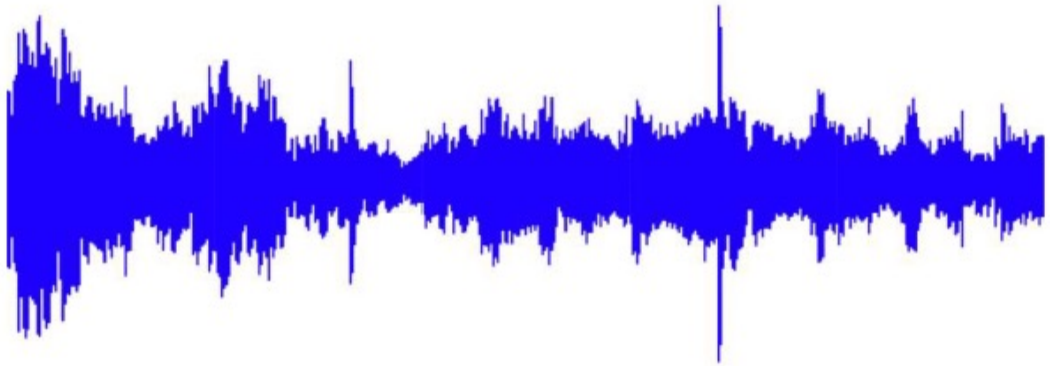

Given an image of a ball,
can you predict where it will go next?


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

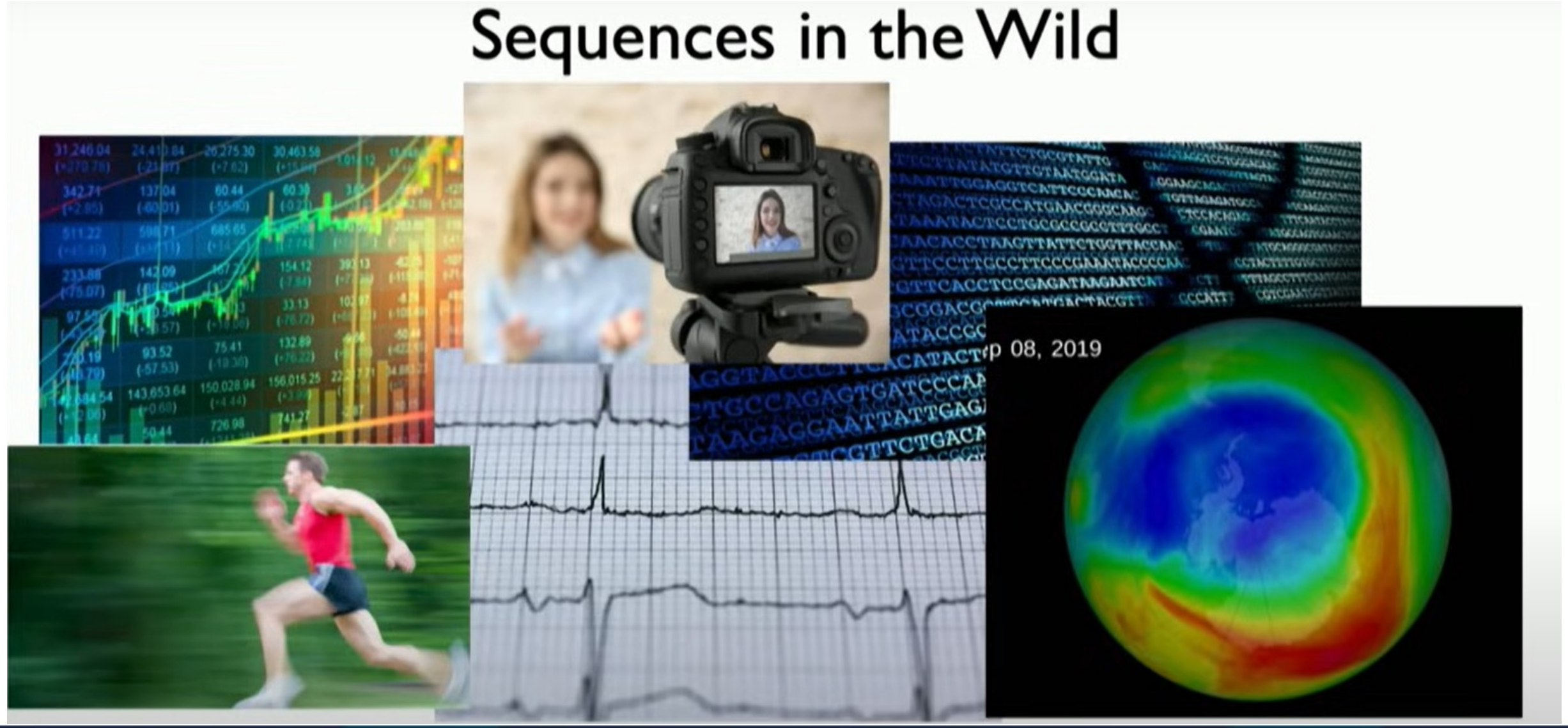
Sequences in the wild

character: 6 . S I 9 I

word: Introduction to Deep Learning

Text

Sequences in the Wild

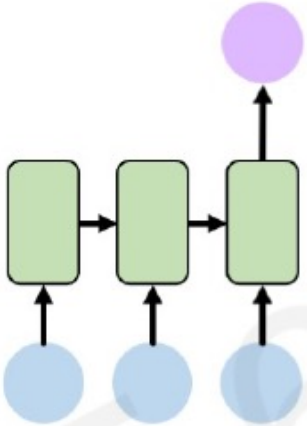


MIT 6.S191 – Deep Sequence Modeling, Introtodeeplearning.com

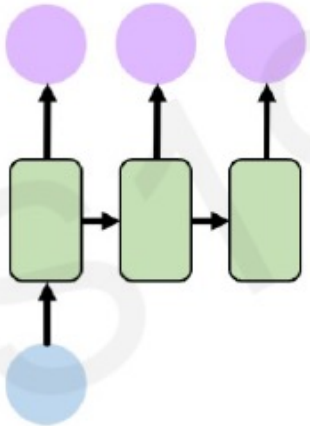
Sequence Modeling Applications

One to One
Binary Classification

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

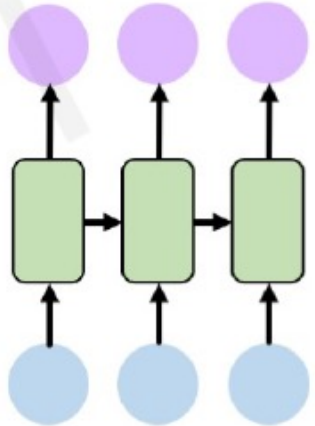


Many to One
Sentiment Classification



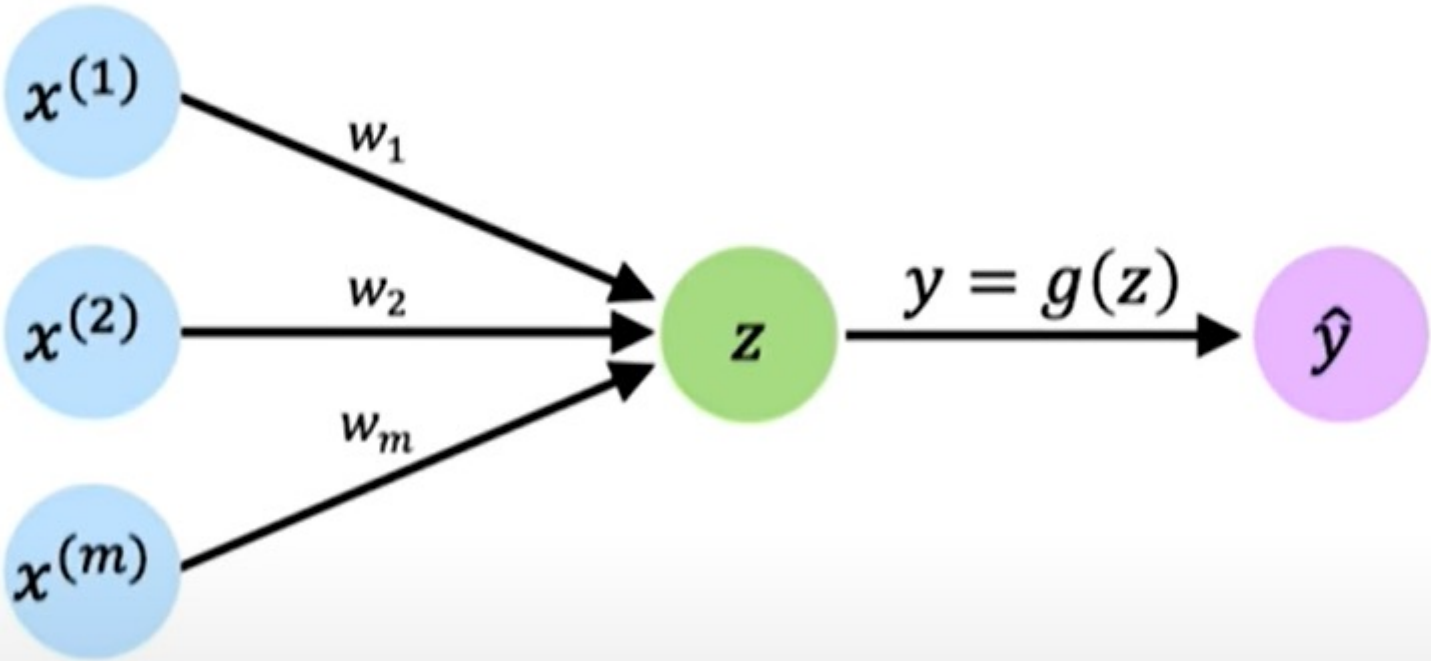
One to Many
Image Captioning

"A baseball player throws a ball."

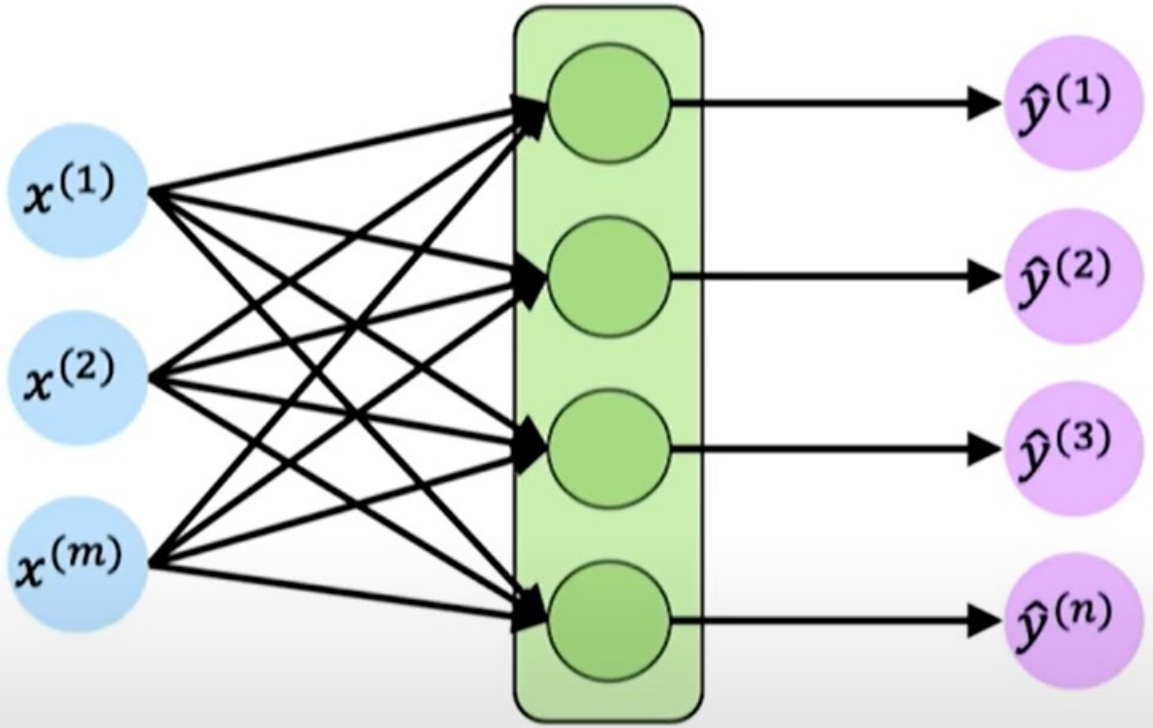


Many to Many
Machine Translation

The Perceptron Revisited

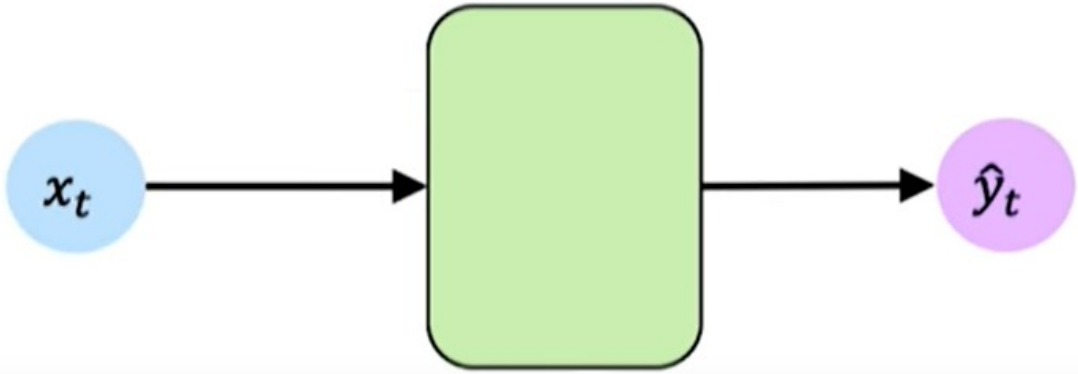


Feed-Forward Networks Revisited



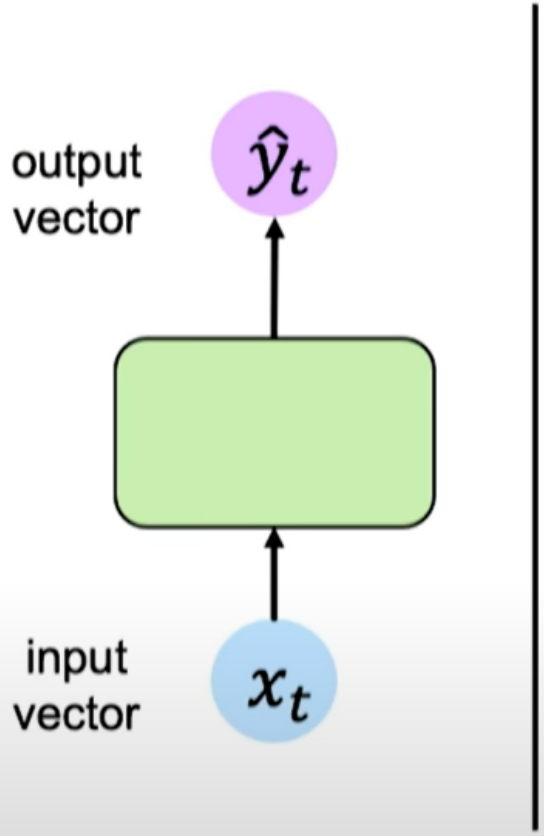
$\boldsymbol{x} \in \mathbb{R}^m$ $\quad \hat{\boldsymbol{y}} \in \mathbb{R}^n$

Feed-Forward Networks Revisited

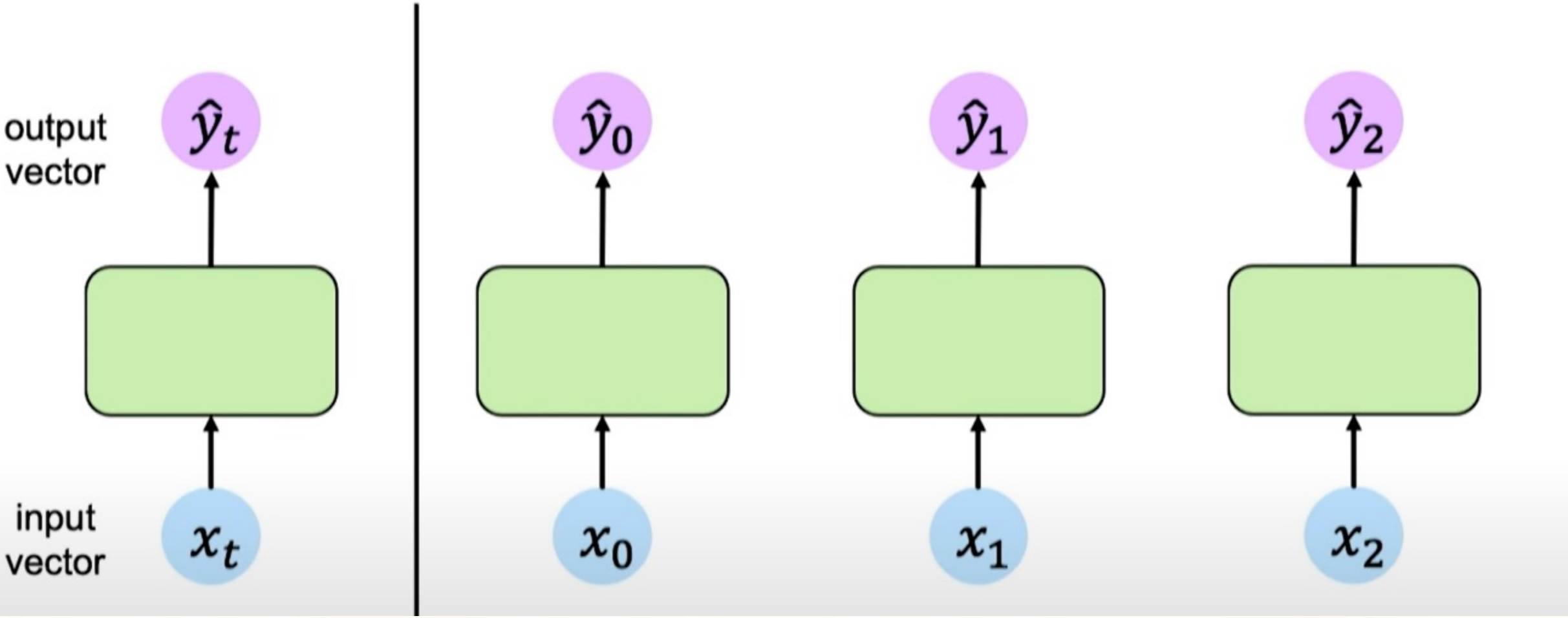


$$x_t \in \mathbb{R}^m \qquad \hat{y}_t \in \mathbb{R}^n$$

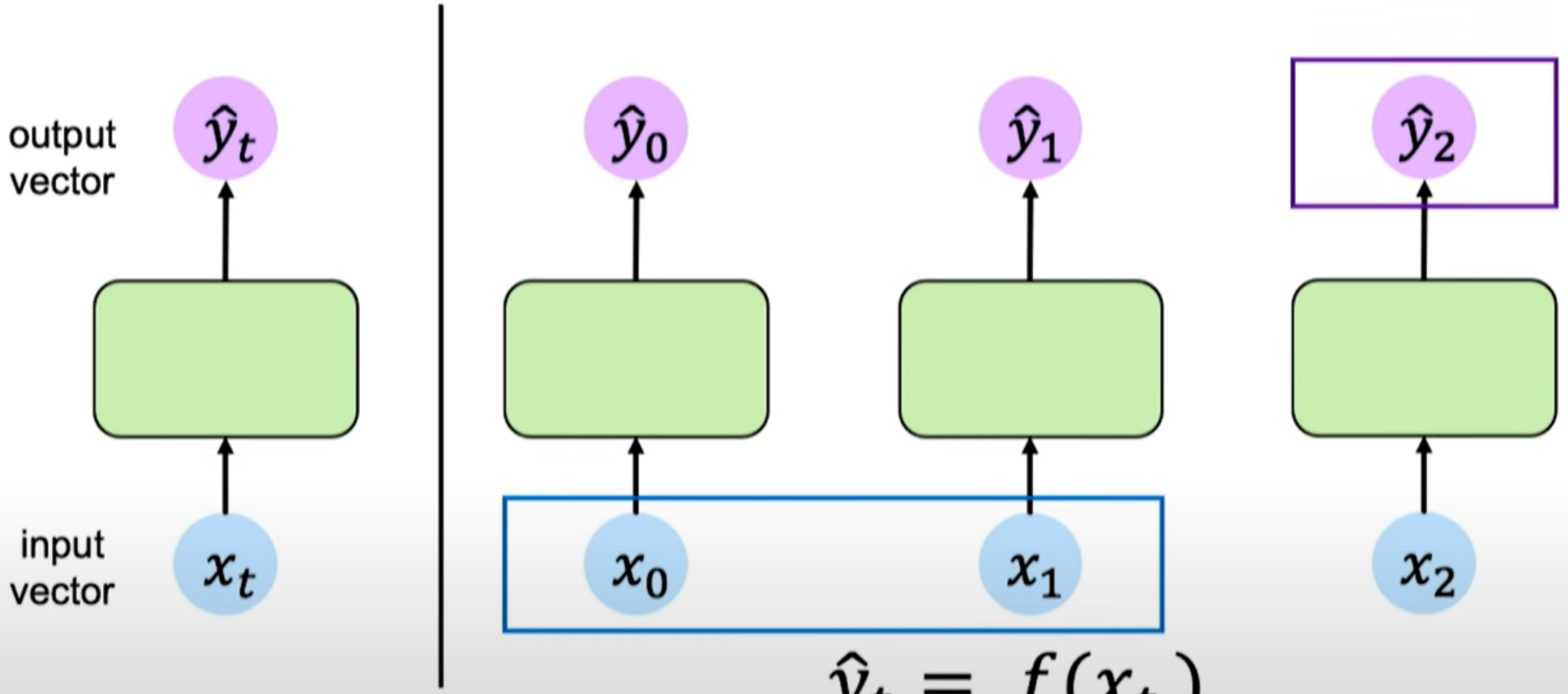
Handling Individual Time Steps



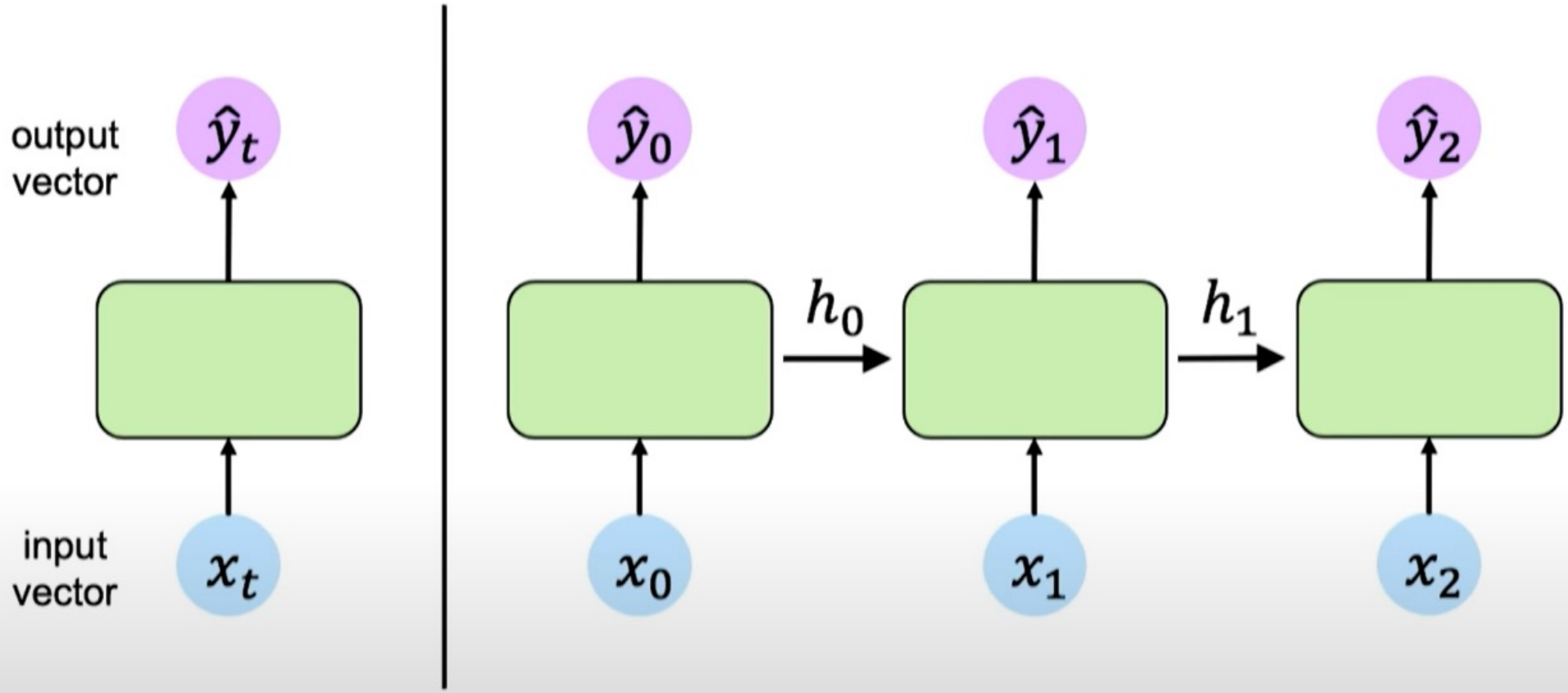
Handling Individual Time Steps



Handling Individual Time Steps

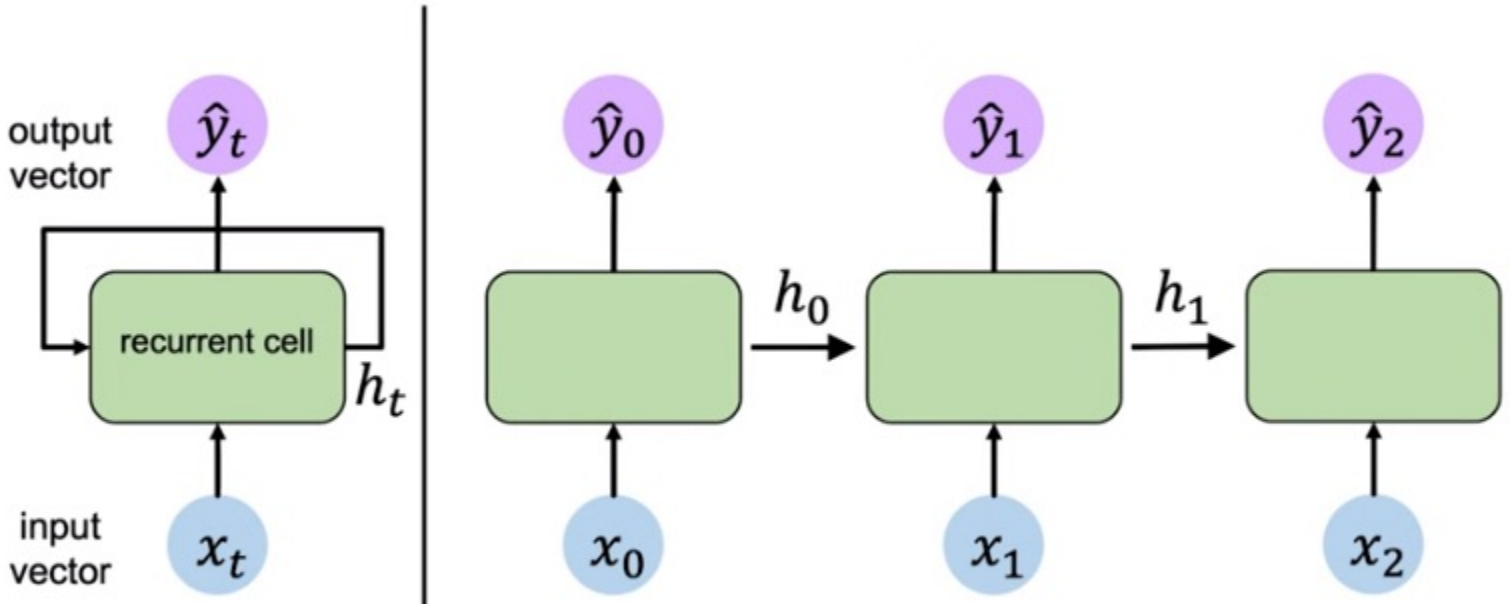


Neurons with Recurrence



Neurons with Recurrence

Unrolling the network in time



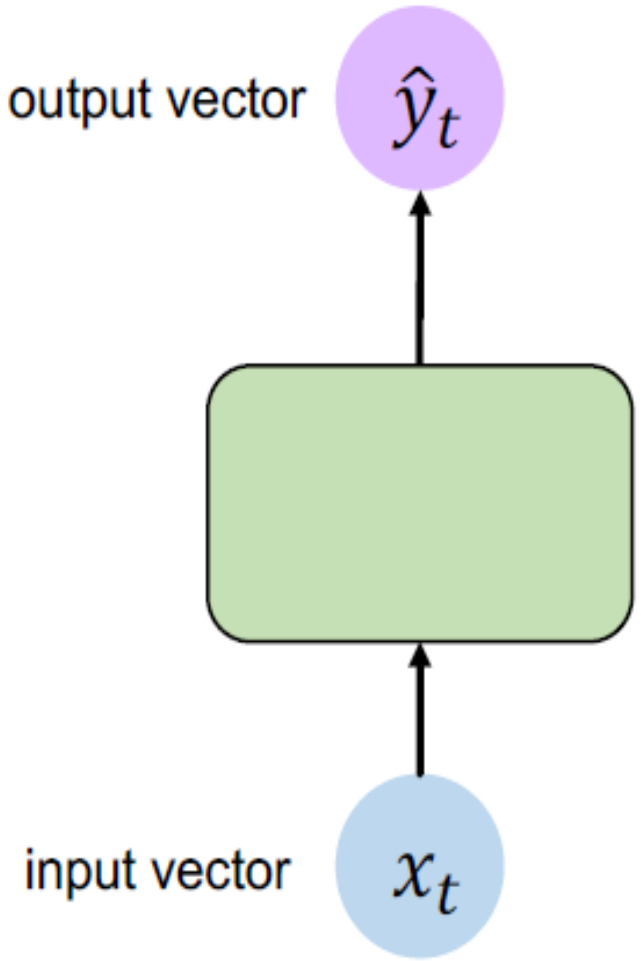
THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

Recurrent Neural Networks (RNNs) - Principles of Operation

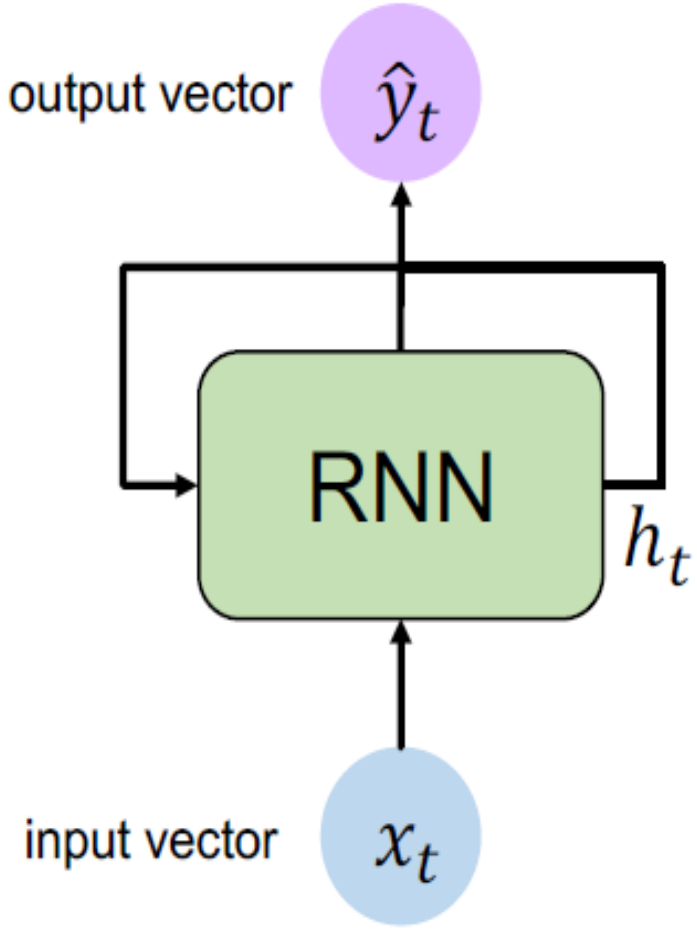
MIT 6.S191 – Deep Sequence Modeling, Introtodeeplearning.com

Professor Hamza F. Alsarhan
SEAS 8505

A standard "vanilla" neural network

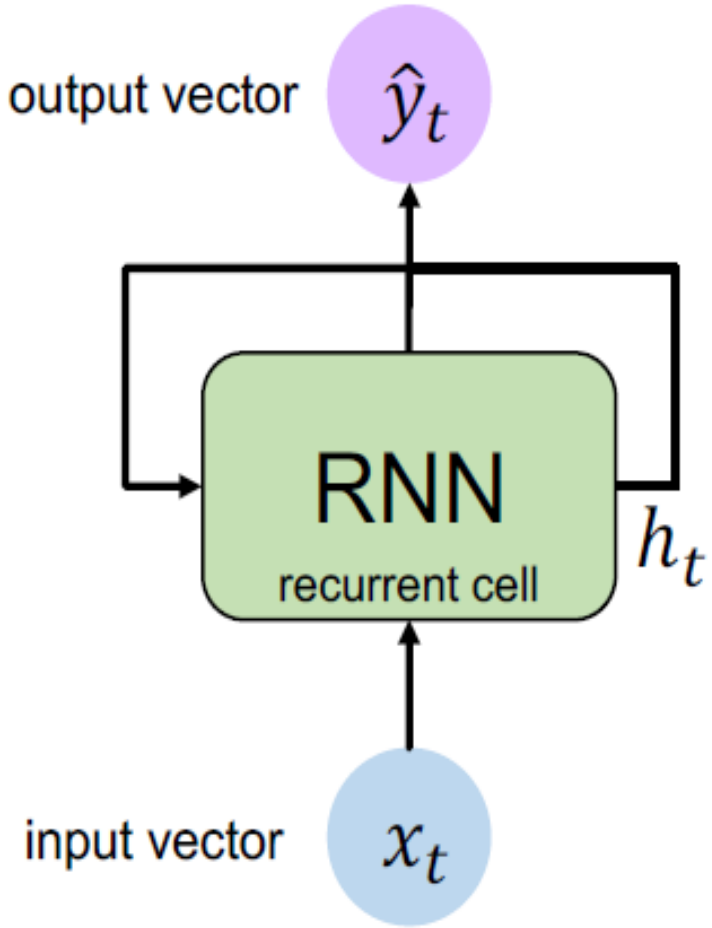


A recurrent neural network (RNN)

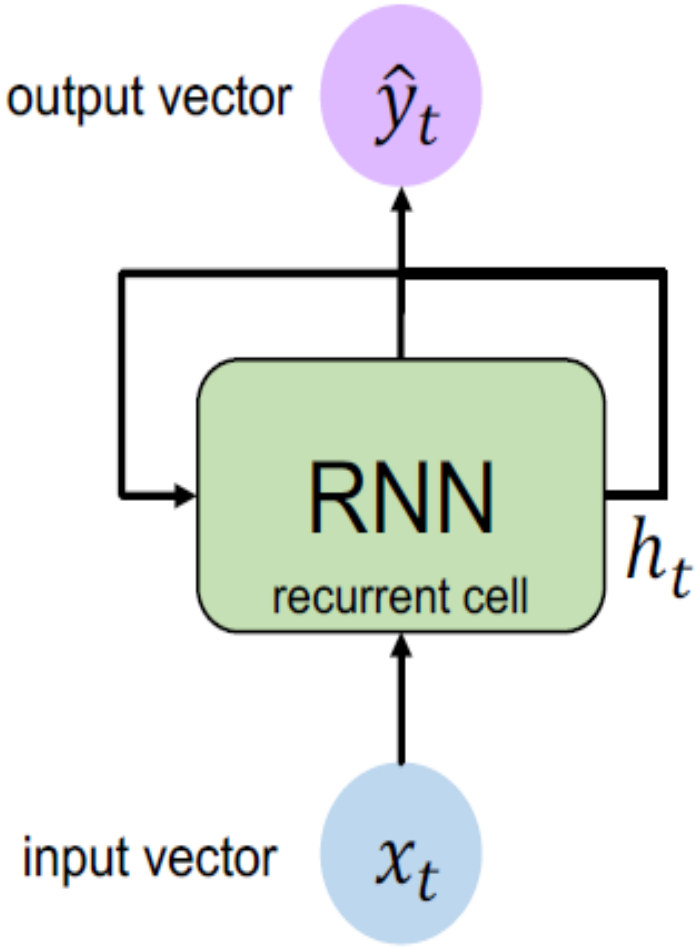


The loop allow information to persist

A recurrent neural network (RNN)



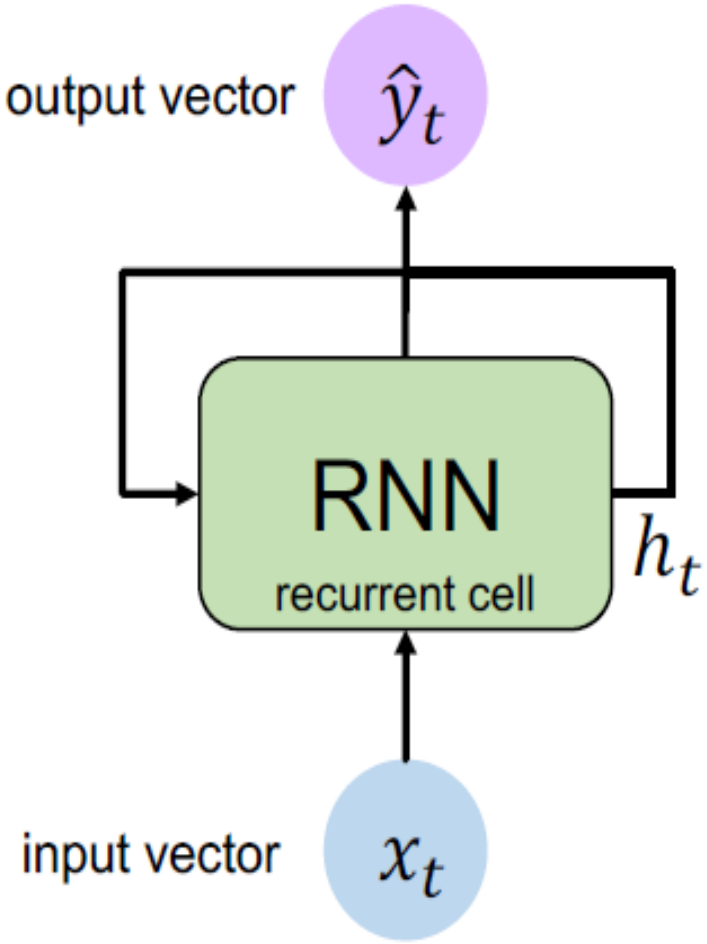
A recurrent neural network (RNN)



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

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

A recurrent neural network (RNN)



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

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

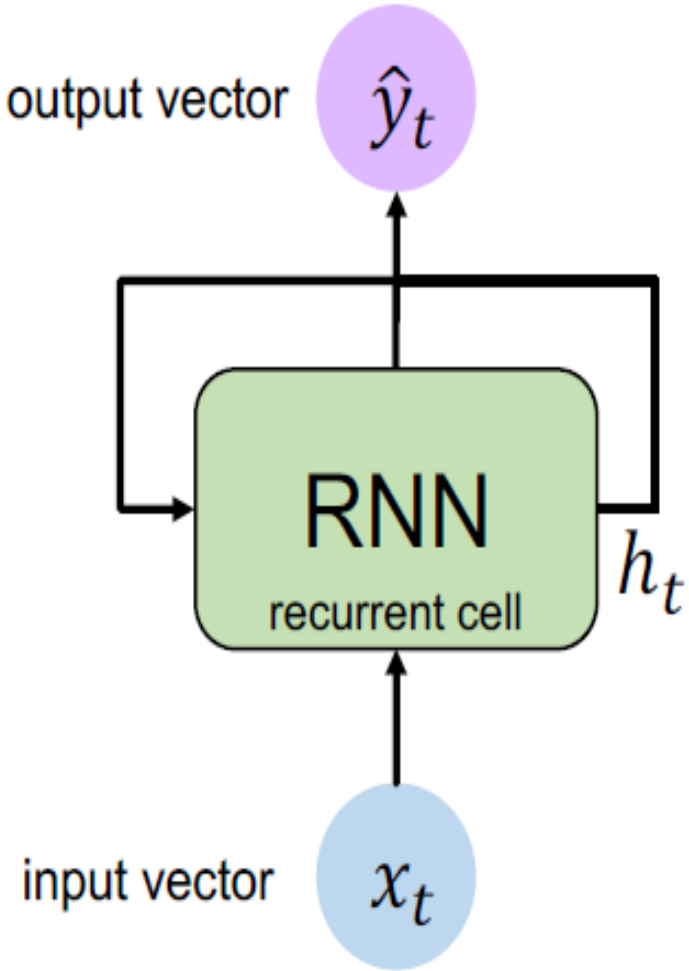
cell state

function parameterized by W

old state

input vector at time step t

A recurrent neural network (RNN)



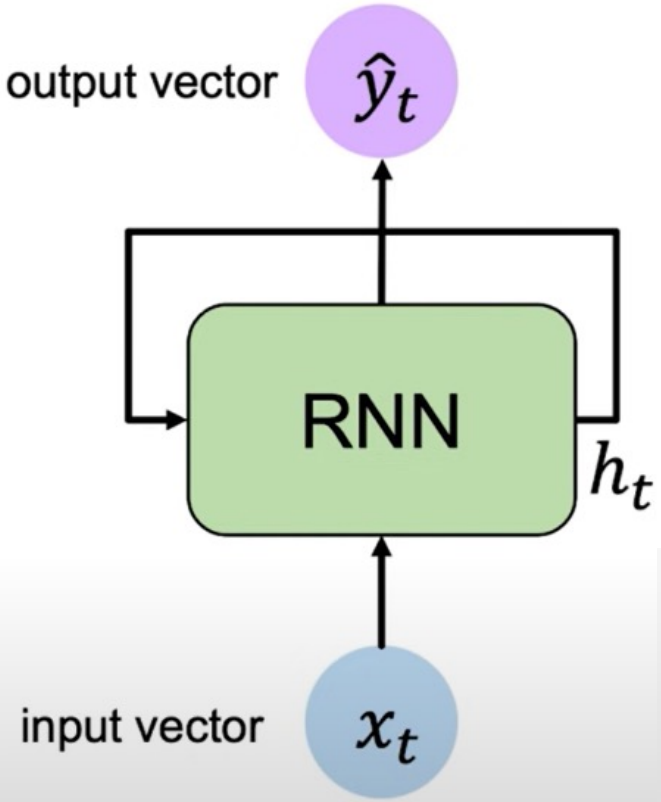
Apply a **recurrence relation** at every time step to process a sequence:

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

new state — function parameterized by W — old state — input vector at time step t

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

Recurrent Neural Networks (RNNs)



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

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

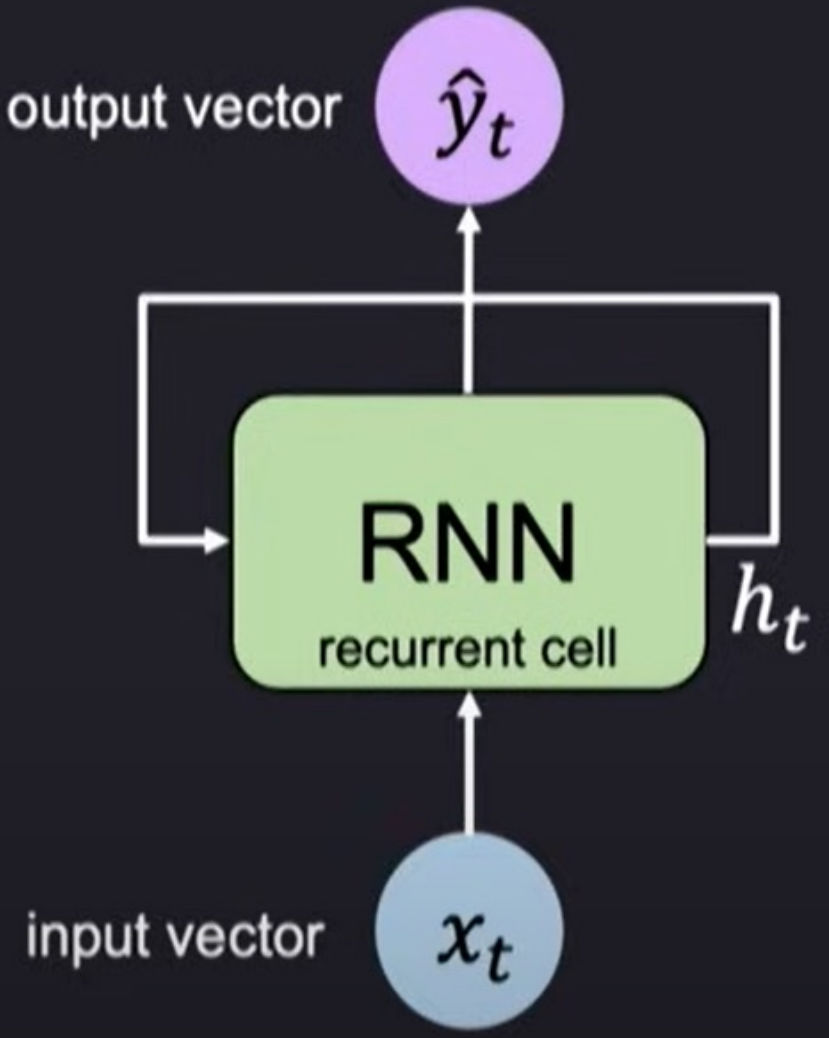
cell state | function with weights W | input | old state

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

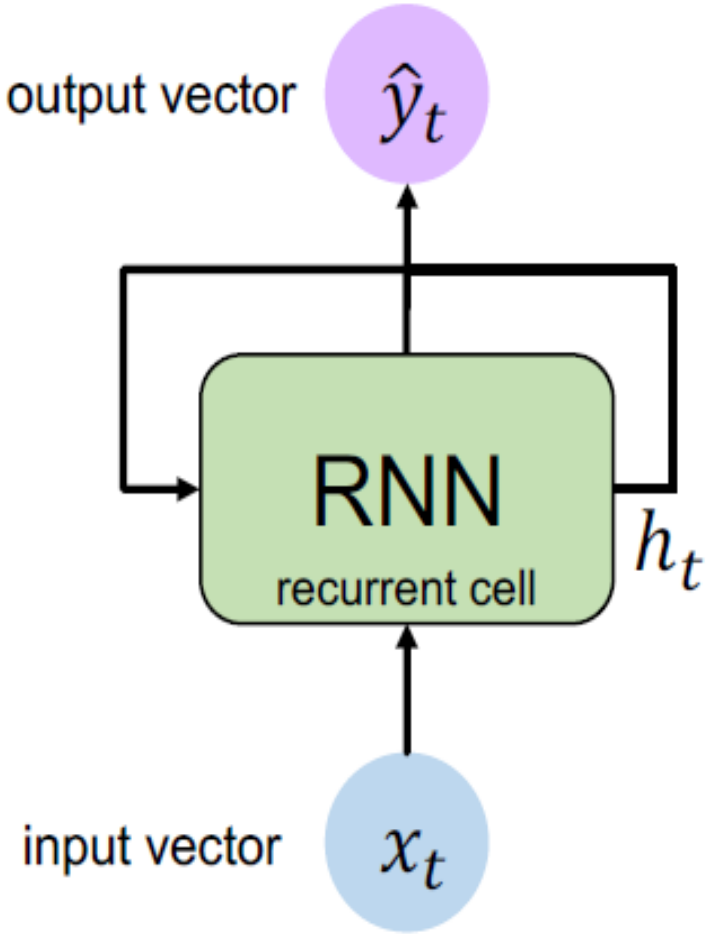
RNNs have a **state**, h_t, that is updated **at each time step** as a sequence is processed

RNN Intuition

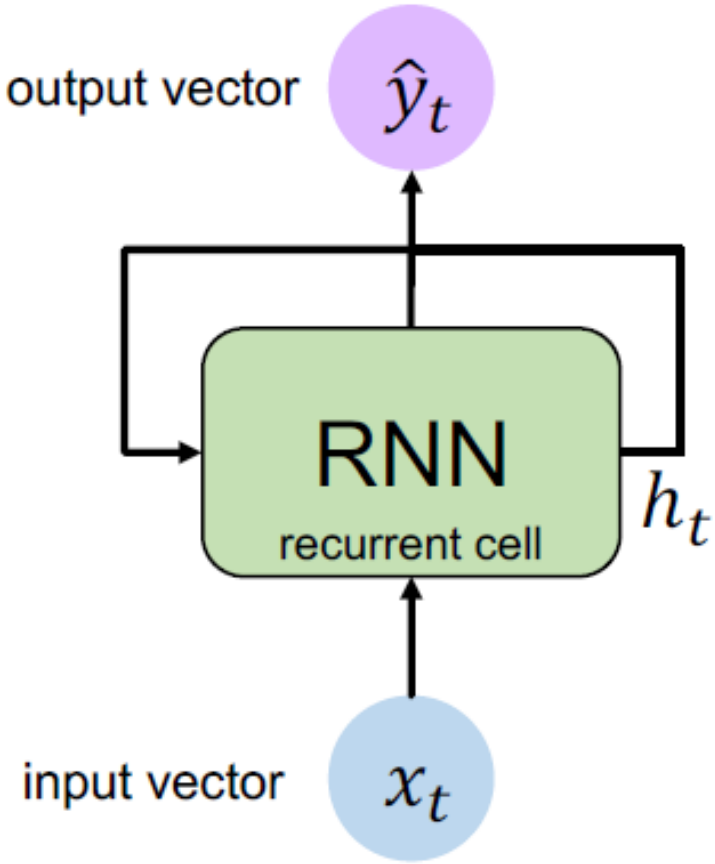
```
my_rnn = RNN()
hidden_state = [0, 0, 0, 0]

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

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

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



RNN state update and output



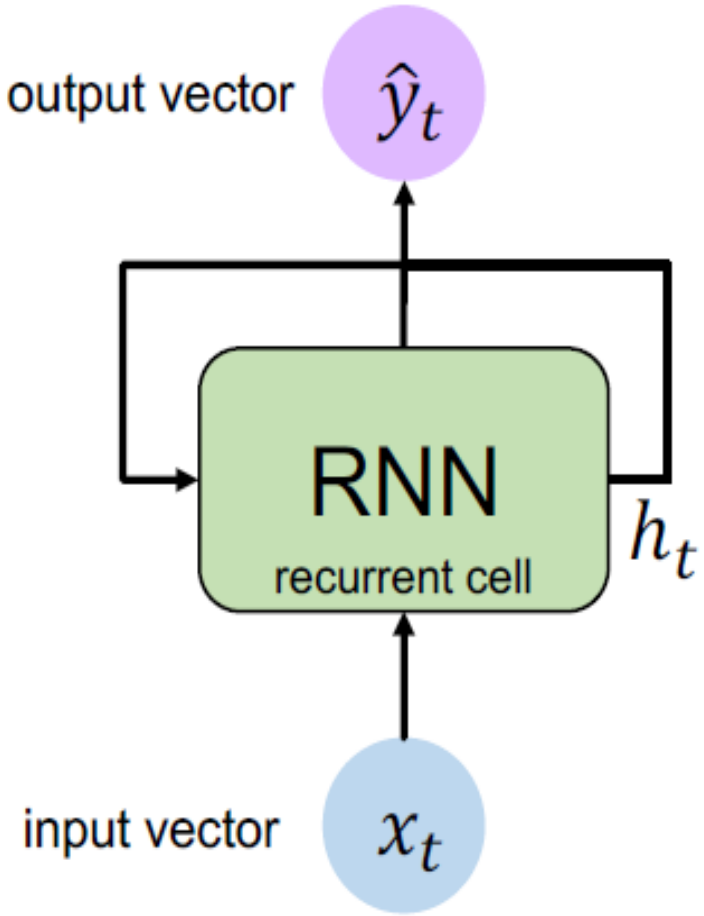
RNN state update and output



Input Vector

x_t

RNN state update and output



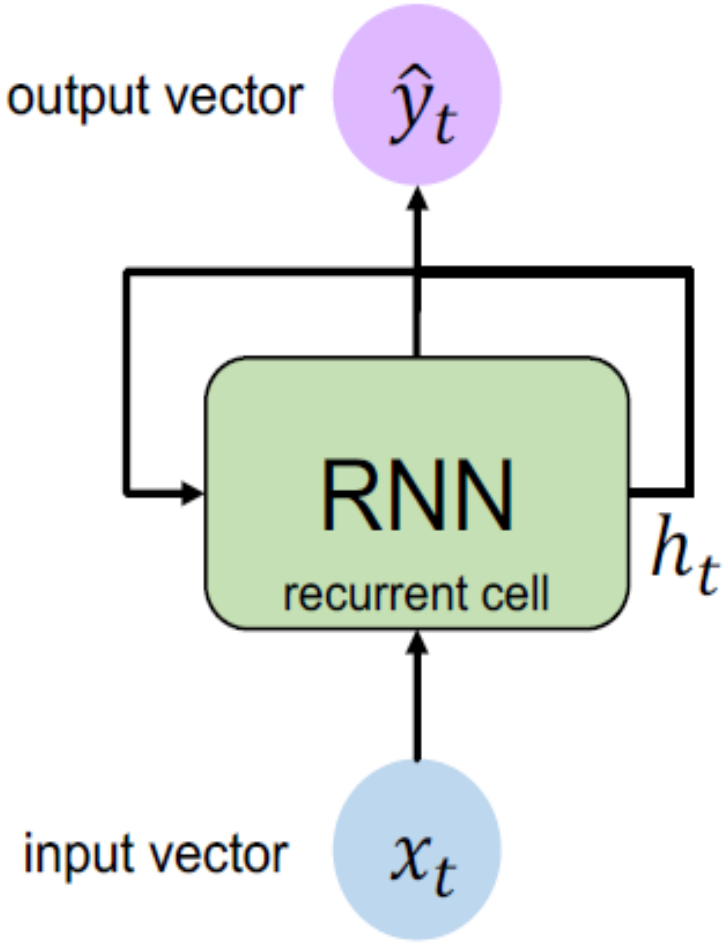
Update Hidden State

$$h_t = \tanh(\boldsymbol{W_{hh}} h_{t-1} + \boldsymbol{W_{xh}} x_t)$$

Input Vector

x_t

RNN state update and output



Output Vector

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

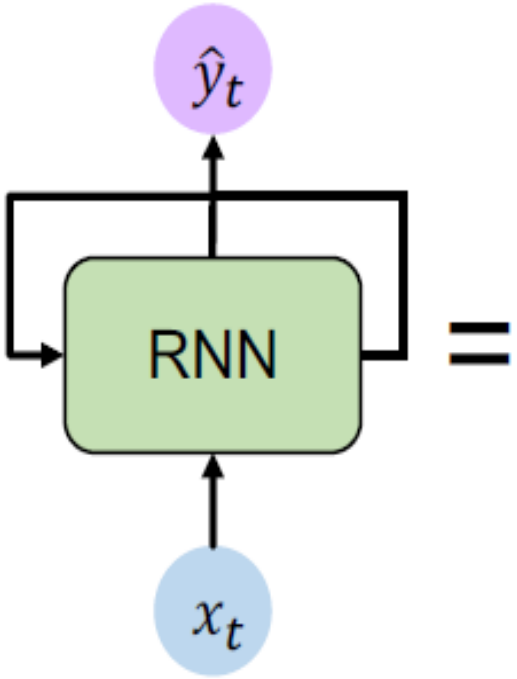
Update Hidden State

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

Input Vector

x_t

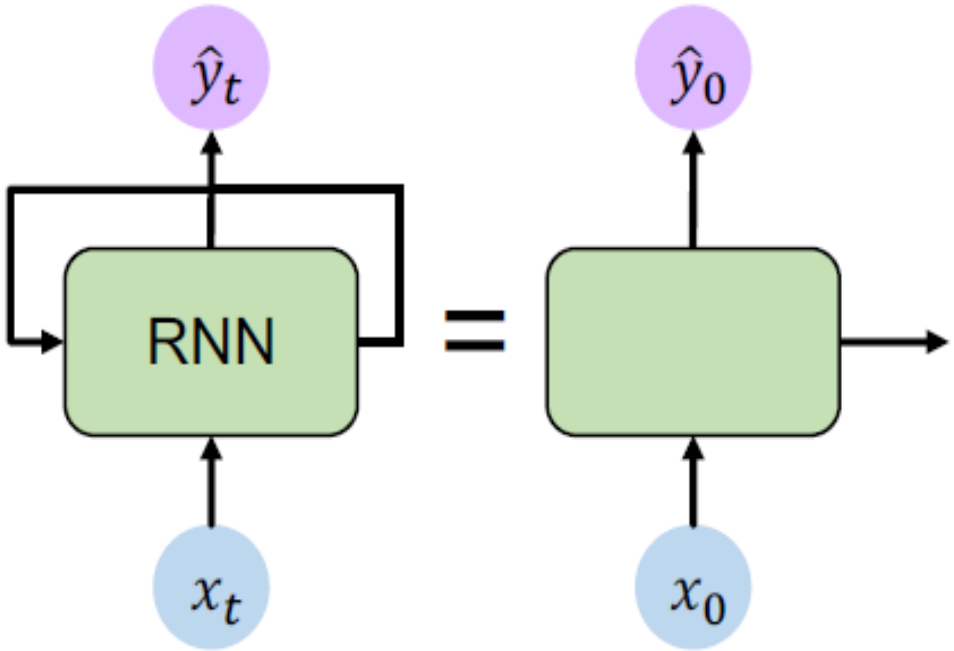
RNNs: computational graph across time



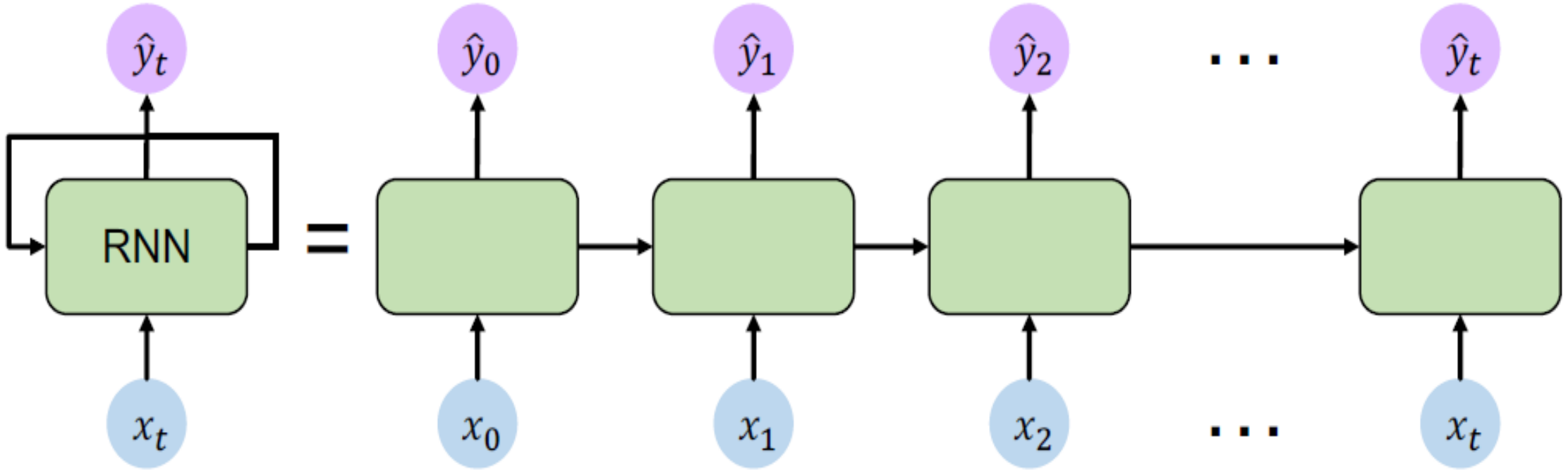
=

Represent as computational graph unrolled across time

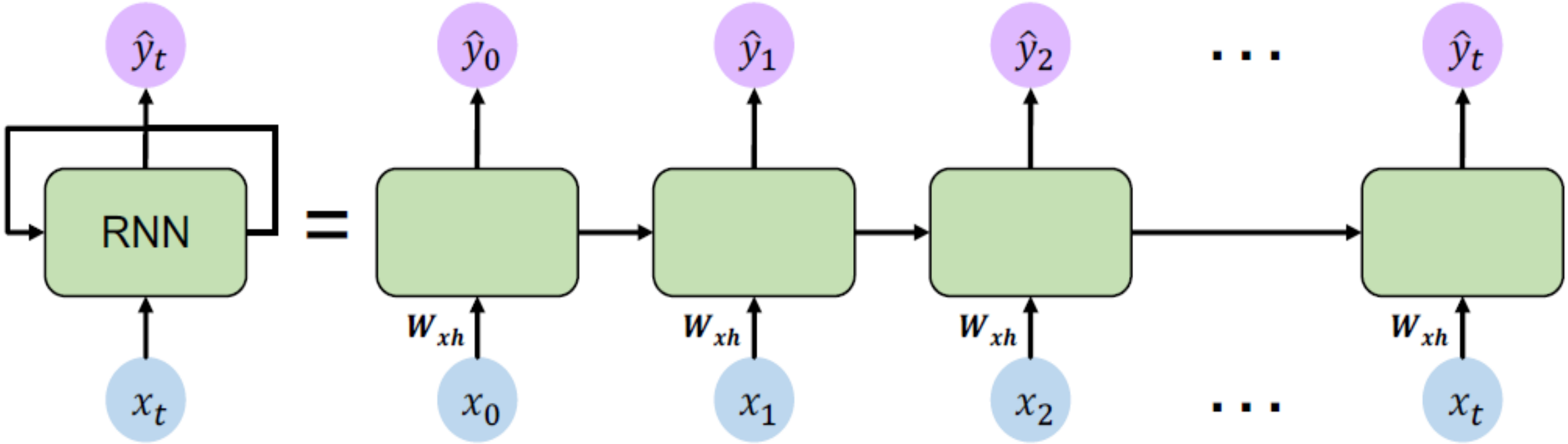
RNNs: computational graph across time



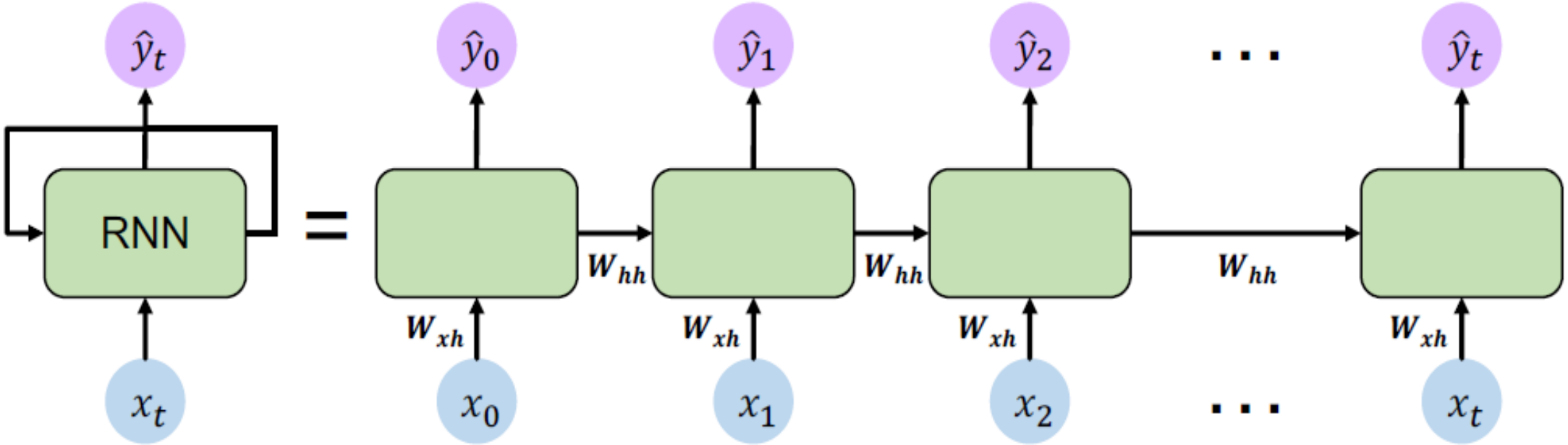
RNNs: computational graph across time



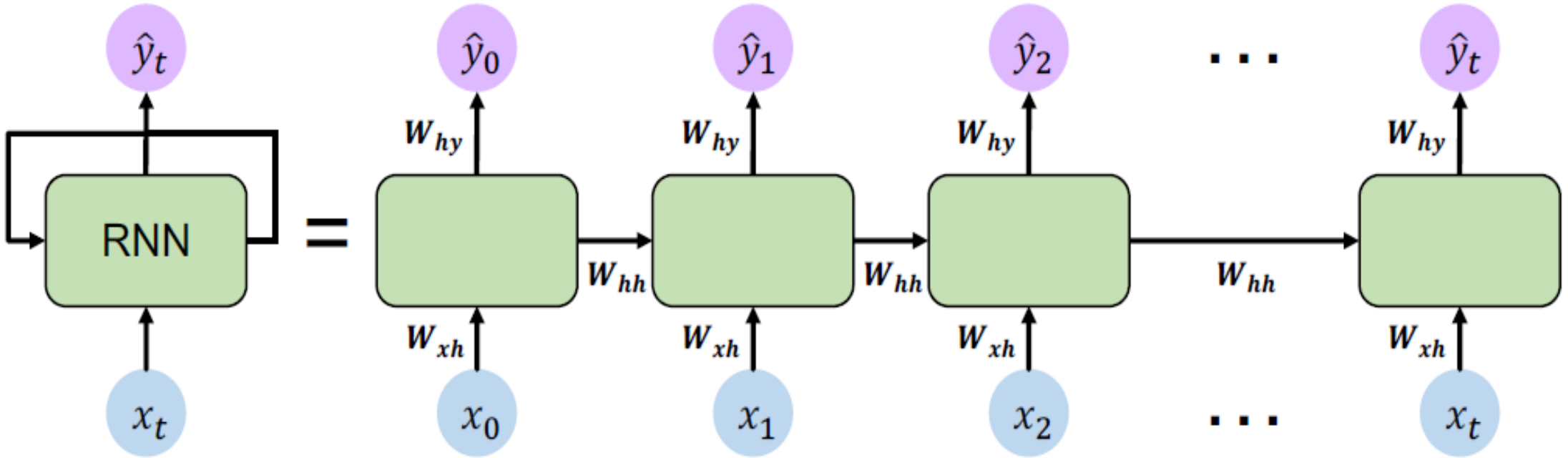
RNNs: computational graph across time



RNNs: computational graph across time

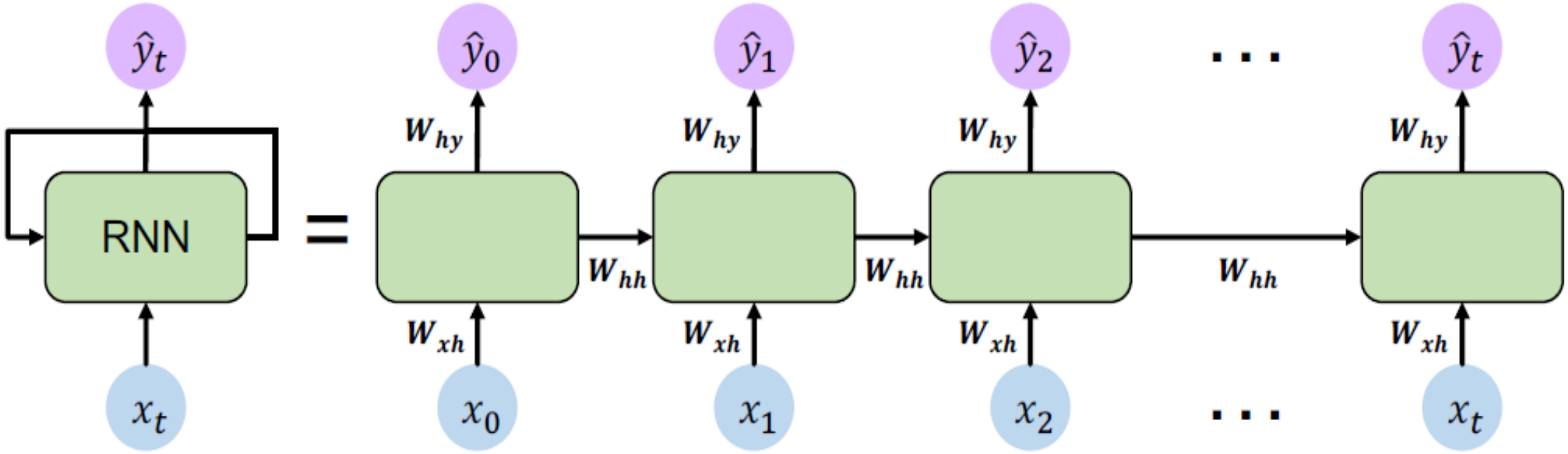


RNNs: computational graph across time

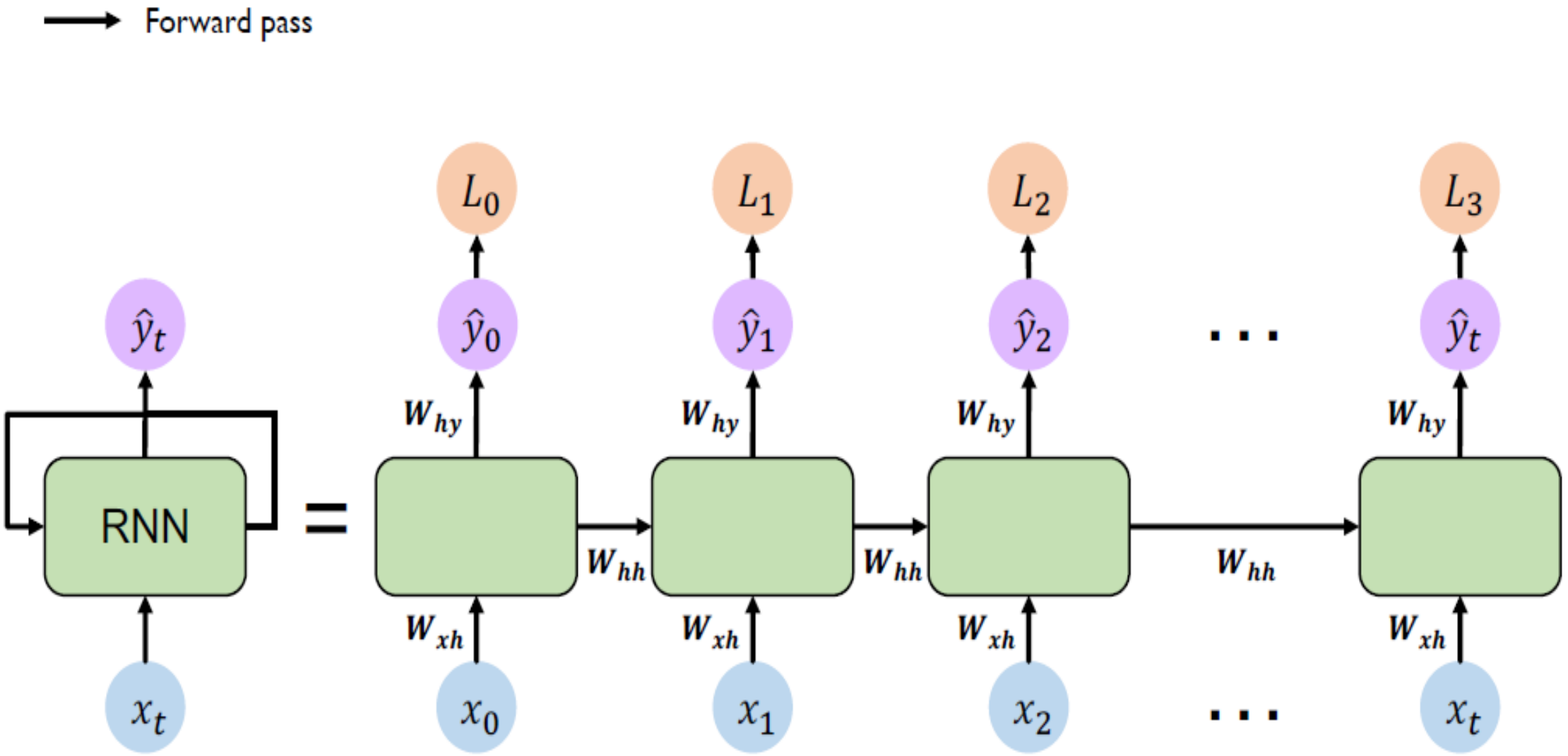


RNNs: computational graph across time

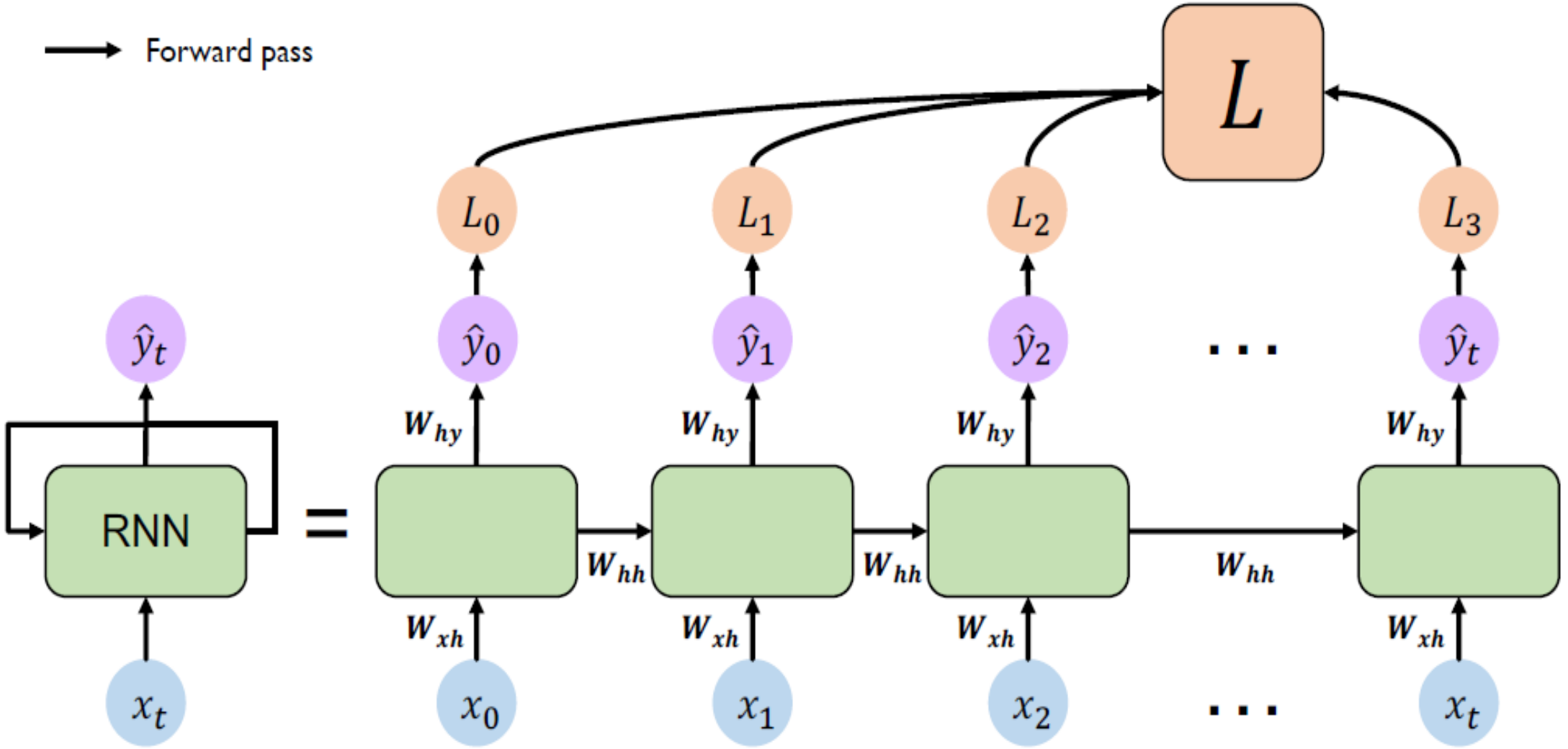
Re-use the **same weight matrices** at every time step



RNNs: computational graph across time

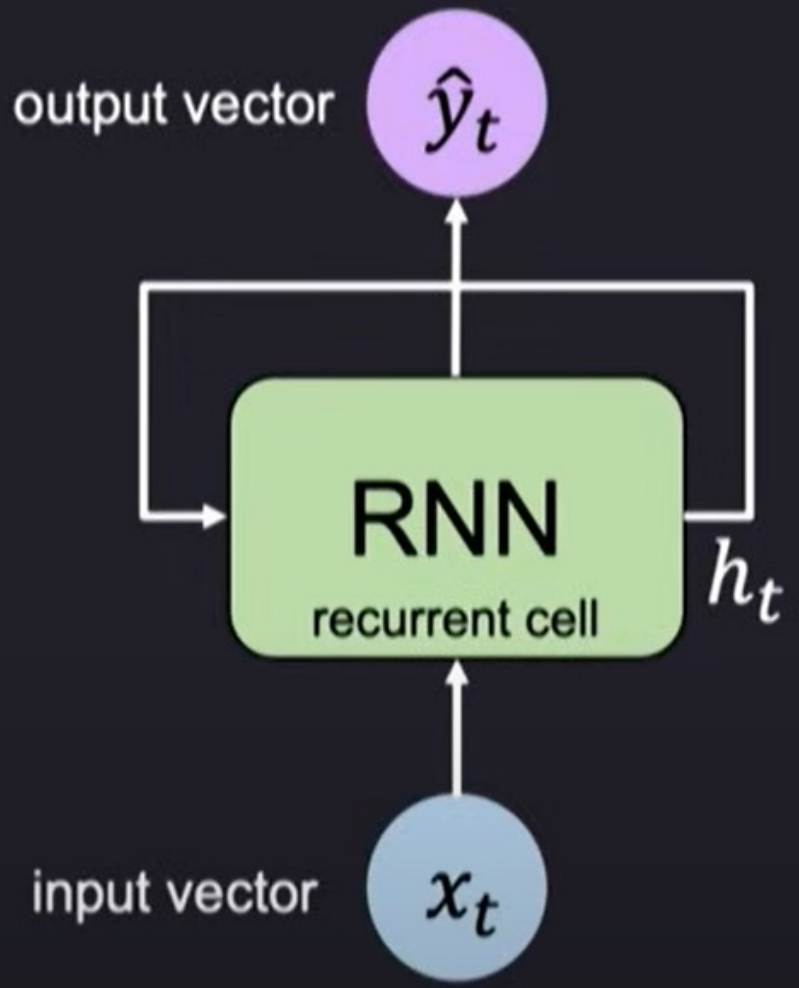


RNNs: computational graph across time



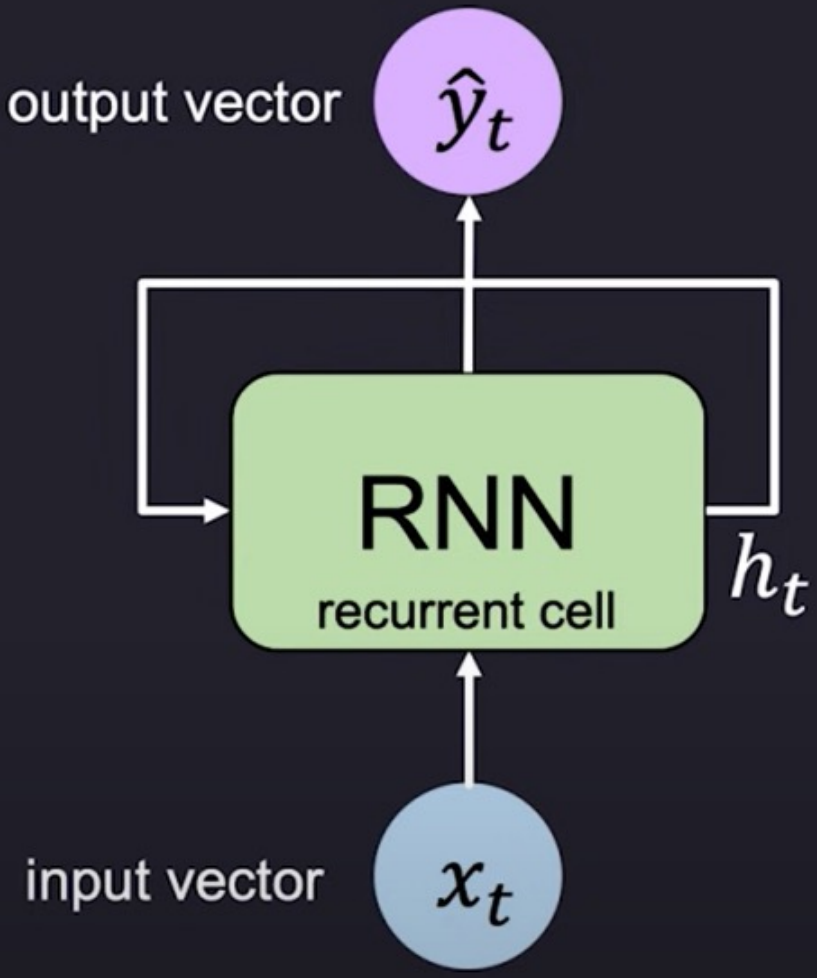
RNNs from Scratch

```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])
```



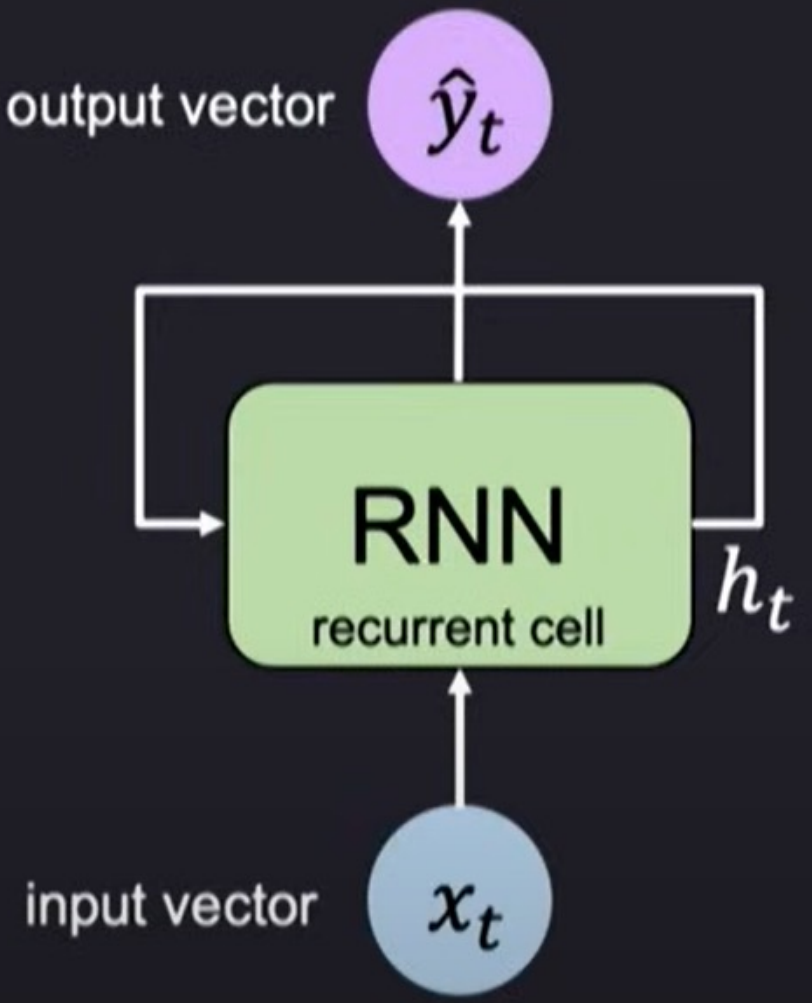
RNNs from Scratch

```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
```



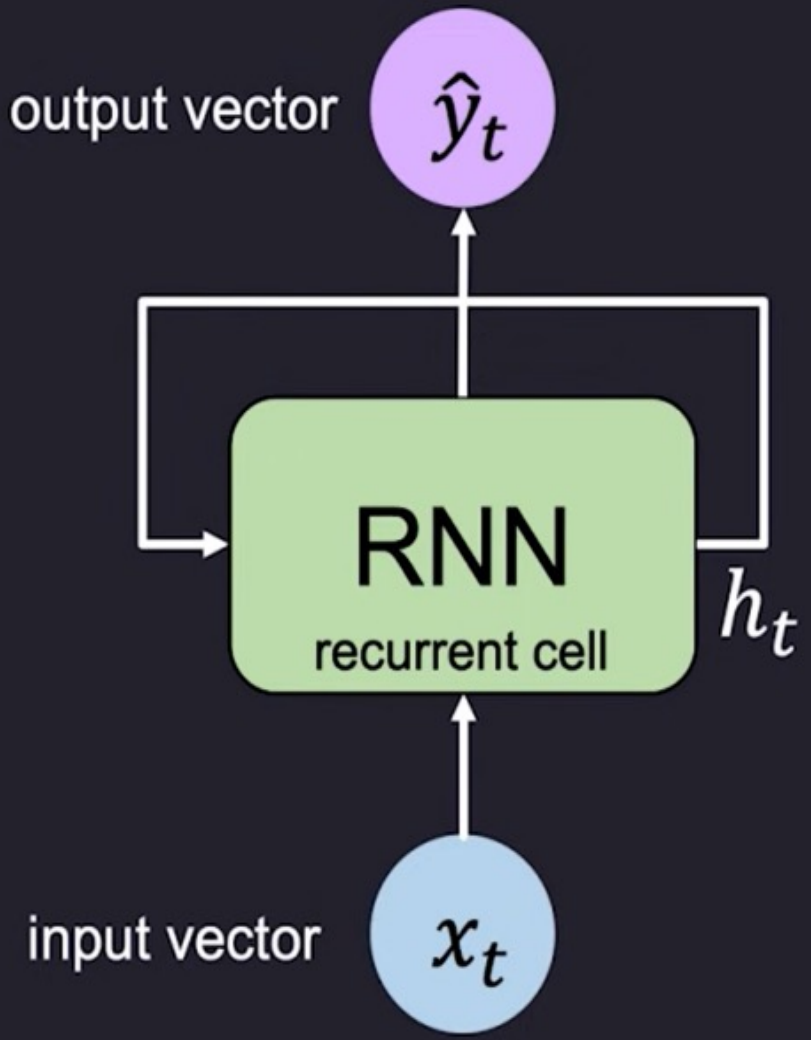
RNNs from Scratch

```
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
```

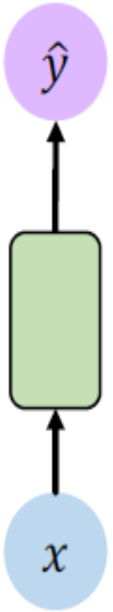


RNN Implementation in TensorFlow

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

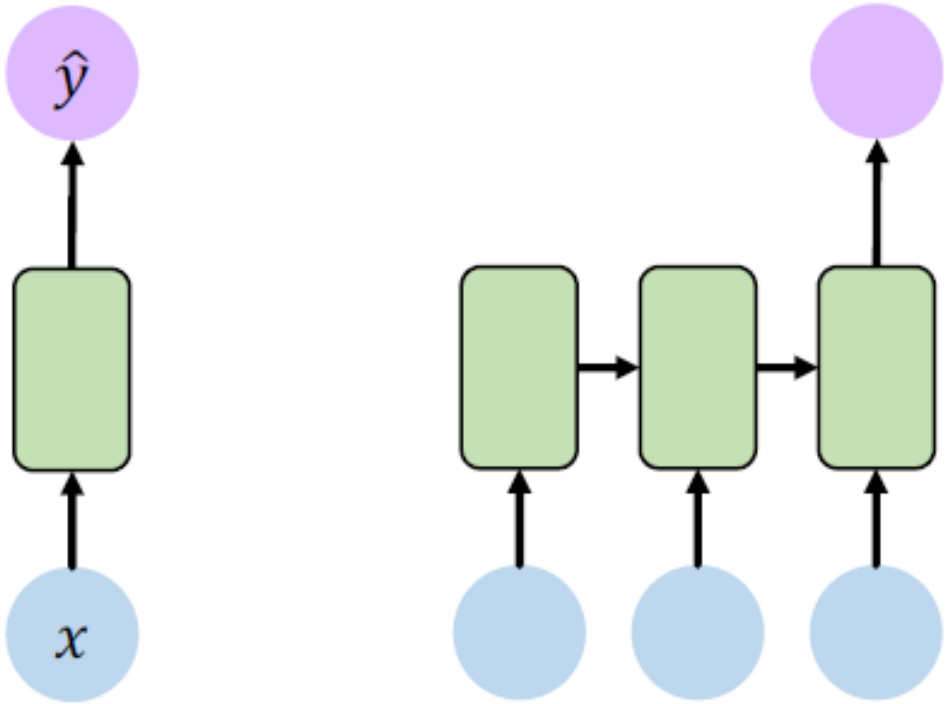


Standard feed-forward neural network



One to One
"Vanilla" neural network

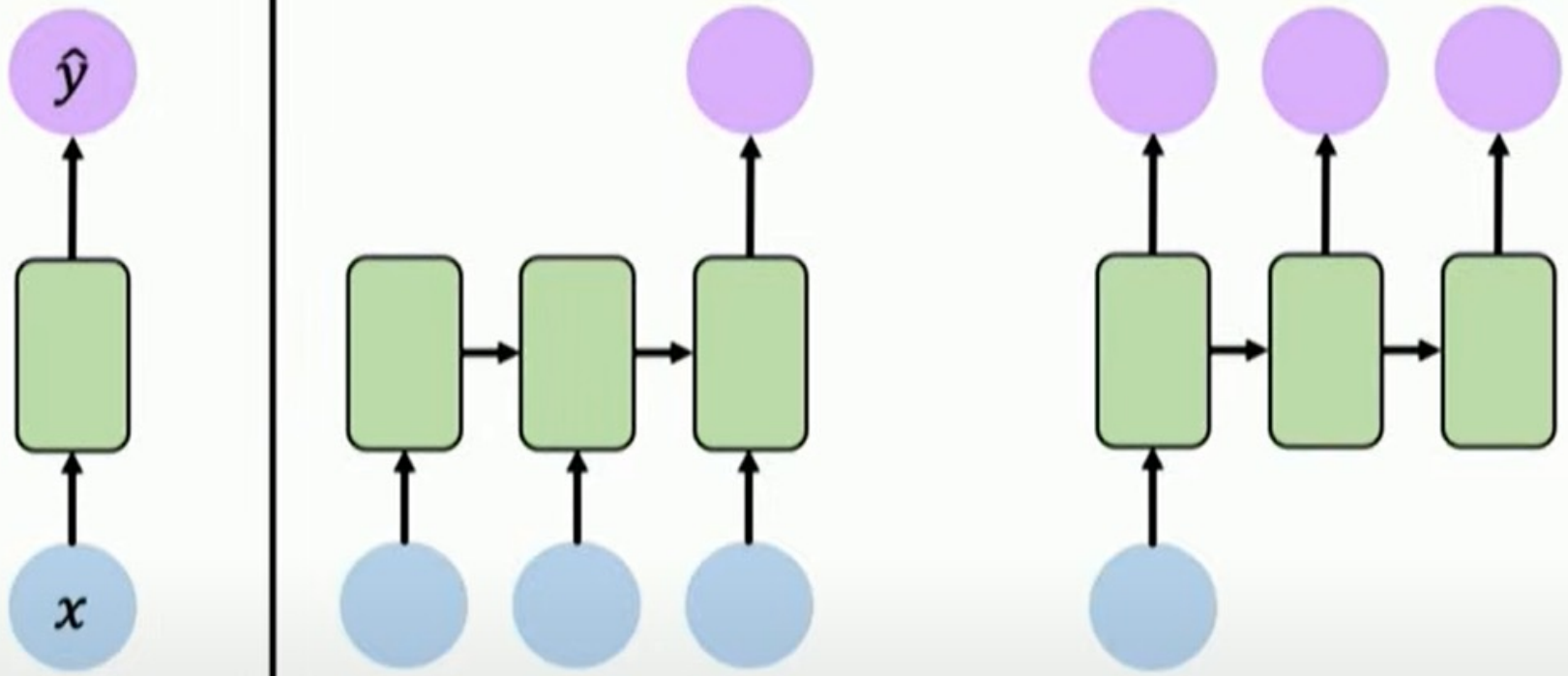
Recurrent neural networks: sequence modeling



One to One
"Vanilla" neural network

Many to One
Sentiment Classification

RNNs for Sequence Modeling

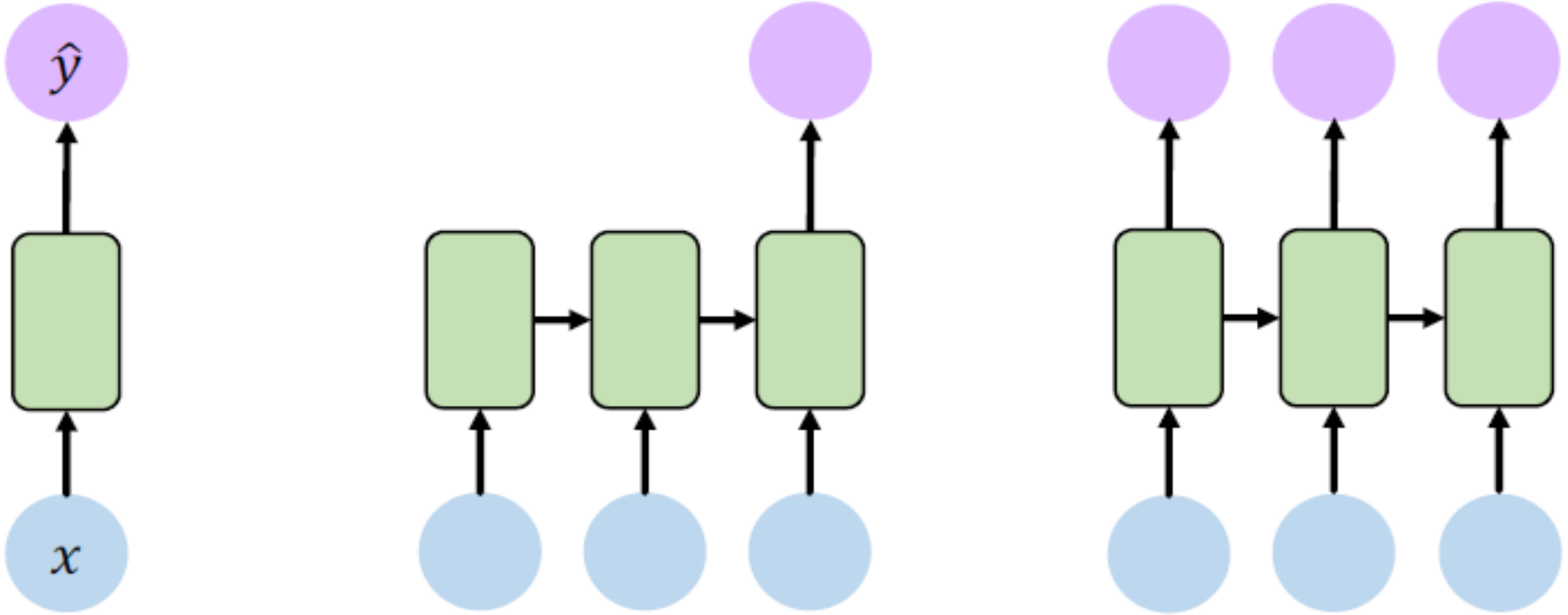


One to One
"Vanilla" NN
Binary classification

Many to One
Sentiment Classification

One to Many
Text Generation
Image Captioning

Recurrent neural networks: sequence modeling

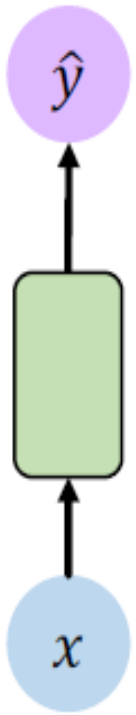


One to One
"Vanilla" neural network

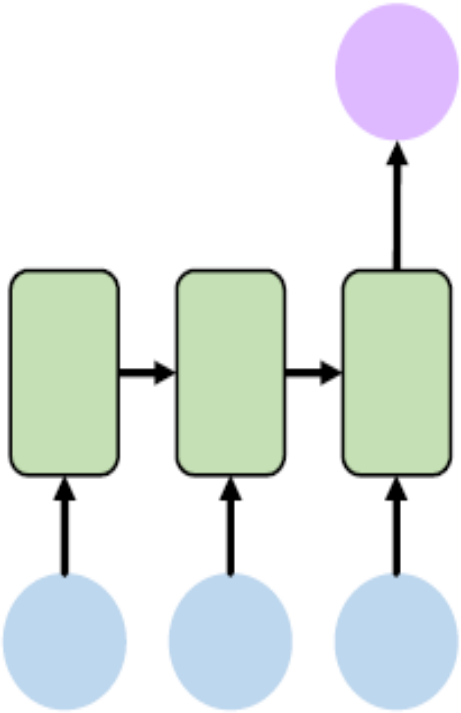
Many to One
Sentiment Classification

Many to Many
Music Generation

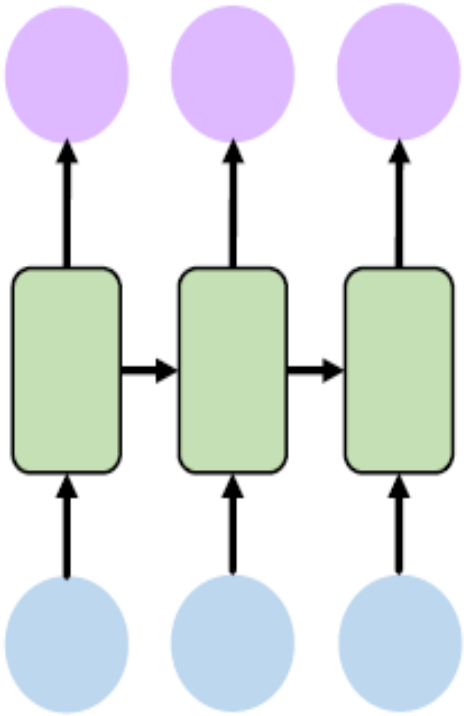
Recurrent neural networks: sequence modeling



One to One
"Vanilla" neural network



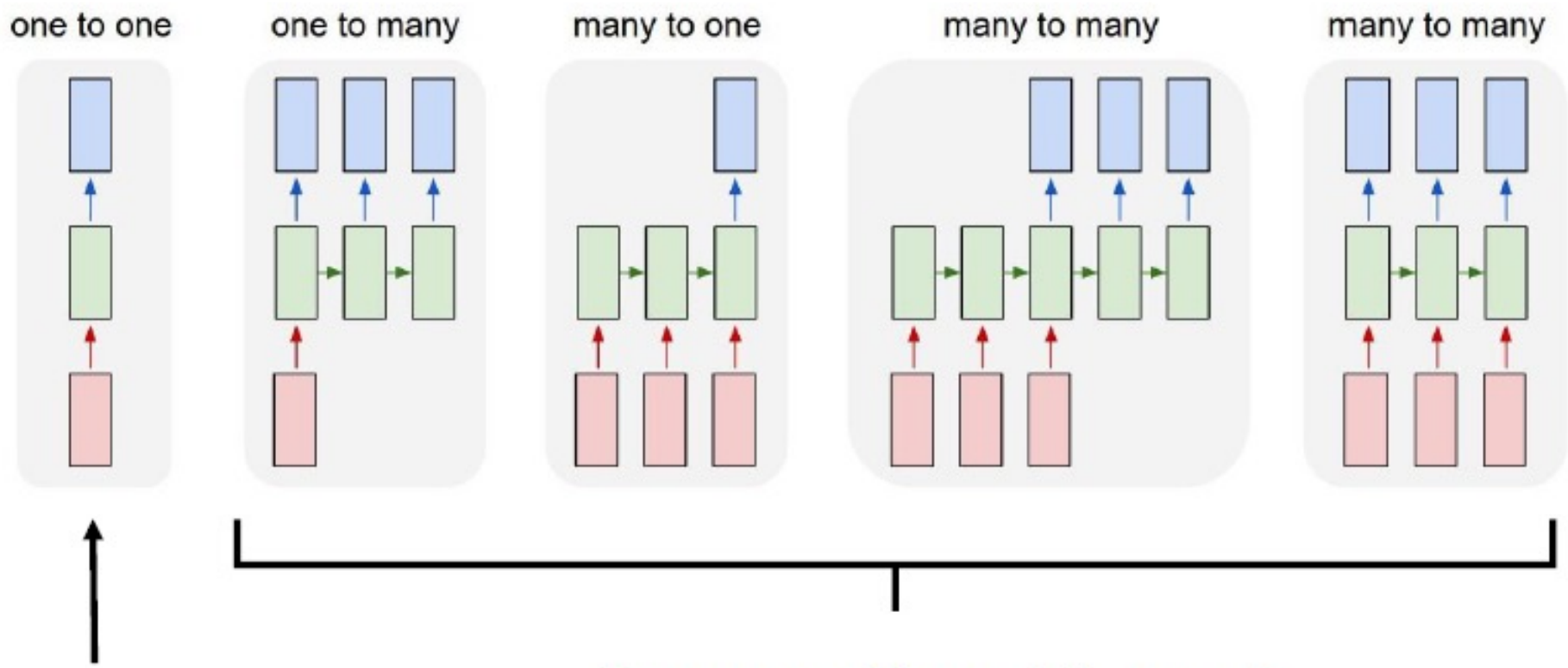
Many to One
Sentiment Classification



Many to Many
Music Generation

… and many other architectures and applications

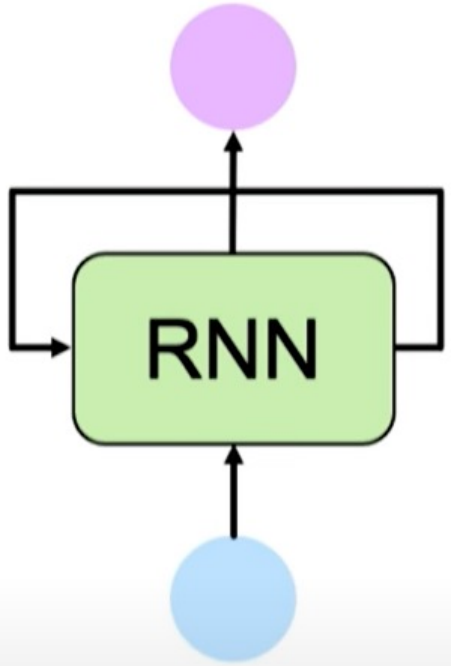
Flavors of Neural Networks



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

Recurrent Neurons and Layers

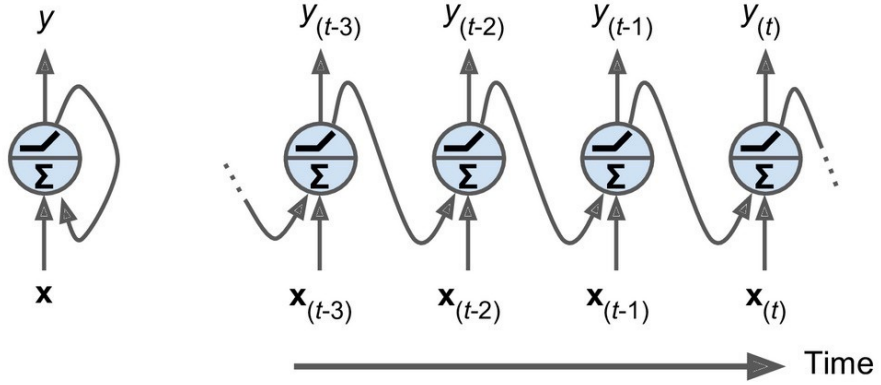


Figure 15-1. A recurrent neuron (left) unrolled through time (right)

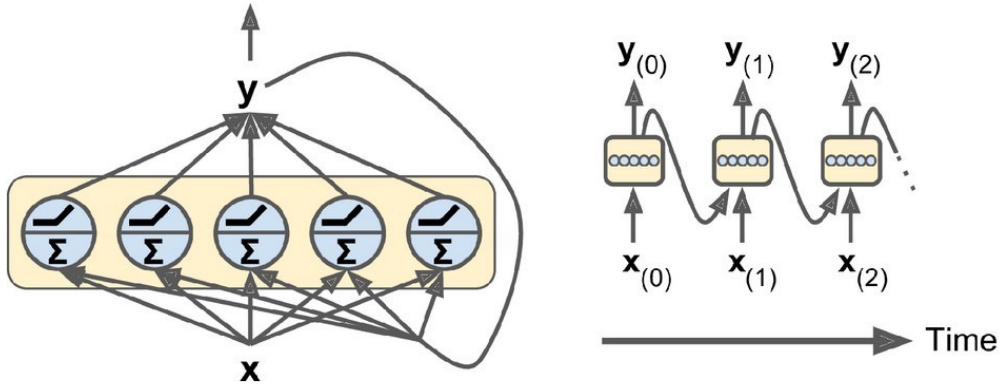


Figure 15-2. A layer of recurrent neurons (left) unrolled through time (right)

- A recurrent neural network looks very much like a feedforward neural network, except it also has connections pointing backward
- A simple RNN is shown which is composed of one neuron receiving inputs, producing an output, and sending that output back to itself, as shown in Figure 15-1 (left)
- At each time step t (also called a frame), this recurrent neuron receives the inputs x(t) as well as its own output from the previous time step, y(t–1)
- Representing network against the time axis is called unrolling the network through time (it's the same recurrent neuron represented once per time step)

Memory Cells

- Since the output of a recurrent neuron at time step t is a function of all the inputs from previous time steps, you could say it has a form of memory.
- A part of a neural network that preserves some state across time steps is called a memory cell (or simply a cell).
- A single recurrent neuron, or a layer of recurrent neurons, is a very basic cell, capable of learning only short patterns (typically about 10 steps long, but this varies depending on the task).
- In general, a cell's state at time step t, denoted h(t) (the "h" stands for "hidden"), is a function of some inputs at that time step and its state at the previous time step.
- Its output at time step t, denoted y(t), is also a function of the previous state and the current inputs.

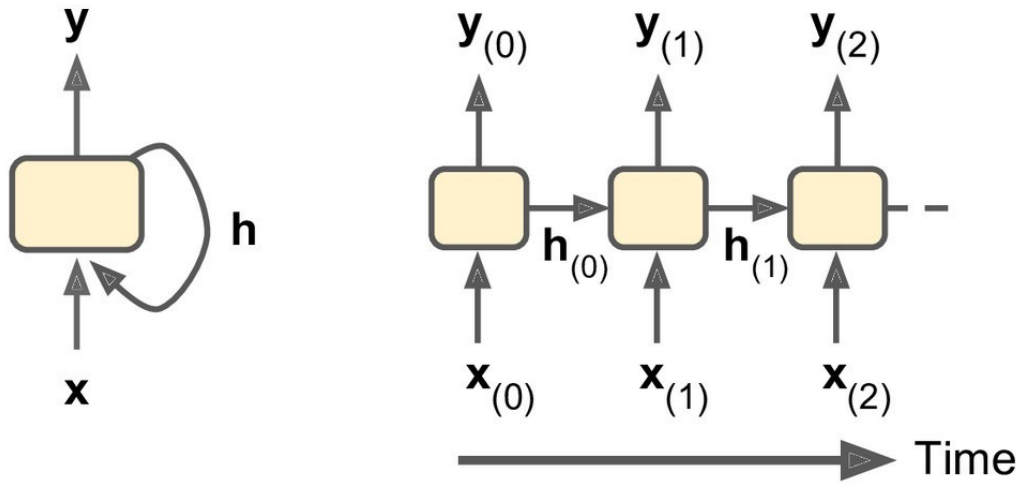
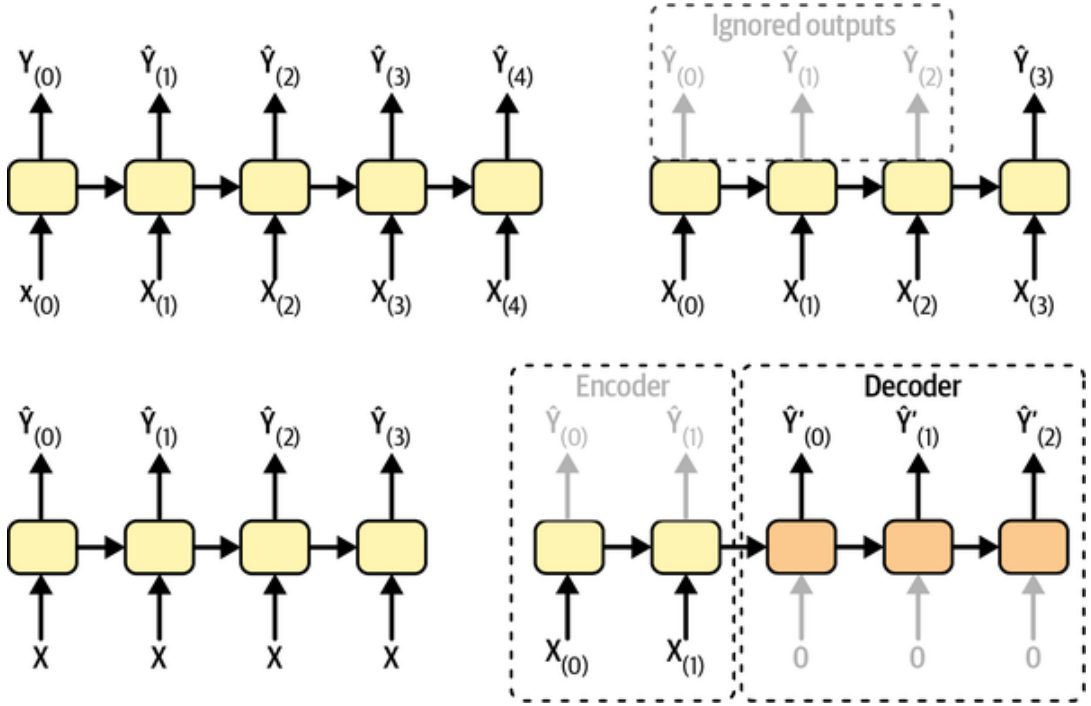


Figure 15-3. A cell's hidden state and its output may be different

Input and Output Sequences

- **Sequence-to-sequence** network - an RNN can simultaneously take a sequence of inputs and produce a sequence of outputs (see the top-left network in Figure 15-4).
  - This type of RNNs is useful for predicting time series such as stock prices: you feed it the prices over the last N days, and it must output the prices shifted by one day into the future (i.e., from N – 1 days ago to tomorrow).
- **Sequence-to-vector** network - feed the network a sequence of inputs and ignore all outputs except for the last one (see the top-right network in Figure 15-4)
  - For example, you could feed the network a sequence of words corresponding to a movie review, and the network would output a sentiment score (e.g., from –1 [hate] to +1 [love]).
- **vector-to-sequence** network - feed the network the same input vector over and over again at each time step and let it output a sequence (see the bottom-left network of Figure 15-4)
  - For example, the input could be an image (or the output of a CNN), and the output could be a caption for that image.



- **Sequence-to-vector** network, called an encoder, followed by a vector-to-sequence network, called a decoder (see the bottom-right network of Figure 15-4). For example, this could be used for translating a sentence from one language to another.

THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

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

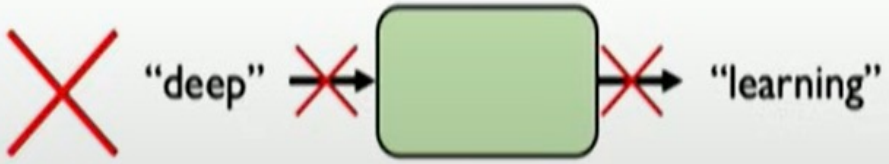
A Sequence Modeling Problem: Predict the Next Word

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

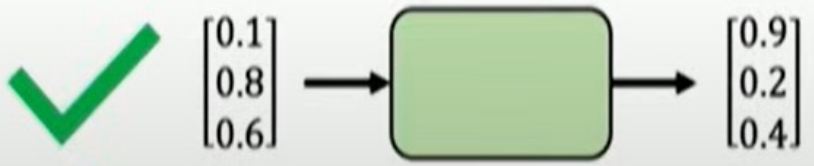
given these words

predict the next word

Representing Language to a Neural Network

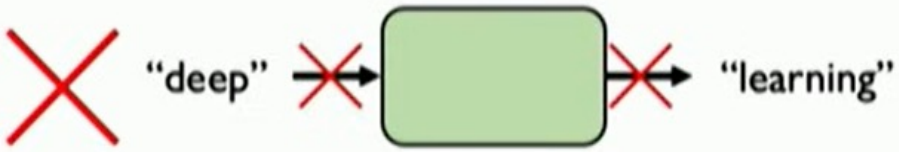


Neural networks cannot interpret words

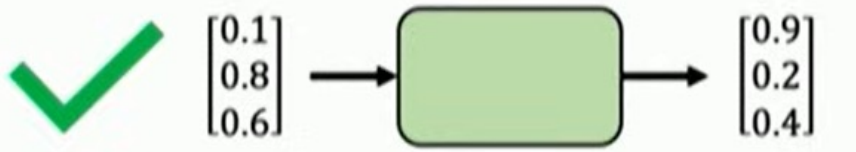


Neural networks require numerical inputs

Encoding Language for a Neural Network

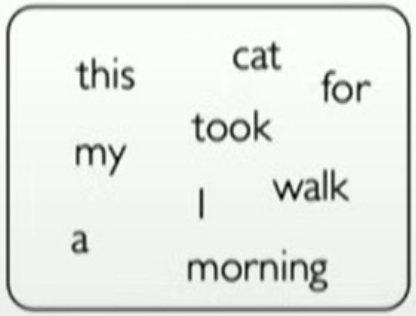


Neural networks cannot interpret words

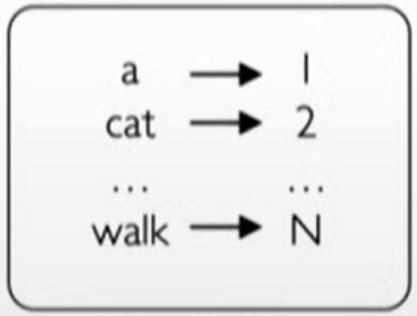


Neural networks require numerical inputs

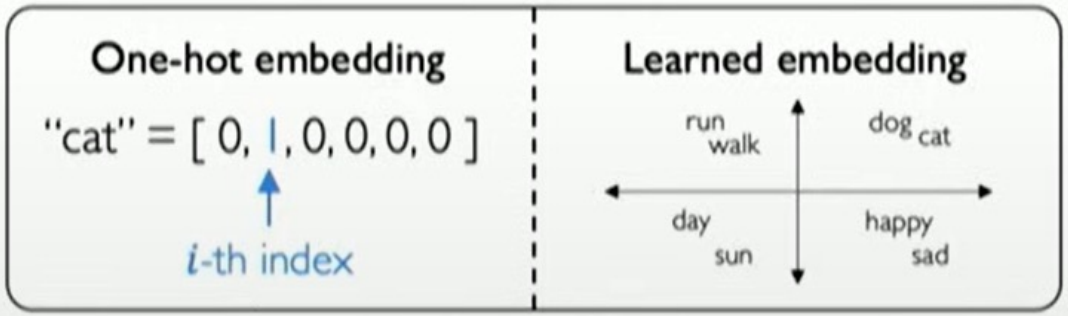
Embedding: transform indexes into a vector of fixed size.



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

vs.

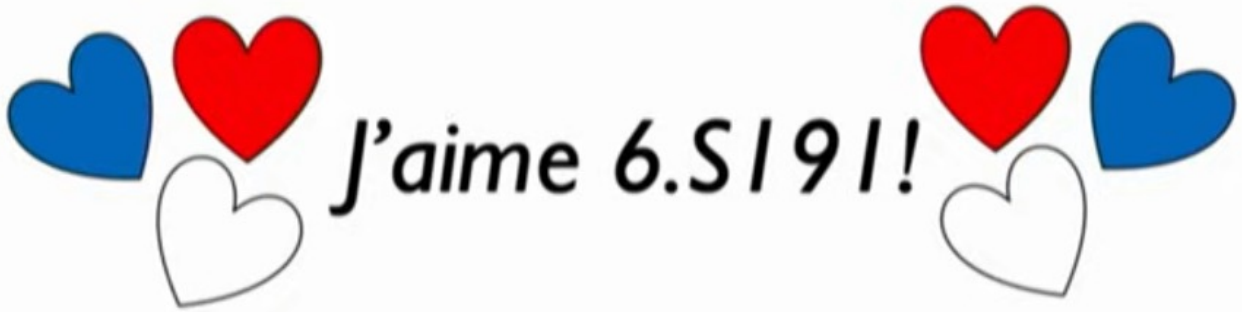
We visited a restaurant for lunch

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

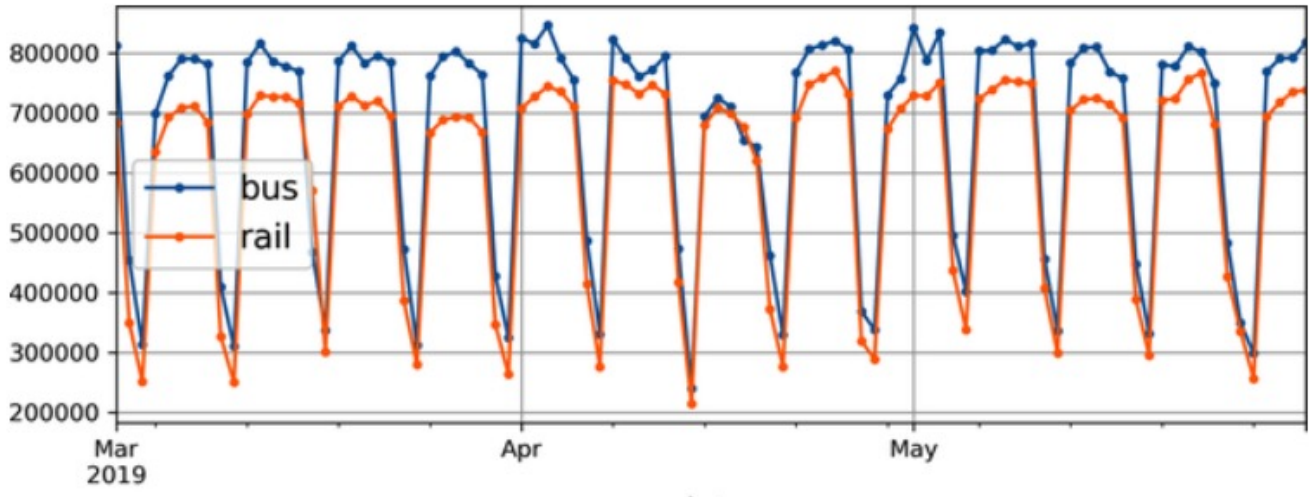
The food was good, not bad at all.

vs.

The food was bad, not good at all.

Forecasting a Time Series

- This is a time series: data with values at different time steps, usually at regular intervals.
- More specifically, since there are multiple values per time step, this is called a multivariate time series.
- If we only looked at the bus column, it would be a univariate time series, with a single value per time step.
- predicting future values (i.e., forecasting) is the most typical task when dealing with time series.
- Other tasks include imputation (filling in missing past values), classification, anomaly detection, and more.



- Looking at the Figure, we can see that a similar pattern is clearly repeated every week.
- This is called a weekly seasonality. In fact, it's so strong in this case that forecasting tomorrow's ridership by just copying the values from a week earlier will yield reasonably good results.
- This is called naive forecasting: simply copying a past value to make our forecast.
- Naive forecasting is often a great baseline, and it can even be tricky to beat in some cases.

Forecasting Using a Simple RNN

Problems with this model:

- The model only has a single recurrent neuron, so the only data it can use to make a prediction at each time step is the input value at
  the current time step and the output value from the previous time step.
- In other words, the RNN's memory is extremely limited: it's just a single number, its previous output.

- The time series contains values from 0 to about 1.4, but since the default activation function is tanh, the recurrent layer can only output values between –1 and +1. There's no way it can predict values between 1.0 and 1.4.

```
model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(1, input_shape=[None, 1])
])
```

Now create a model with a larger recurrent layer, containing 32 recurrent neurons, and add a dense output layer on top of it with a single output neuron and no activation function. The recurrent layer will be able to carry much more information from one time step to the next, and the dense output layer will project the final output from 32 dimensions down to 1, without any value range constraints:

```
univar_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, input_shape=[None, 1]),
    tf.keras.layers.Dense(1)  # no activation function by default
])
```

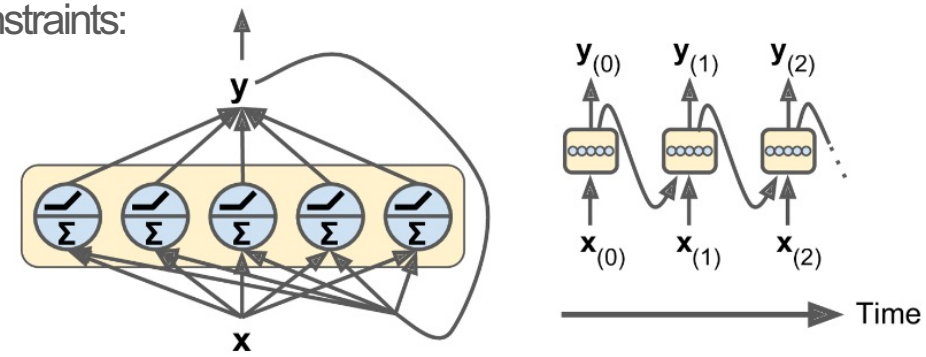


Figure 4-2. A layer of recurrent neurons (left), unrolled through time (right)

Hands-On Machine Learning

Forecasting Using Deep RNNs

- It is quite common to stack multiple layers of cells
- To implement a deep RNN - just stack recurrent layers
- In this example, we use three SimpleRNN layers
  - Could add any other type of recurrent layer, such as an LSTM layer or a Gated Recurrent Unit (GRU) layer

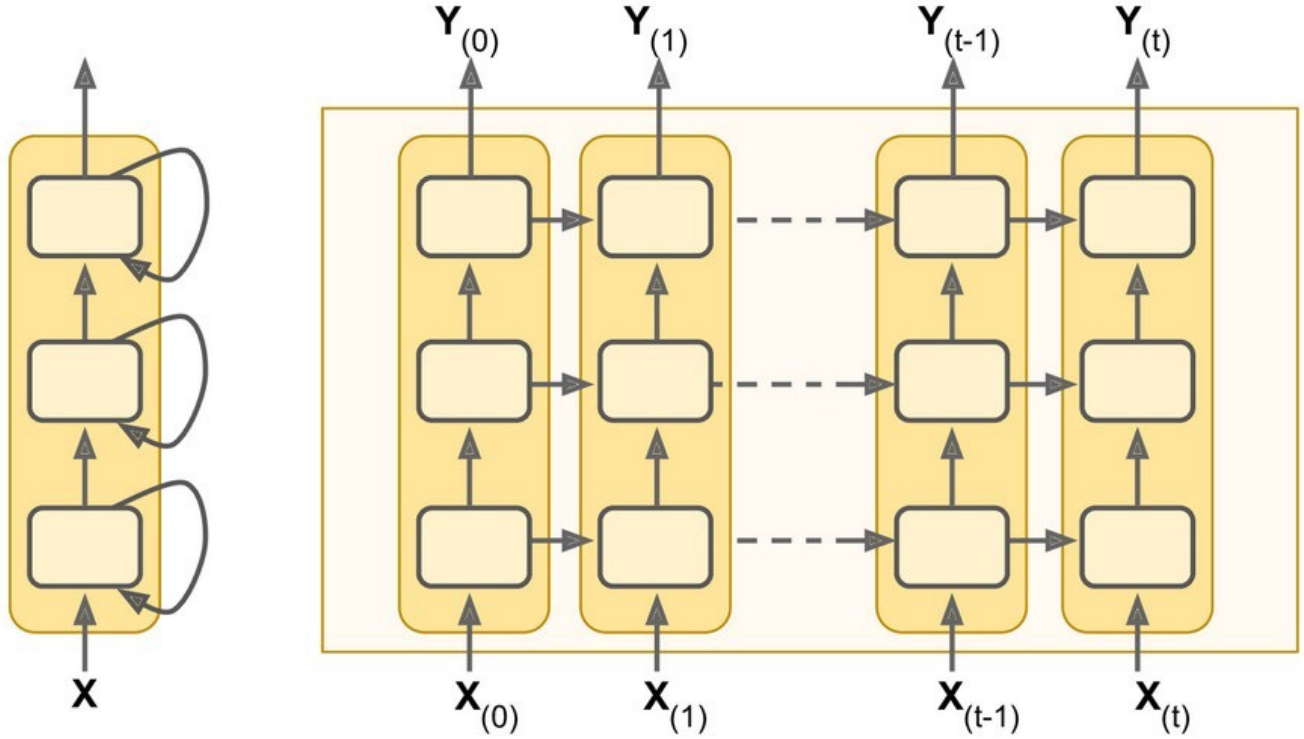


Figure 15-7. Deep RNN (left) unrolled through time (right)

```
deep_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, return_sequences=True, input_shape=
    tf.keras.layers.SimpleRNN(32, return_sequences=True),
    tf.keras.layers.SimpleRNN(32),
    tf.keras.layers.Dense(1)
])
```

Forecasting Multivariate Time Series

Univariate Time Series:

1. In univariate time series data, there is only one variable or feature being observed over time.
2. For example, if you're tracking the temperature of a city over the course of a year, and you only record the temperature at different time points, then you have a univariate time series.

Multivariate Time Series:

1. In multivariate time series data, there are multiple variables or features being observed over time, and these variables can influence each other.
2. For example, in addition to temperature, you might also track humidity, wind speed, and precipitation over the same period of time. Each of these variables constitutes a different dimension or feature in the dataset.
3. Multivariate time series data can be more complex to analyze because the relationships between different variables need to be taken into account. However, they can provide richer insights and allow for more comprehensive modeling of complex systems.

Forecasting Several Timesteps Ahead

Hands-On Machine Learning

Option 1:

```
series = generate_time_series(1, n_steps + 10)
X_new, Y_new = series[:, :n_steps], series[:, n_steps:]
X = X_new
for step_ahead in range(10):
    y_pred_one = model.predict(X[:, step_ahead:])[:, np.newaxis, :]
    X = np.concatenate([X, y_pred_one], axis=1)

Y_pred = X[:, n_steps:]
```

Option 2:

```
series = generate_time_series(10000, n_steps + 10)
X_train, Y_train = series[:7000, :n_steps], series[:7000, -10:, 0]
X_valid, Y_valid = series[7000:9000, :n_steps], series[7000:9000, -1
X_test, Y_test = series[9000:, :n_steps], series[9000:, -10:, 0]
```

Now we just need the output layer to have 10 units instead of 1:

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[N
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(10)
])
```

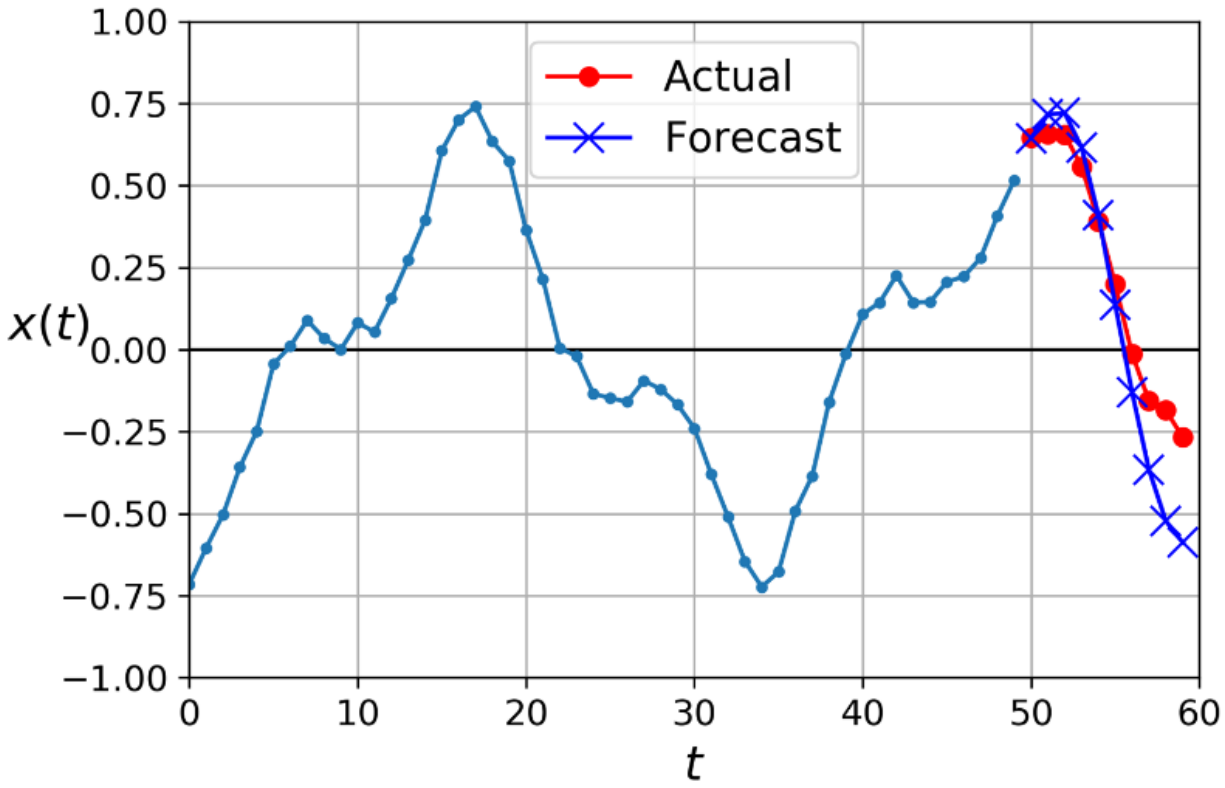


Figure 15-8. Forecasting 10 steps ahead, 1 step at a time

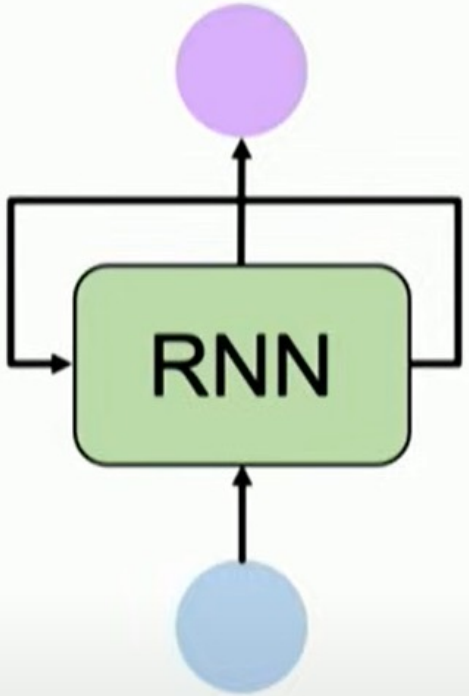
Handling Long Sequences

- To train an RNN on long sequences, we must run it over many time steps, making the unrolled RNN a very deep network.
- When an RNN processes a long sequence, it will gradually forget the first inputs in the sequence.
- Due to the transformations that the data goes through when traversing an RNN, some information is lost at each time step.
  - After a while, the RNN's state contains virtually no trace of the first inputs.

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

THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

Backpropagation Through Time (BPTT)

MIT 6.S191 – Deep Sequence Modeling, Introtodeeplearning.com

Professor Hamza F. Alsarhan
SEAS 8505

Training RNNs

$$\mathscr{L}(\mathbf{Y}_{(2)}, \mathbf{Y}_{(3)}, \mathbf{Y}_{(4)}; \hat{\mathbf{Y}}_{(2)}, \hat{\mathbf{Y}}_{(3)}, \hat{\mathbf{Y}}_{(4)})$$

- Unroll it through time and then simply use regular backpropagation
  - This strategy is called backpropagation through time (BPTT)
- Just like in regular backpropagation, there is a first forward pass through the unrolled network (represented by the dashed arrows).
- Then the output sequence is evaluated using a cost function C(Y(0), Y(1), …Y(T)) (where T is the max time step).
  - Note that this cost function may ignore some outputs, as shown in Figure 15-5 (for example, in a sequence-to-vector RNN, all outputs are ignored except for the very last one).
- The cost function gradients are propagated backward through the unrolled network
- Finally the model parameters are updated using the gradients computed during BPTT.
  - Note that the gradients flow backward through all the outputs used by the cost function, not just through the final output

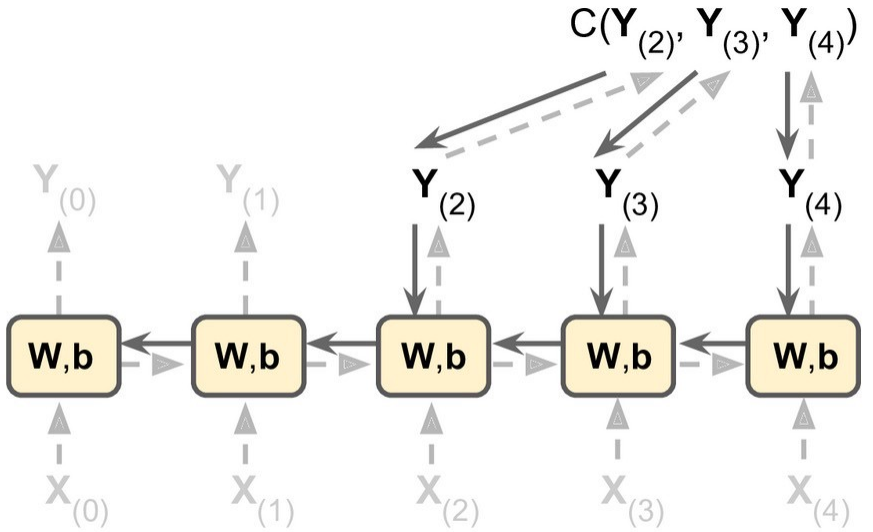
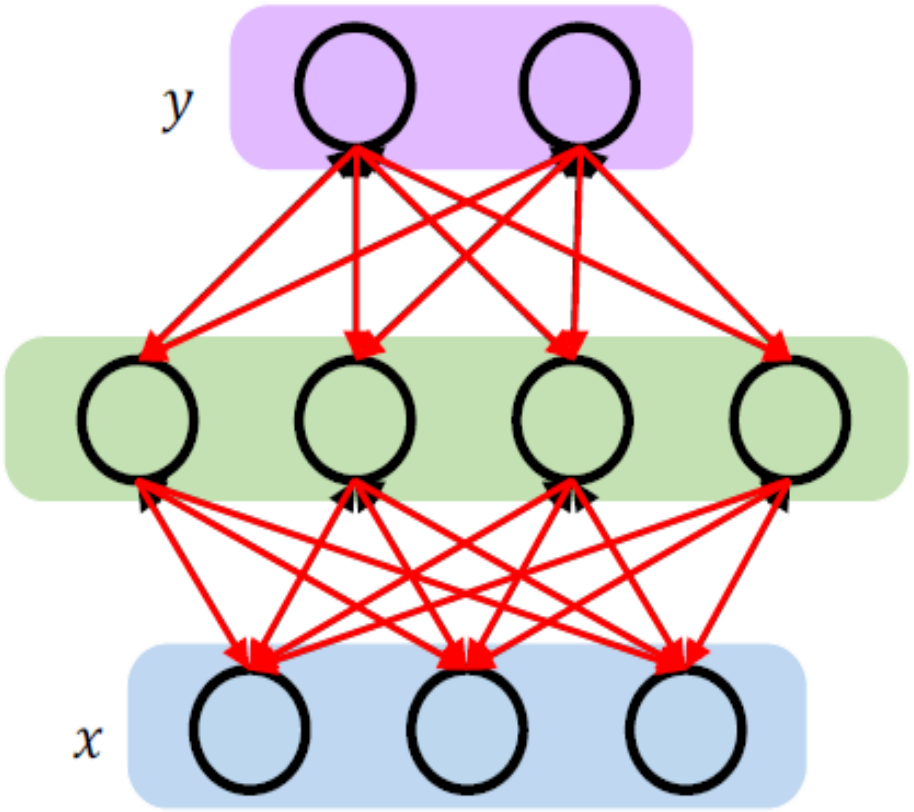
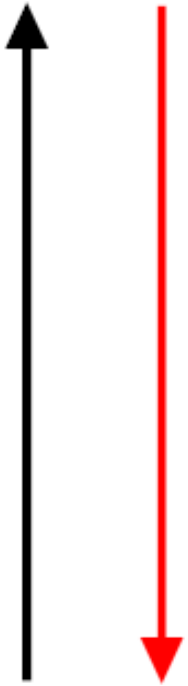


Figure 15-5. Backpropagation through time

Recall: backpropagation in feed forward models

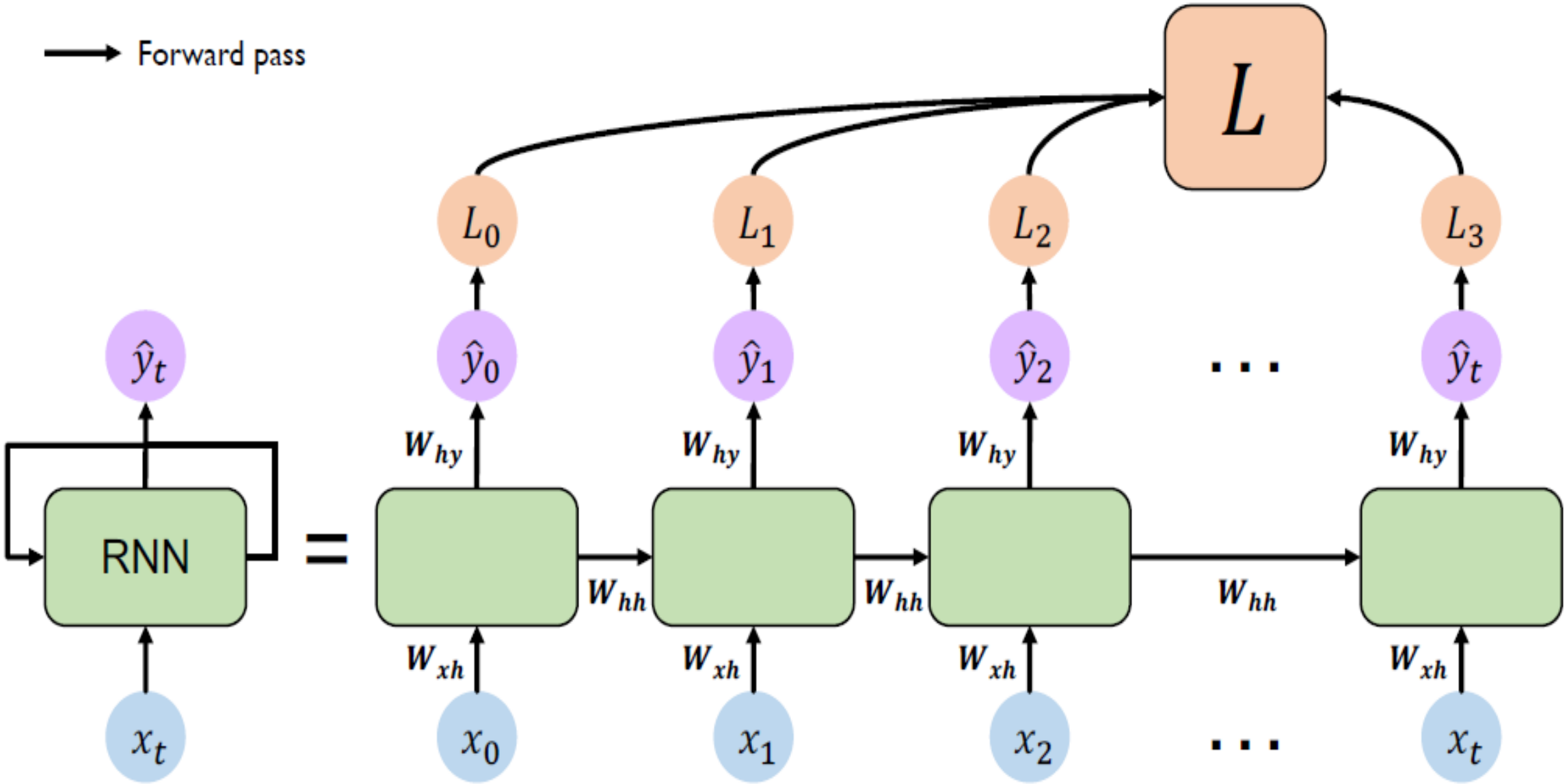




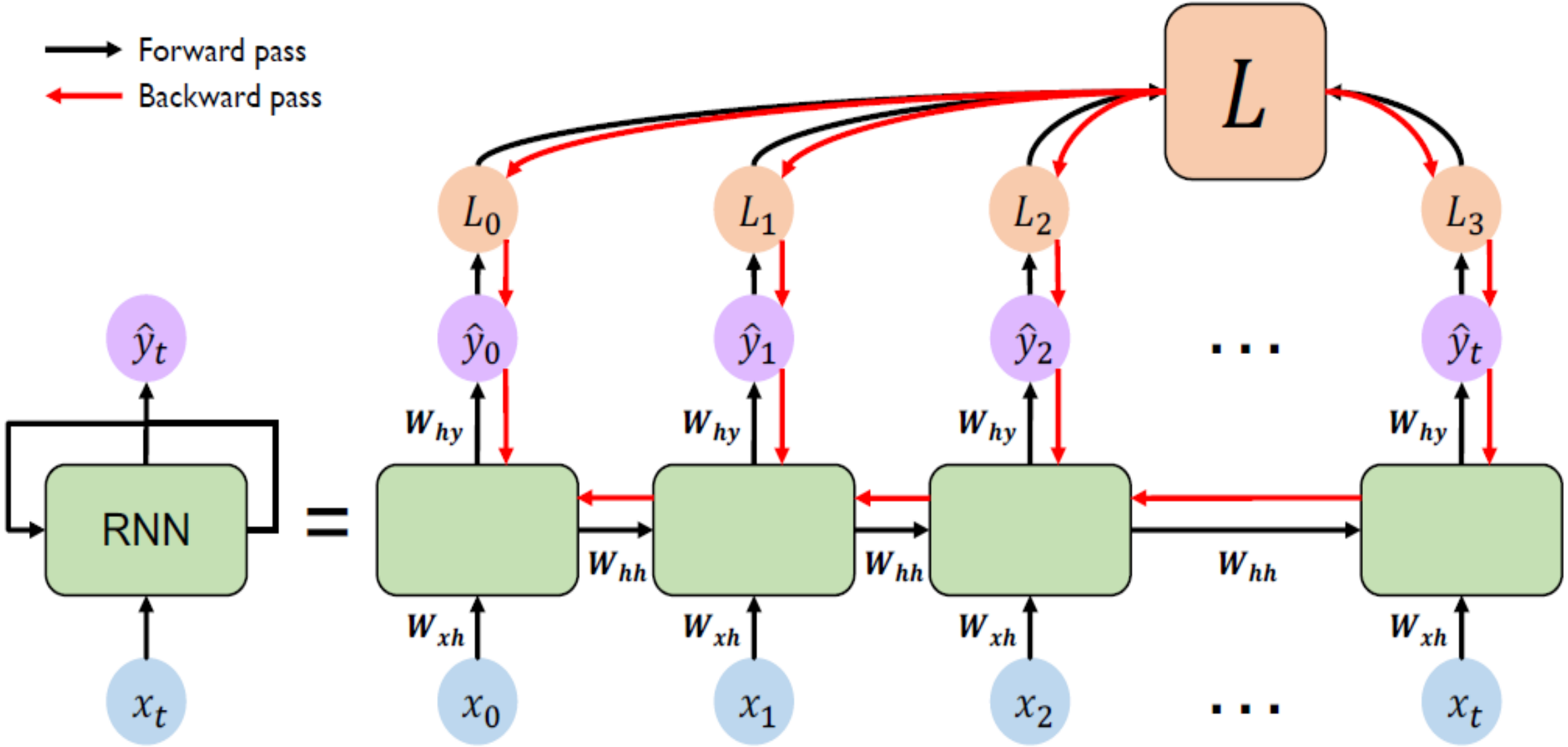
Backpropagation algorithm:

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

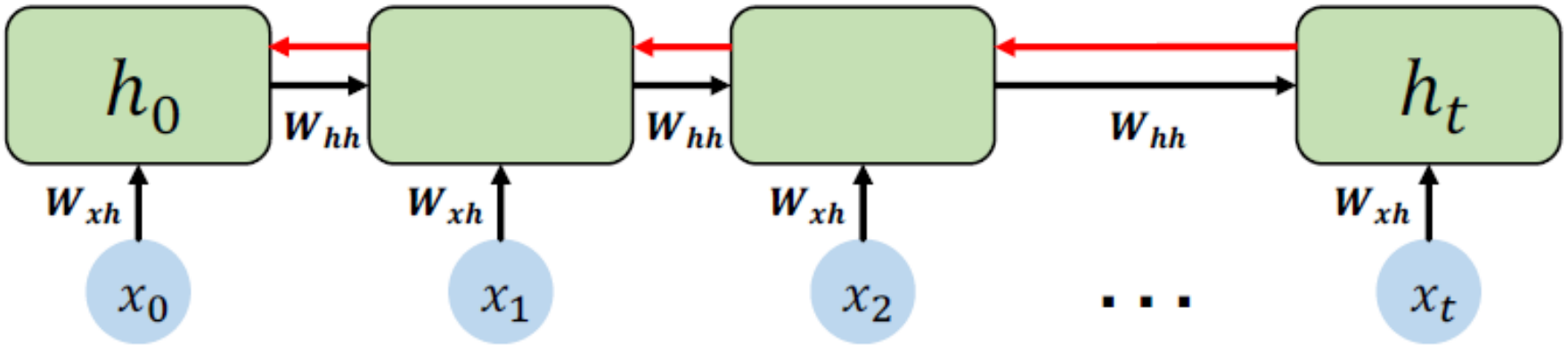
RNNs: backpropagation through time



RNNs: backpropagation through time



Standard RNN gradient flow



Vanishing & Unstable Gradients

Vanishing Gradients Issue:

1. This occurs when gradients become extremely small as they propagate backward through many layers during training.
2. It primarily affects deep neural networks with many layers, particularly those using activation functions like sigmoid or tanh, which have gradients that diminish as absolute values increase.
3. Vanishing gradients can prevent lower layers from learning effectively, leading to slow or stalled training and poor performance, especially in deep architectures like recurrent neural networks (RNNs) and deep feedforward neural networks.

Unstable Gradients Issue:

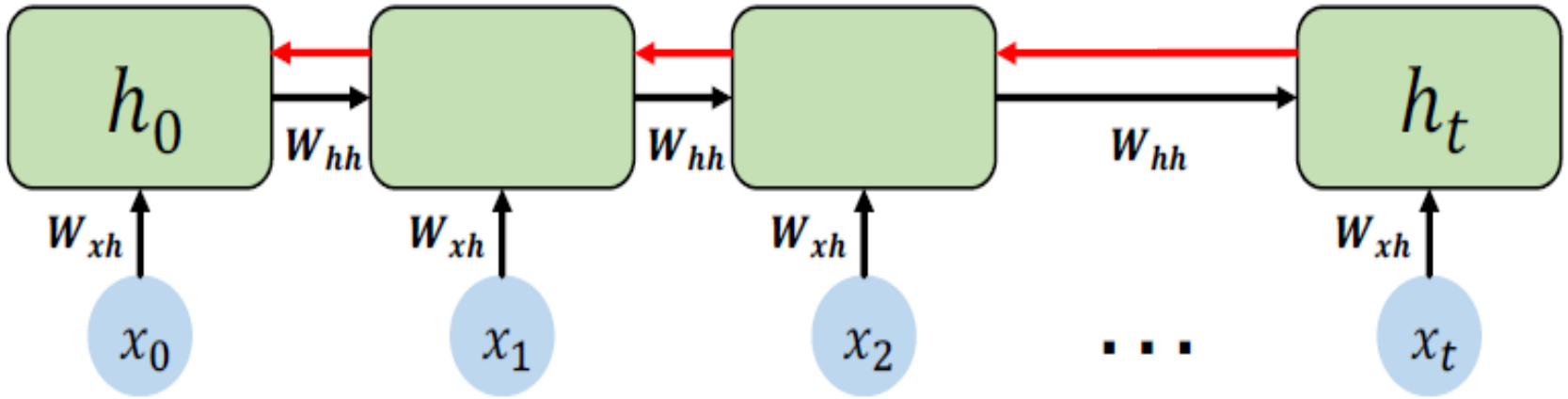
1. This refers to the situation where gradients during training become erratic or highly variable, leading to difficulties in convergence and unstable training behavior.
2. Unstable gradients can occur due to various factors, including poor parameter initialization, the choice of optimization algorithm, or the architecture of the neural network itself.
3. While vanishing gradients contribute to the unstable gradients problem, it's not the only cause. Gradient explosion, where gradients become extremely large, can also contribute to instability during training.

Fighting Unstable Gradients

Many strategies employed in deep neural networks to mitigate the unstable gradients problem are also applicable to RNNs:

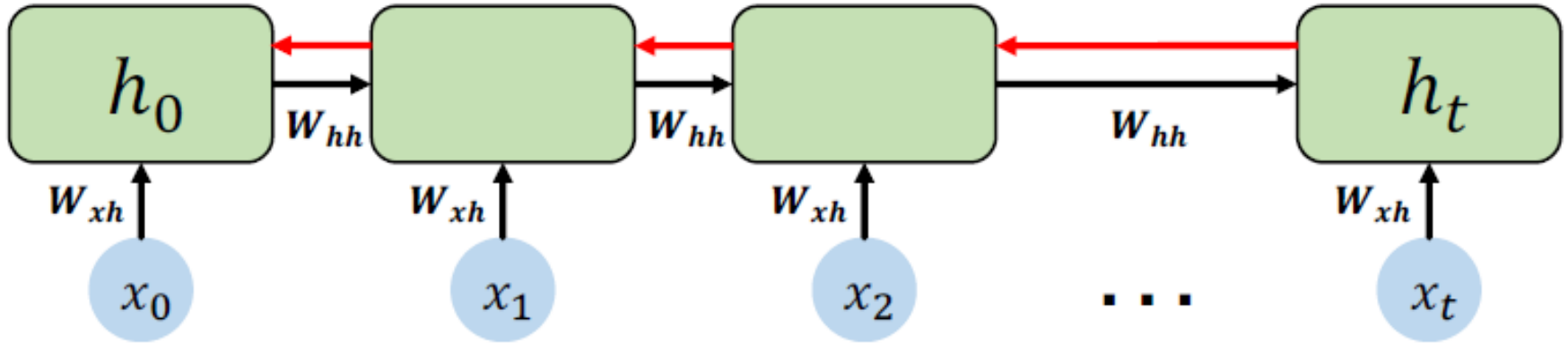
- Proper parameter initialization, faster optimization algorithms, dropout regularization, among others.
- However, non-saturating activation functions like ReLU may not be as effective in this context and could potentially exacerbate instability during training.
- To mitigate this risk, one can opt for a smaller learning rate or utilize a saturating activation function such as hyperbolic tangent (tanh), which is often the default choice.
- Similarly, the gradients themselves may also experience explosion. If training instability is observed, it may be beneficial to monitor the size of gradients.

Standard RNN gradient flow



Computing the gradient wrt h_0 involves **many factors of W_{hh}** (and repeated f'!)

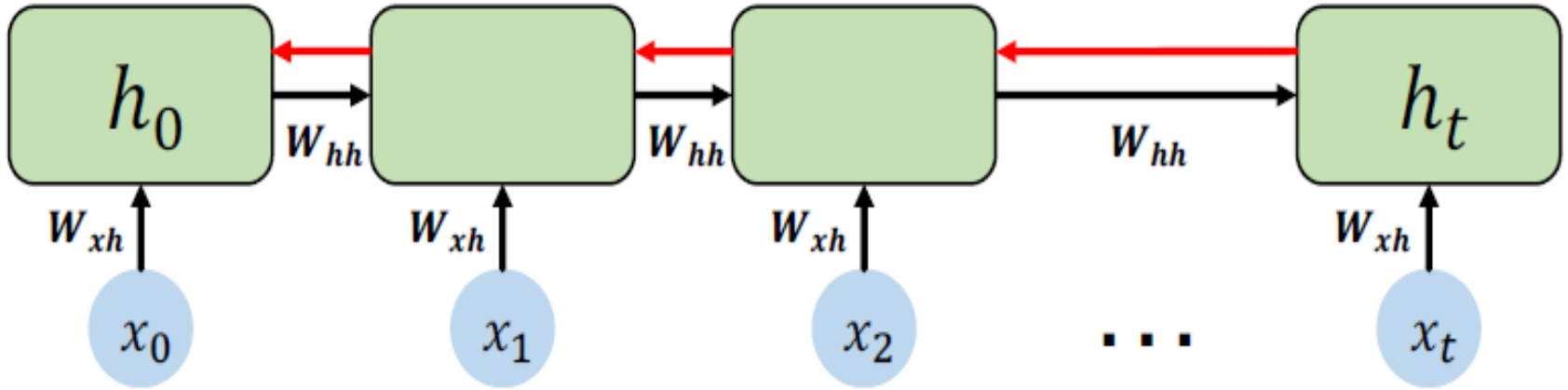
Standard RNN gradient flow: exploding gradients



Computing the gradient wrt h_0 involves **many factors of W_{hh}** (and repeated f'!)

Many values > 1:
exploding gradients

Standard RNN gradient flow: exploding gradients

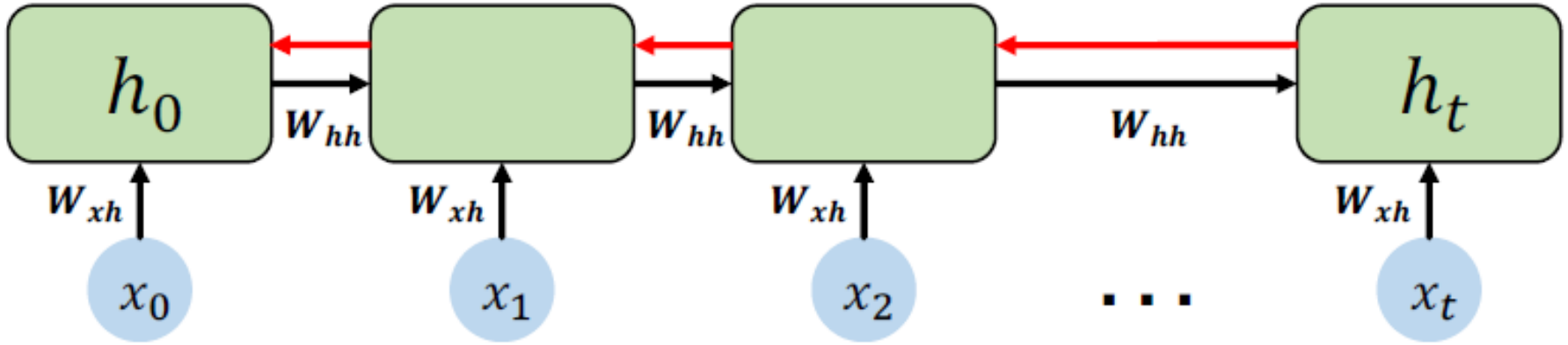


Computing the gradient wrt h_0 involves **many factors of W_{hh}** (and repeated f'!)

Many values > 1:
exploding gradients

Gradient clipping to
scale big gradients

Standard RNN gradient flow: vanishing gradients



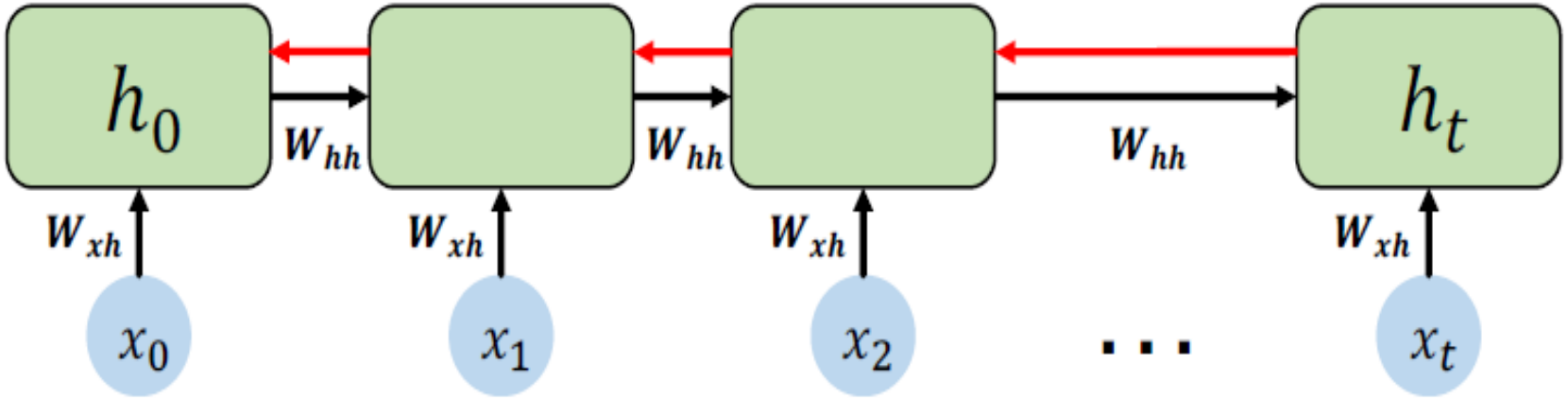
Computing the gradient wrt h_0 involves **many factors of W_{hh}** (and repeated f'!)

Many values > 1:
exploding gradients

Gradient clipping to
scale big gradients

Many values < 1:
vanishing gradients

Standard RNN gradient flow: vanishing gradients



Computing the gradient wrt h_0 involves **many factors of W_{hh}** (and repeated f'!)

Largest singular value > 1:
exploding gradients

Gradient clipping to
scale big gradients

Largest singular value < 1:
vanishing gradients

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

The problem of long-term dependencies

Why are vanishing gradients a problem?

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 network to capture short-term dependencies

The problem of long-term dependencies

Why are vanishing gradients a problem?

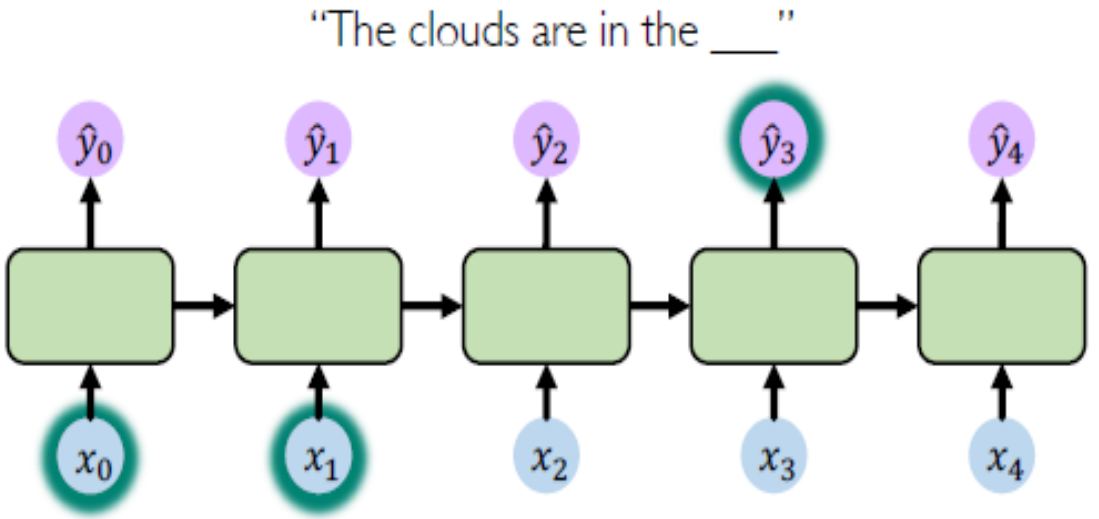
Multiply many **small numbers** together

↓

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

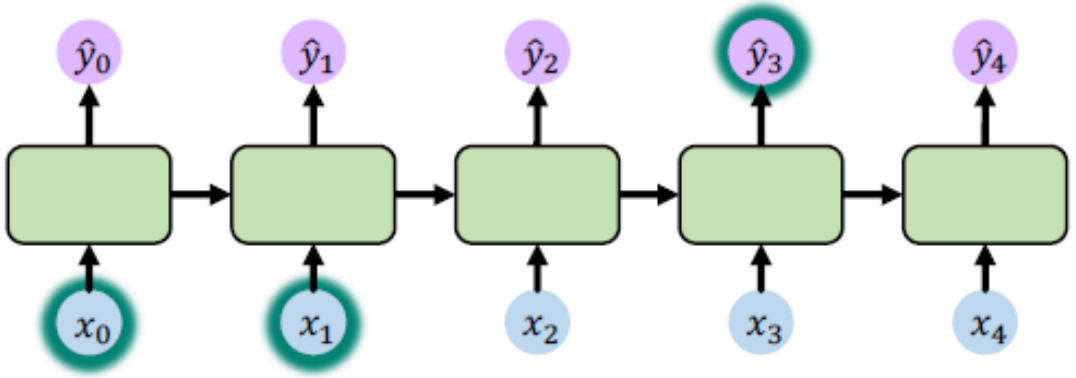
↓

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

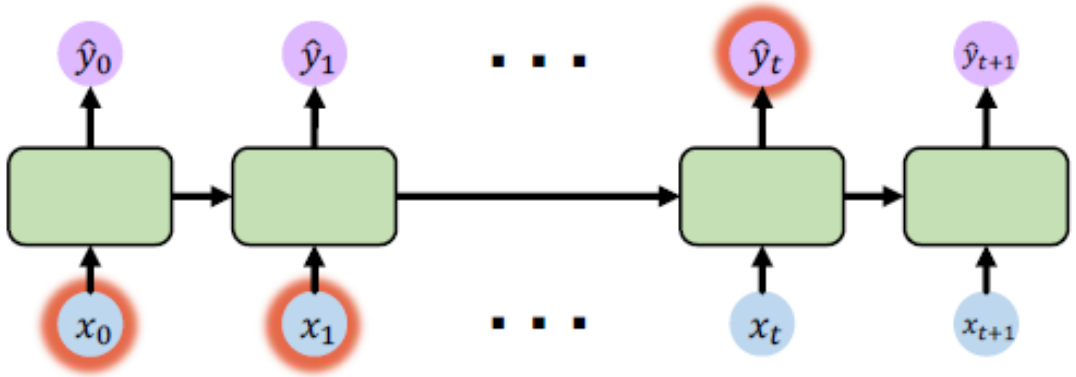
↓

Bias parameters to capture short-term dependencies

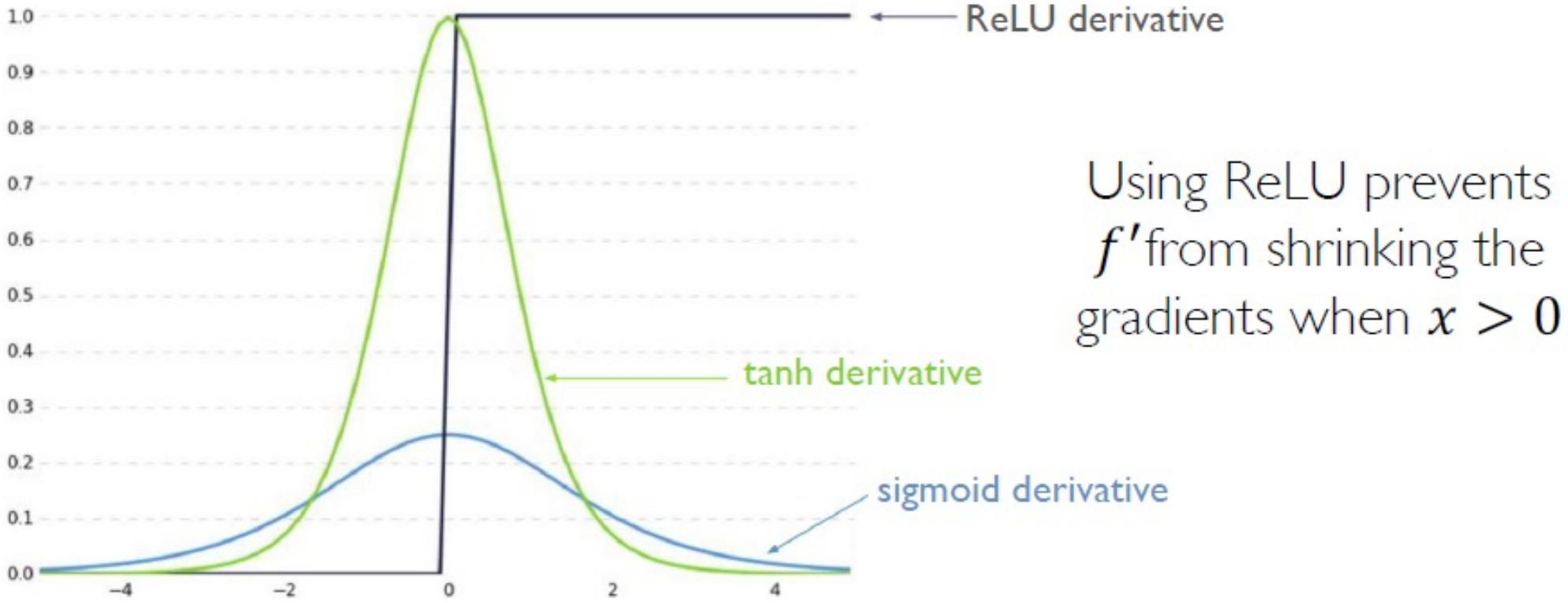
"The clouds are in the ___"



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



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

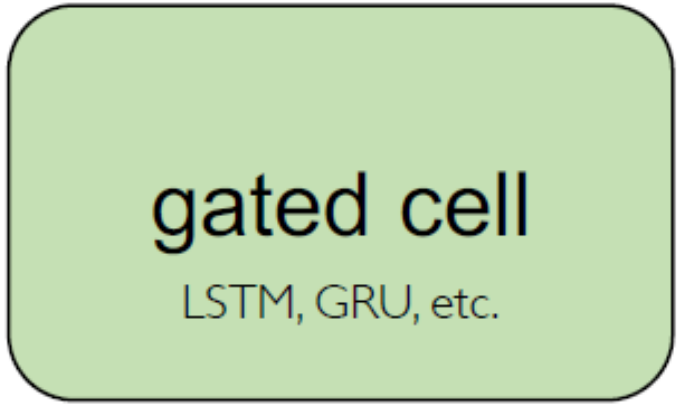
$$I_n = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

This helps prevent the weights from shrinking to zero.

Adapted from H. Suresh, 6.S191 2018

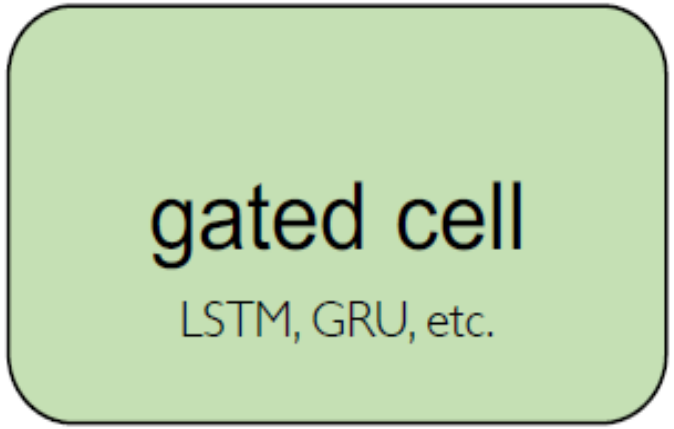
Solution #3: gated cells

Idea: use a more **complex recurrent unit with gates** to control what information is passed through



Solution #3: gated cells

Idea: use a more **complex recurrent unit with gates** to control what information is passed through



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

Adapted from H. Suresh, 6.S191 2018

Tackling the Short-Term Memory Problem

- As data passes through an RNN, it undergoes transformations at each time step, leading to the loss of some information.
- Over time, the RNN's internal state gradually loses track of the initial inputs, which can pose a significant challenge.
- Consider Dory the fish attempting to translate a lengthy sentence:
  - by the time she completes reading it, she struggles to recall how it began.
- To address this issue, specialized cells with long-term memory capabilities have been developed.
- These advanced cells, notably the LSTM cell, have proven highly effective, rendering the basic cells less commonly utilized.
- The Long Short Term Memory cell is the most popular among these long-term memory cells.

Gated RNNs

```
model = tf.keras.Sequential([
    tf.keras.layers.LSTM(32, return_sequences=True, input_shape=[None,
    tf.keras.layers.Dense(14)
])
```

- Most effective recurrent neural network models
- Includes LSTM and Gated Recurrent Units (GRU)
- Can store information within their units for future use
- Units decide what information to store and when to allow reads, writes, and erasures through the concept of gates
- A gated unit stores an input

Based on MLB

THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

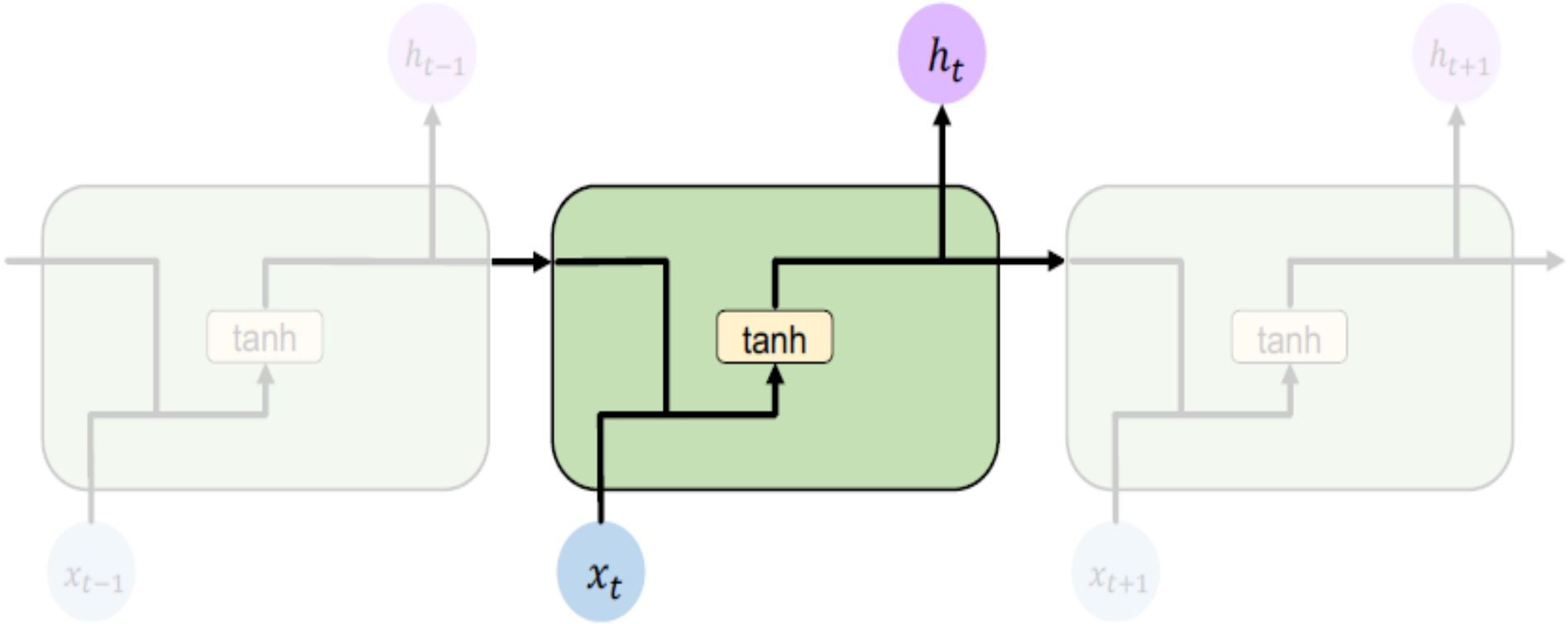
Long Short-Term Memory (LSTM) Networks

MIT 6.S191 – Deep Sequence Modeling, Introtodeeplearning.com

Professor Hamza F. Alsarhan
SEAS 8505

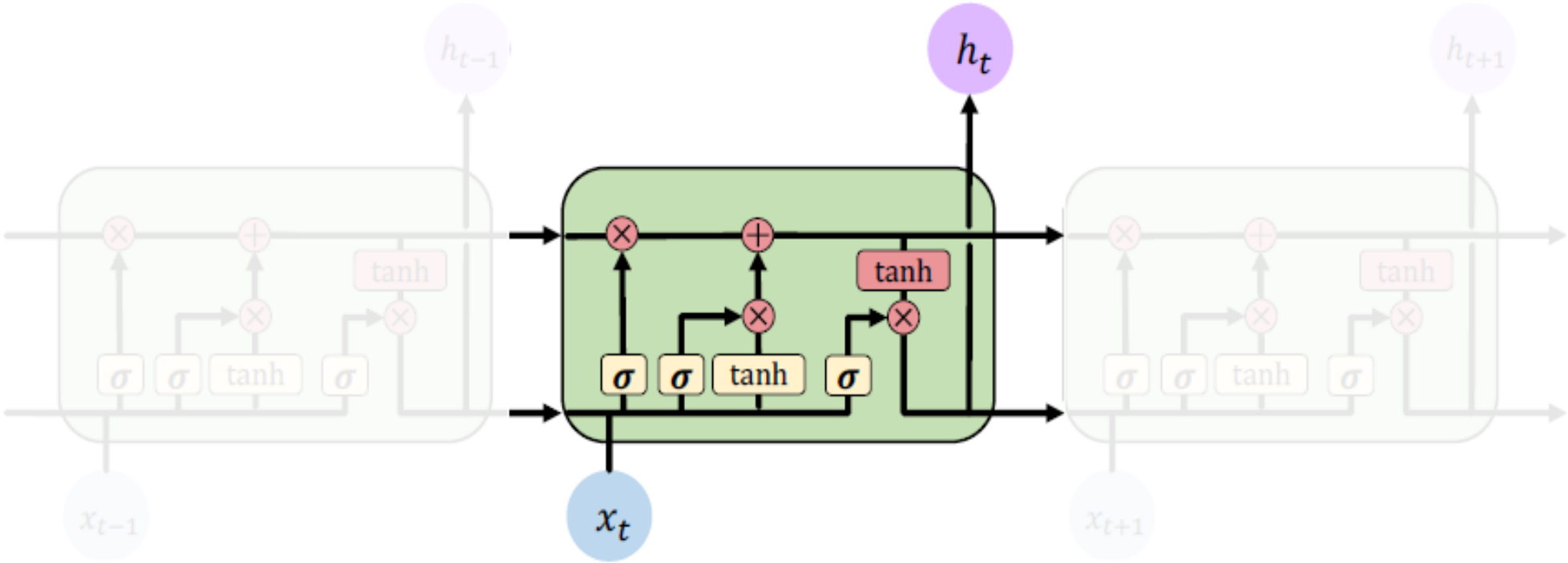
Standard RNN

In a standard RNN, repeating modules contain a **simple computation node**



Long Short Term Memory (LSTMs)

LSTM repeating modules contain **interacting layers** that **control information flow**



LSTM cells are able to track information throughout many timesteps

Long Short Term Memory (LSTMs)

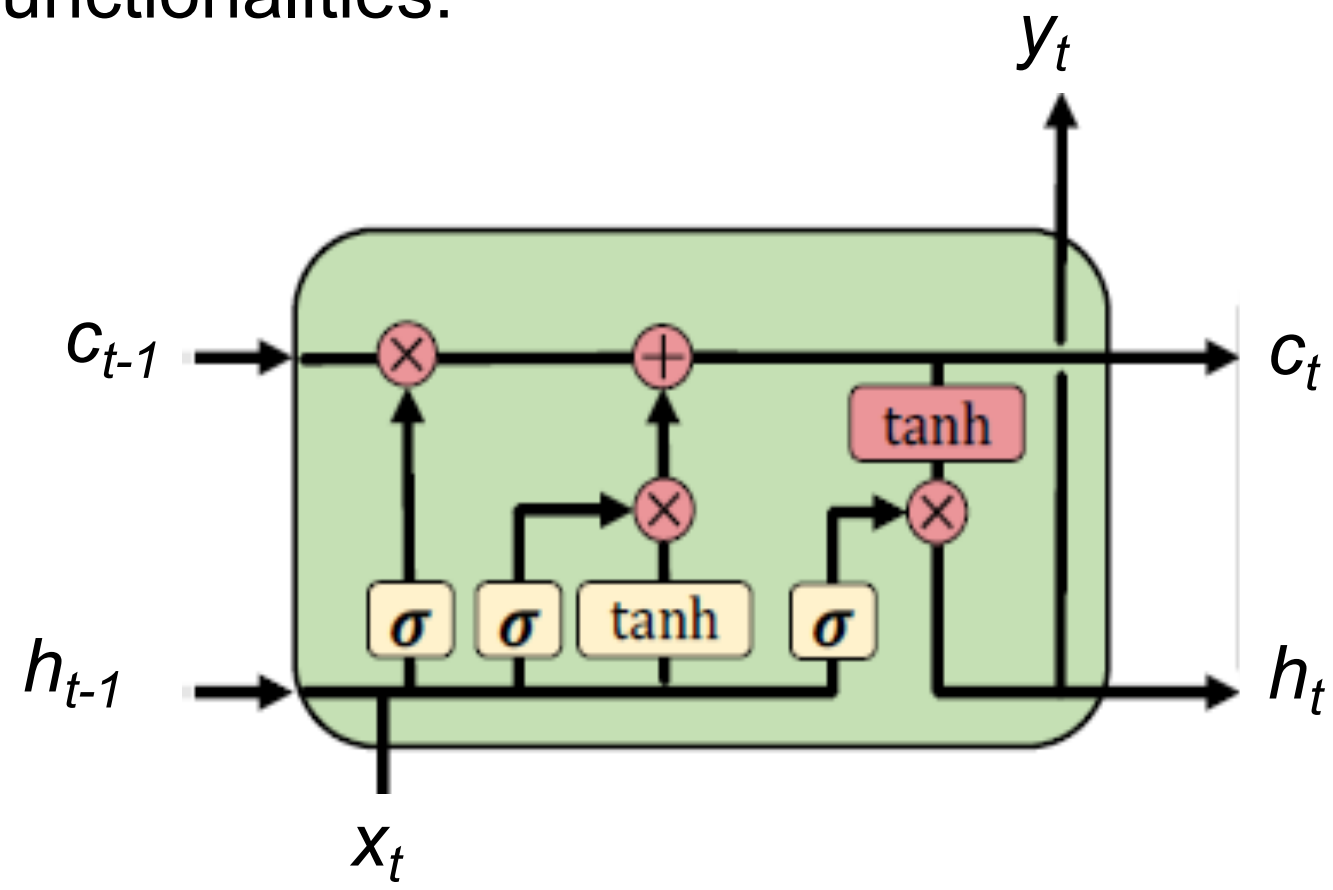
They have 4 functionalities:

1- Forget

2- Store

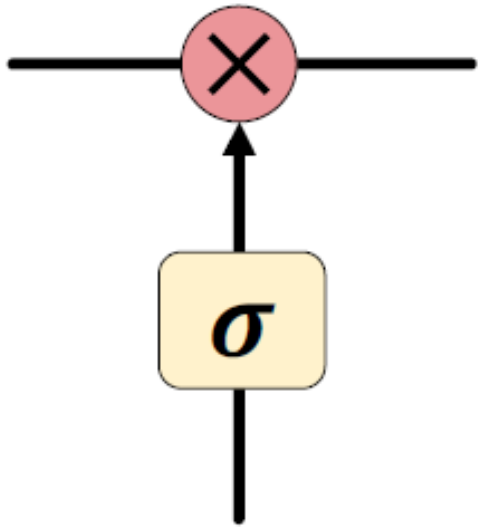
3- Update

4- Output



Long Short Term Memory (LSTMs)

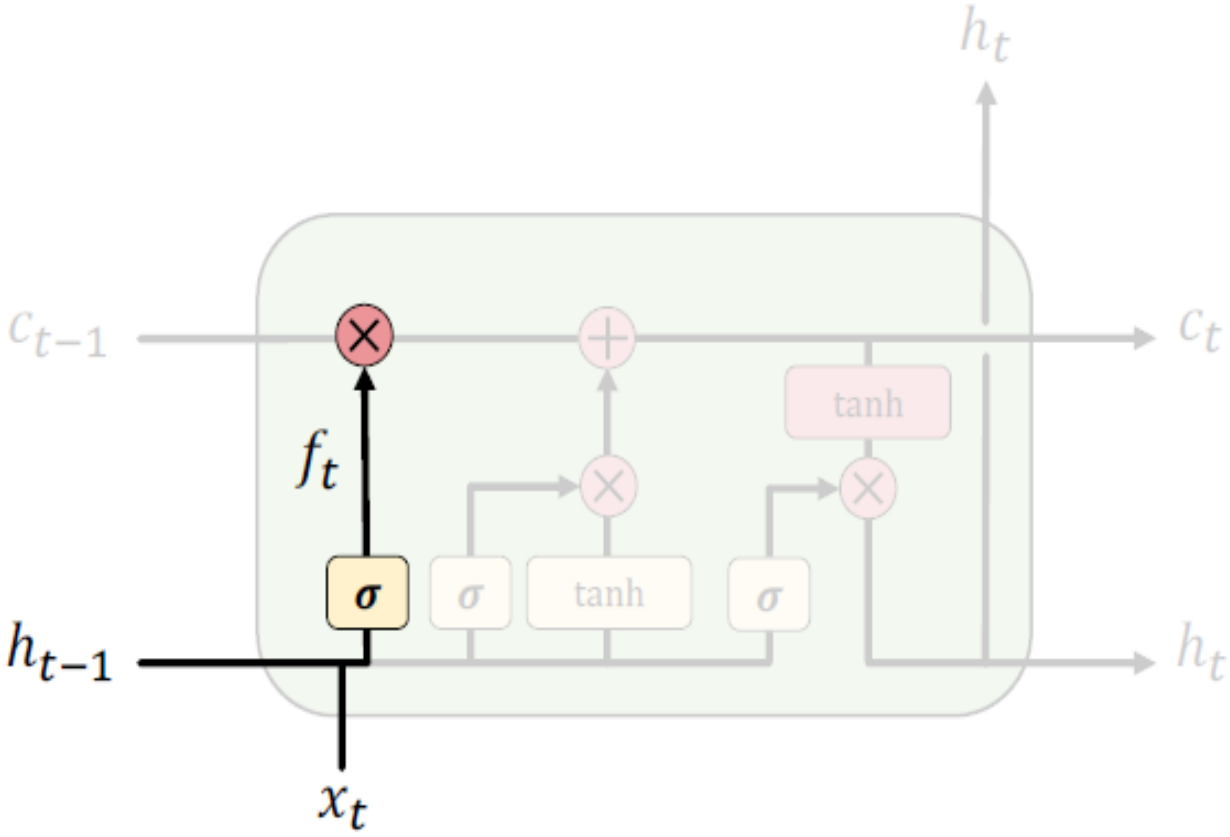
Information is **added** or **removed** to cell state through structures called **gates**



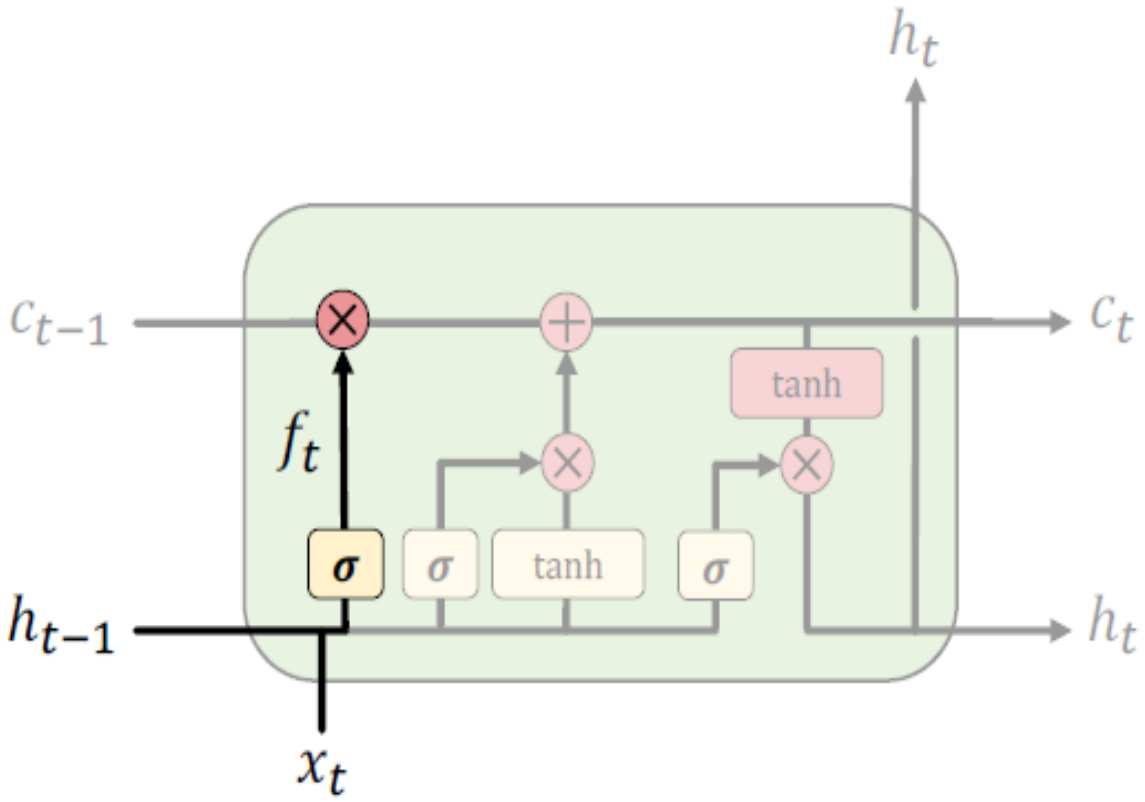
Gates optionally let information through, via a sigmoid neural net layer and pointwise multiplication

Long Short Term Memory (LSTMs)

LSTMs **forget irrelevant** parts of the previous state



LSTMs: forget irrelevant information



$$f_t = \sigma(\boldsymbol{W_i}[\, h_{t-1}, x_t \,] + b_f)$$

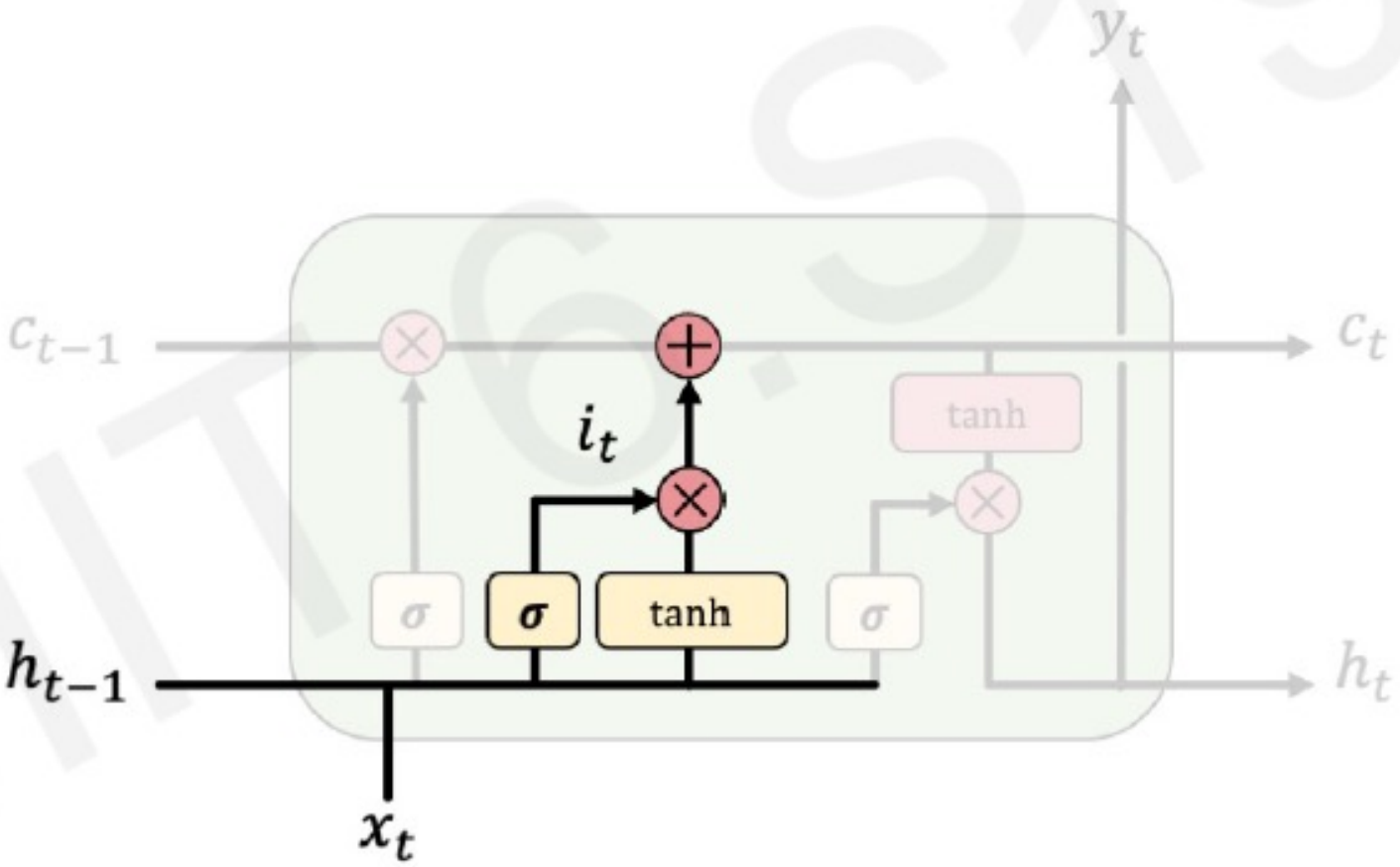
- Use previous cell output and input
- Sigmoid: value 0 and 1 – "completely forget" vs. "completely keep"

ex: Forget the gender pronoun of previous subject in sentence.

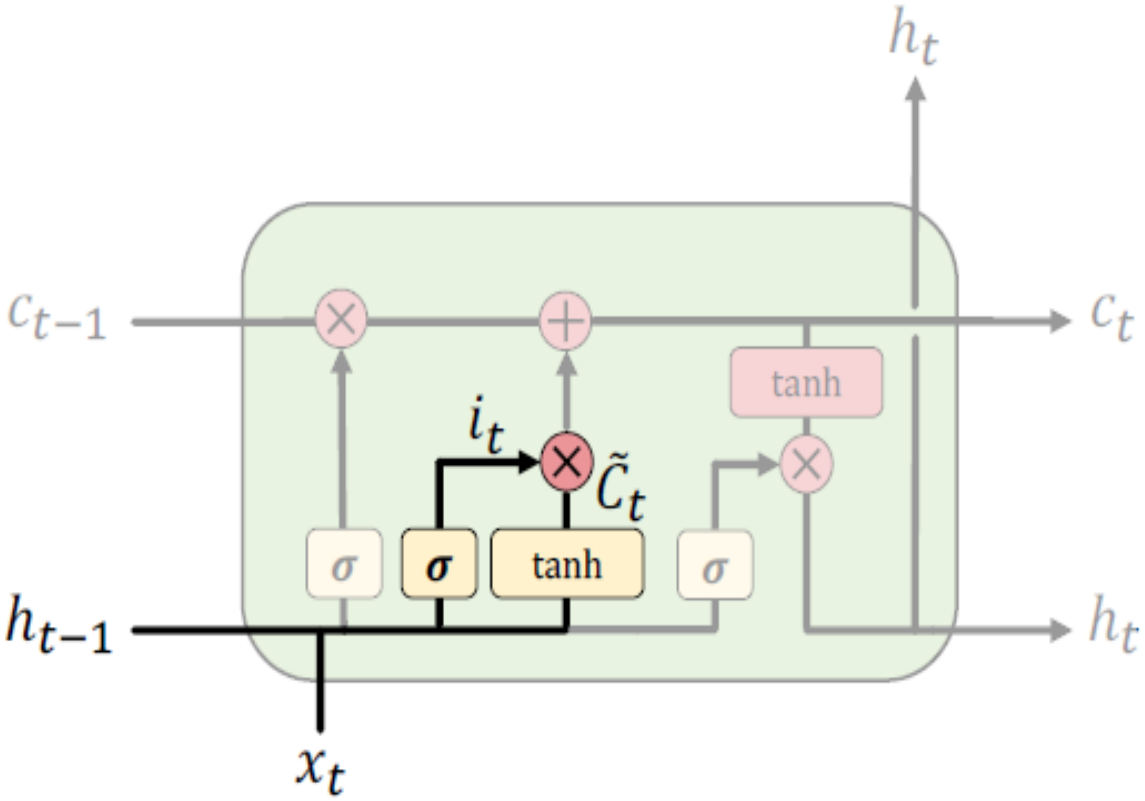
Long Short Term Memory (LSTMs)

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

LSTMs **store relevant** new information into the cell state



LSTMs: identify new information to be stored



$$i_t = \sigma(\boldsymbol{W_i}[\, h_{t-1}, x_t \,] + b_i)$$

$$\tilde{C}_t = \tanh(\boldsymbol{W_C}[\, h_{t-1}, x_t \,] + b_C)$$

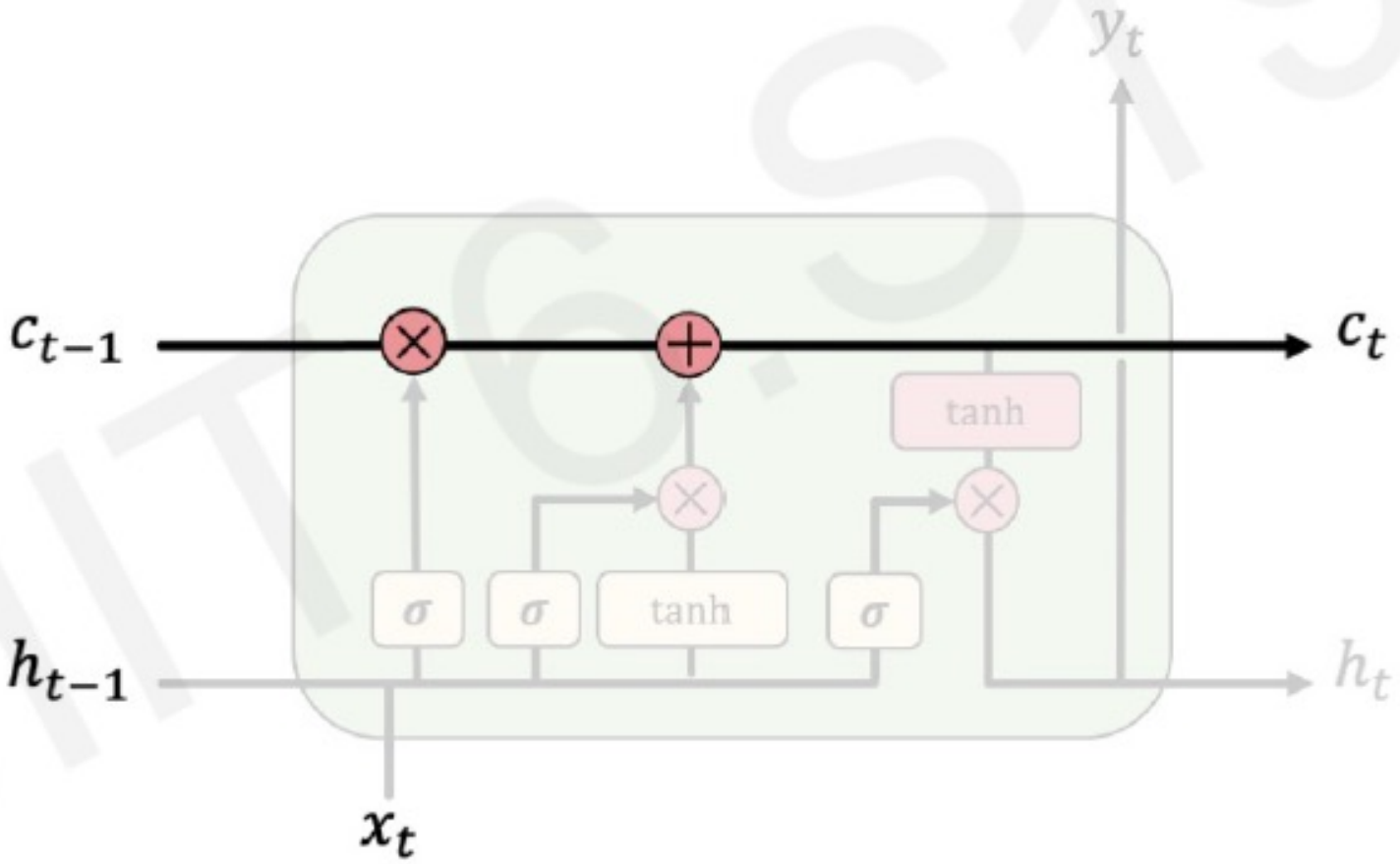
- Sigmoid layer: decide what values to update
- Tanh layer: generate new vector of "candidate values" that could be added to the state

ex: Add gender of new subject to replace that of old subject.

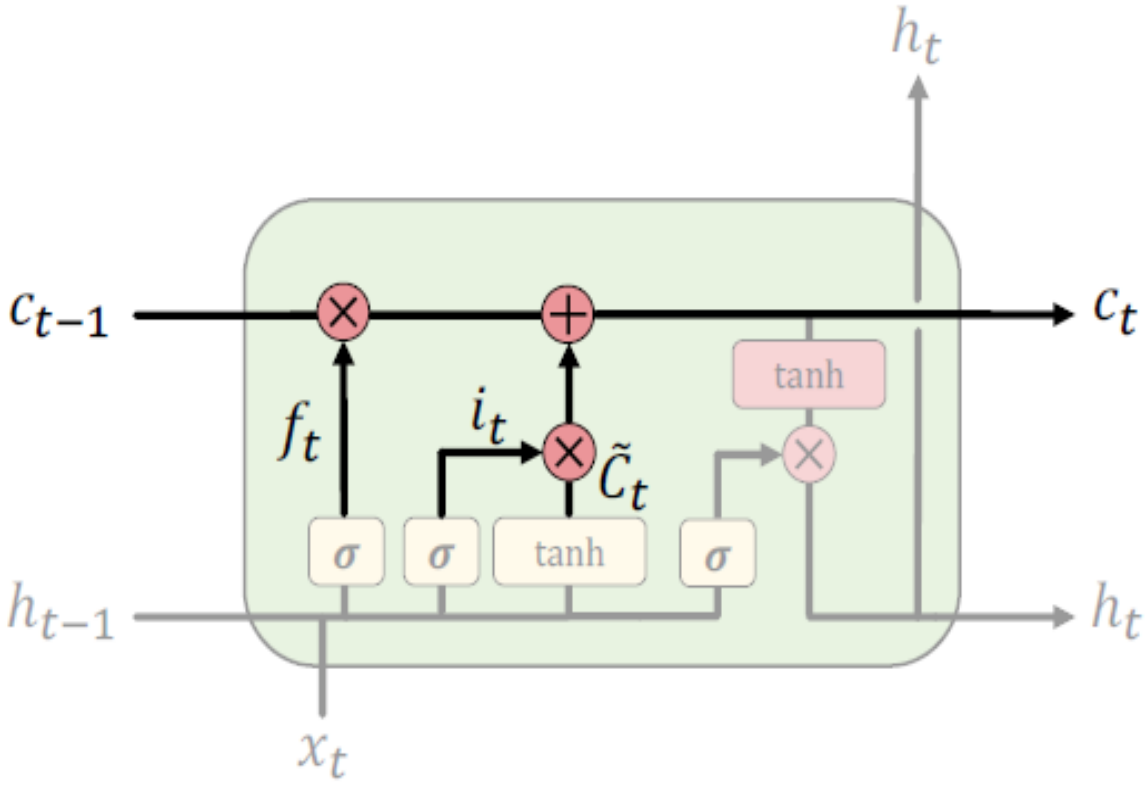
Long Short Term Memory (LSTMs)

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

LSTMs **selectively update** cell state values



LSTMs: update cell state



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

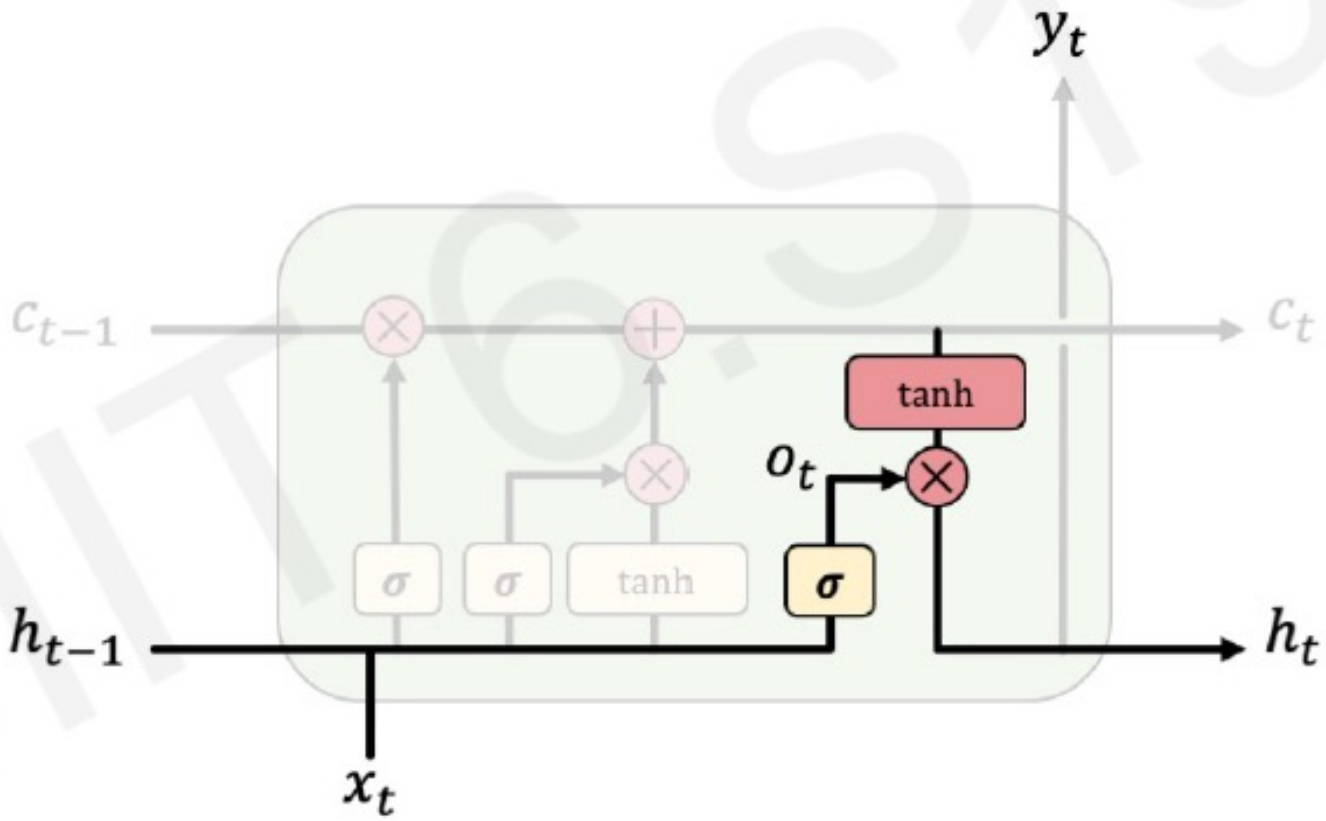
- Apply forget operation to previous internal cell state: $f_t * C_{t-1}$
- Add new candidate values, scaled by how much we decided to update: $i_t * \tilde{C}_t$

ex: Actually drop old information and add new information about subject's gender.

Long Short Term Memory (LSTMs)

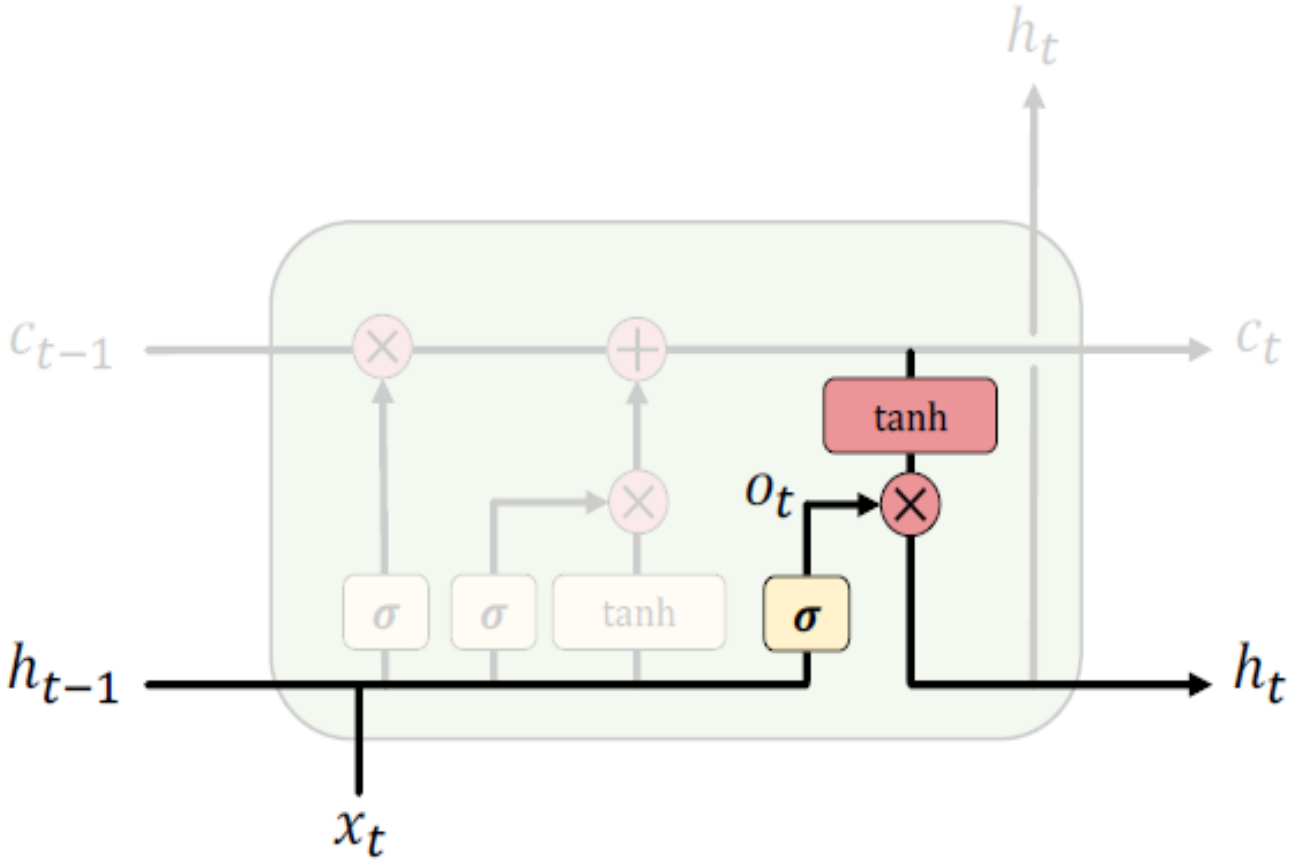
1) Forget 2) Store 3) Update **4) Output**

The **output gate** controls what information is sent to the next time step

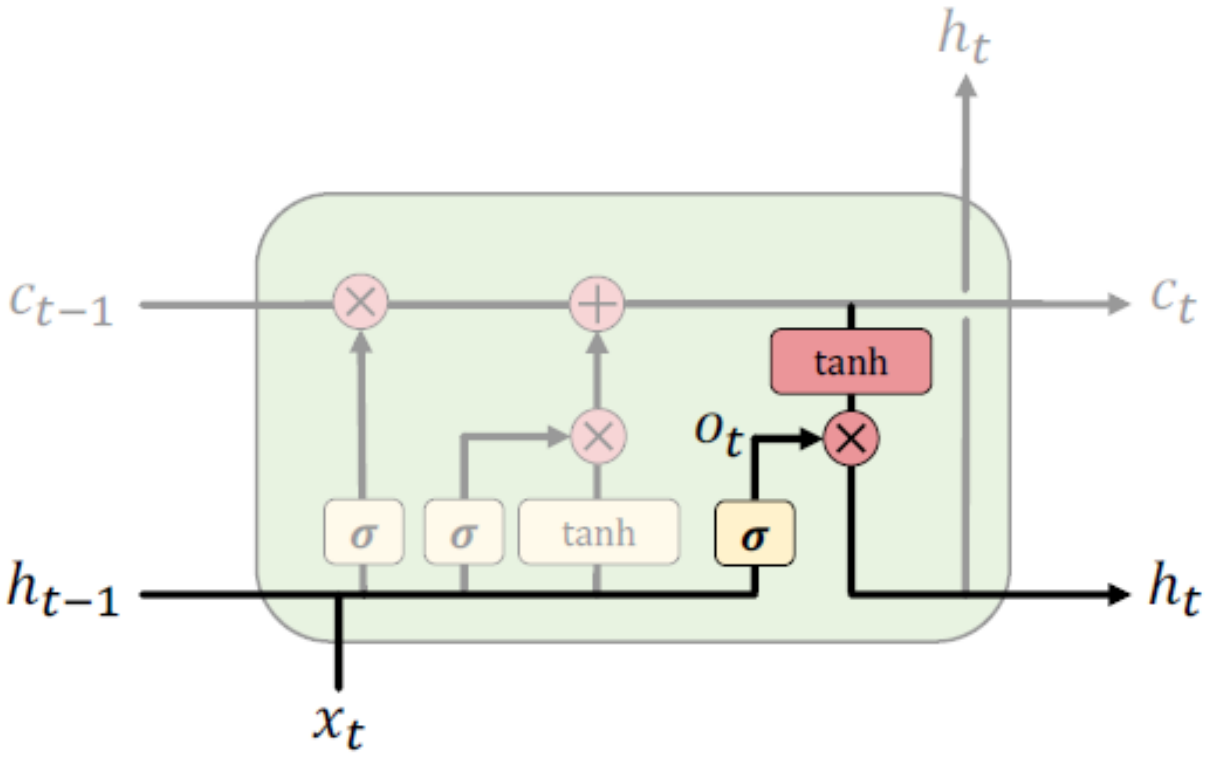


Long Short Term Memory (LSTMs)

LSTMs use an **output gate** to output certain parts of the cell state



LSTMs: output filtered version of cell state



$$o_t = \sigma(\boldsymbol{W_o}[\, h_{t-1}, x_t \,] + b_o)$$

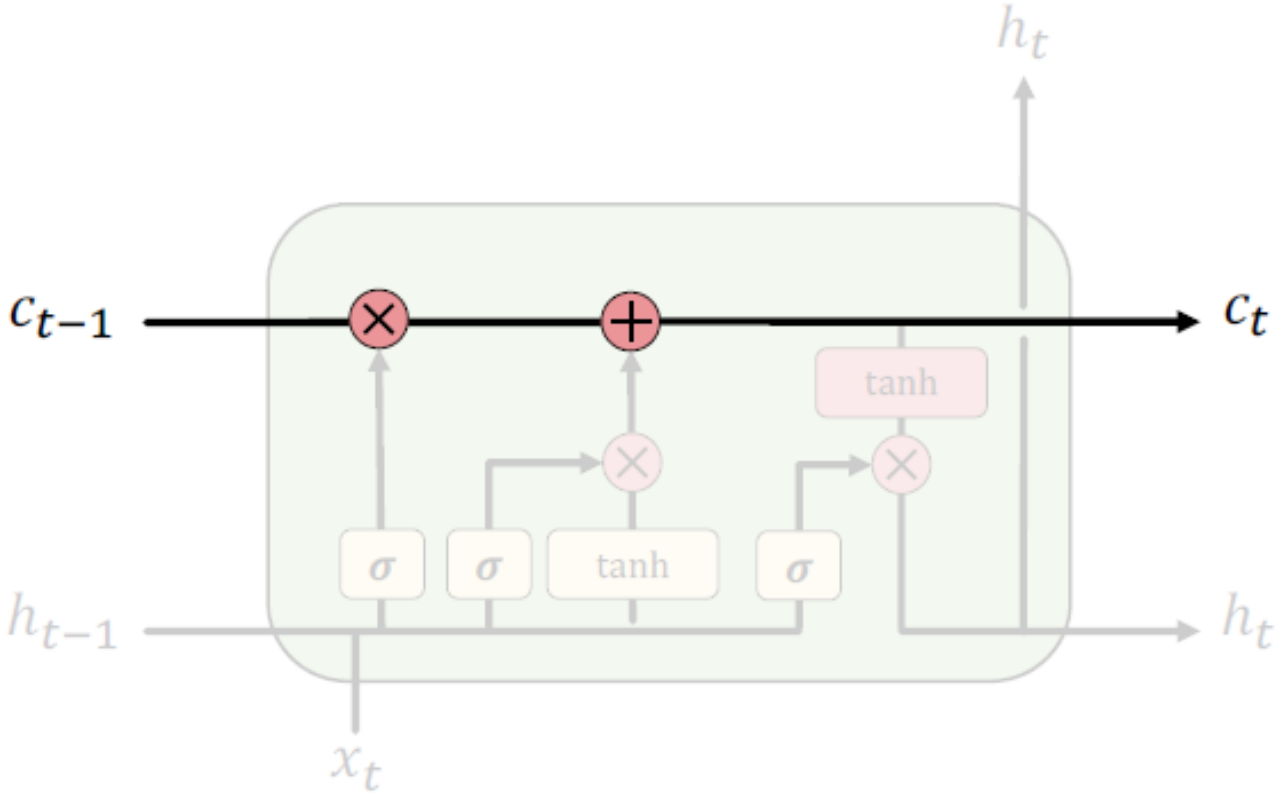
$$h_t = o_t * \tanh(C_t)$$

- Sigmoid layer: decide what parts of state to output
- Tanh layer: squash values between -1 and 1
- $o_t * \tanh(C_t)$: output filtered version of cell state

ex: Having seen a subject, may output information relating to a verb.

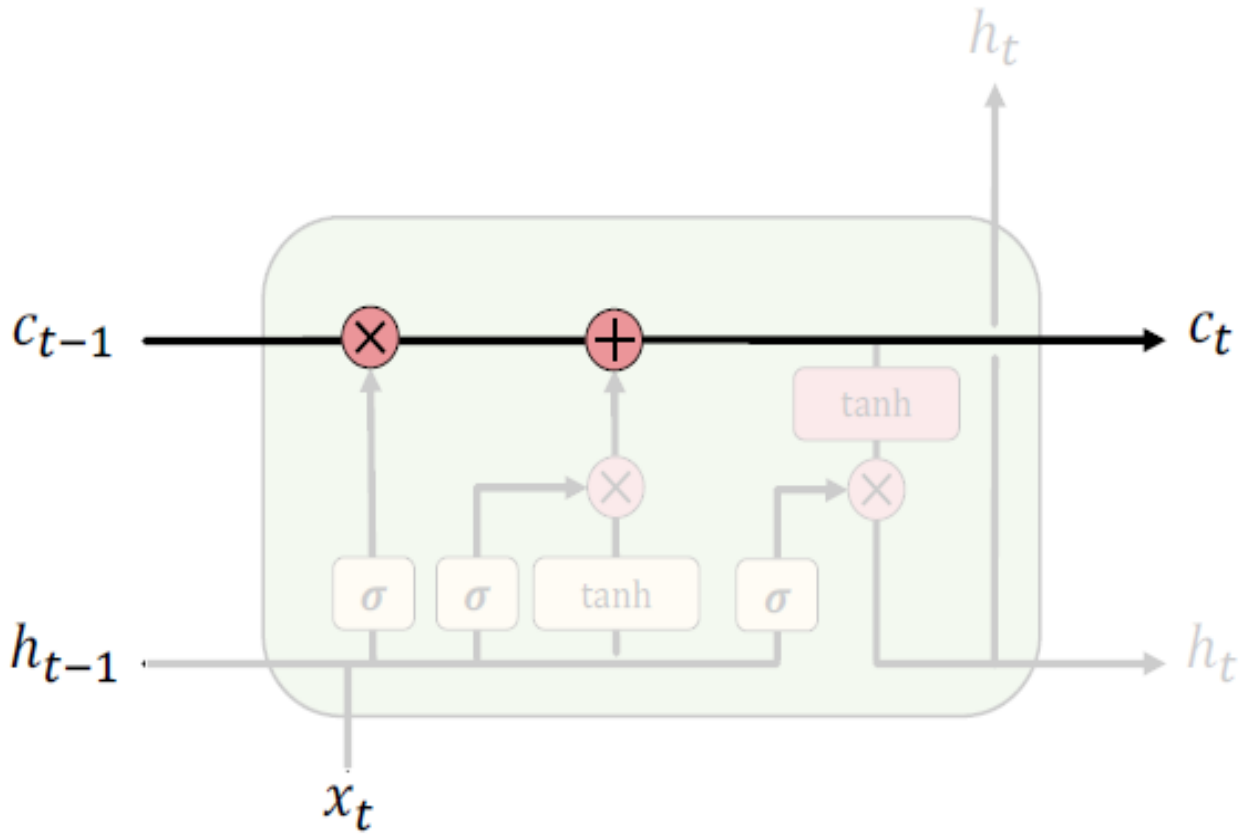
Long Short Term Memory (LSTMs)

LSTMs maintain a **cell state** c_t where it's easy for information to flow



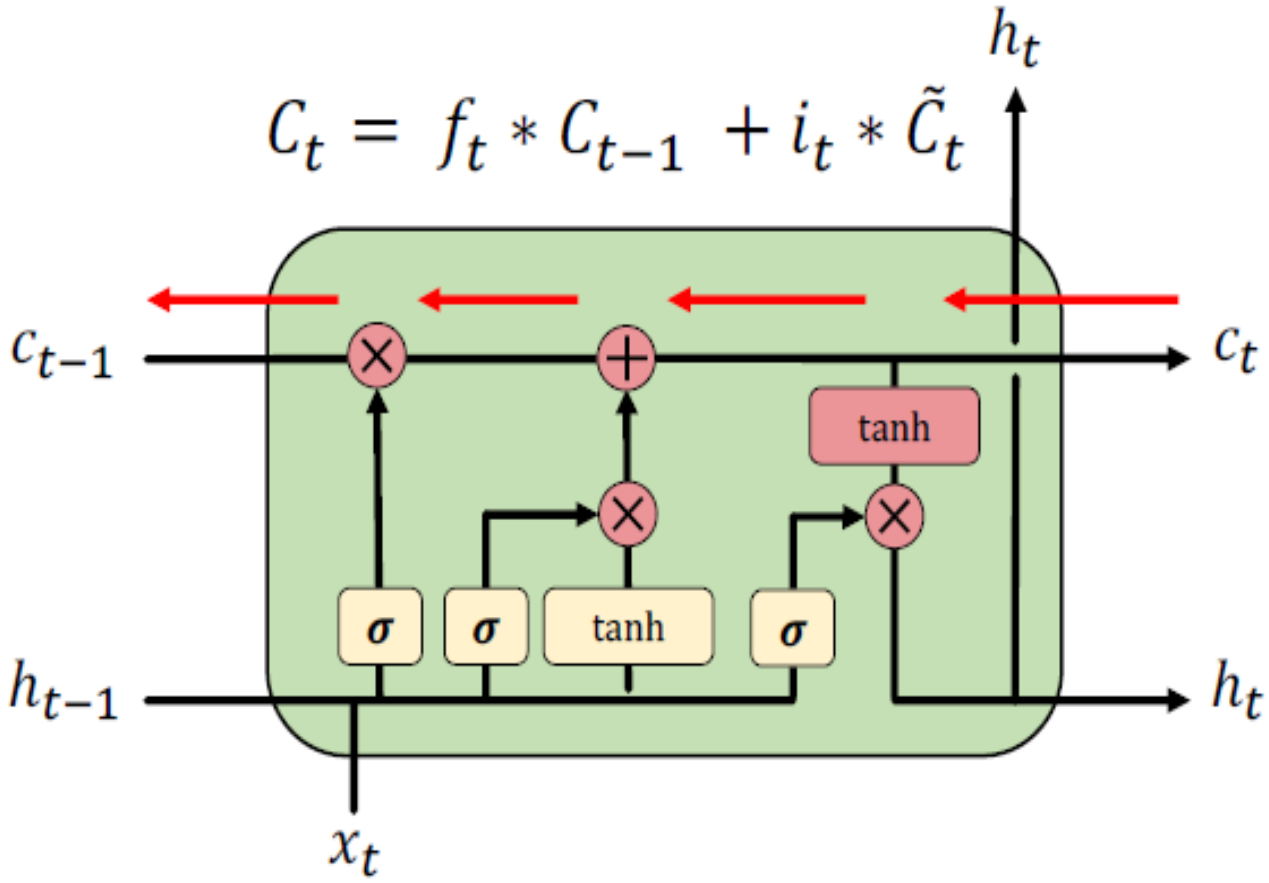
Long Short Term Memory (LSTMs)

LSTMs **selectively update** cell state values

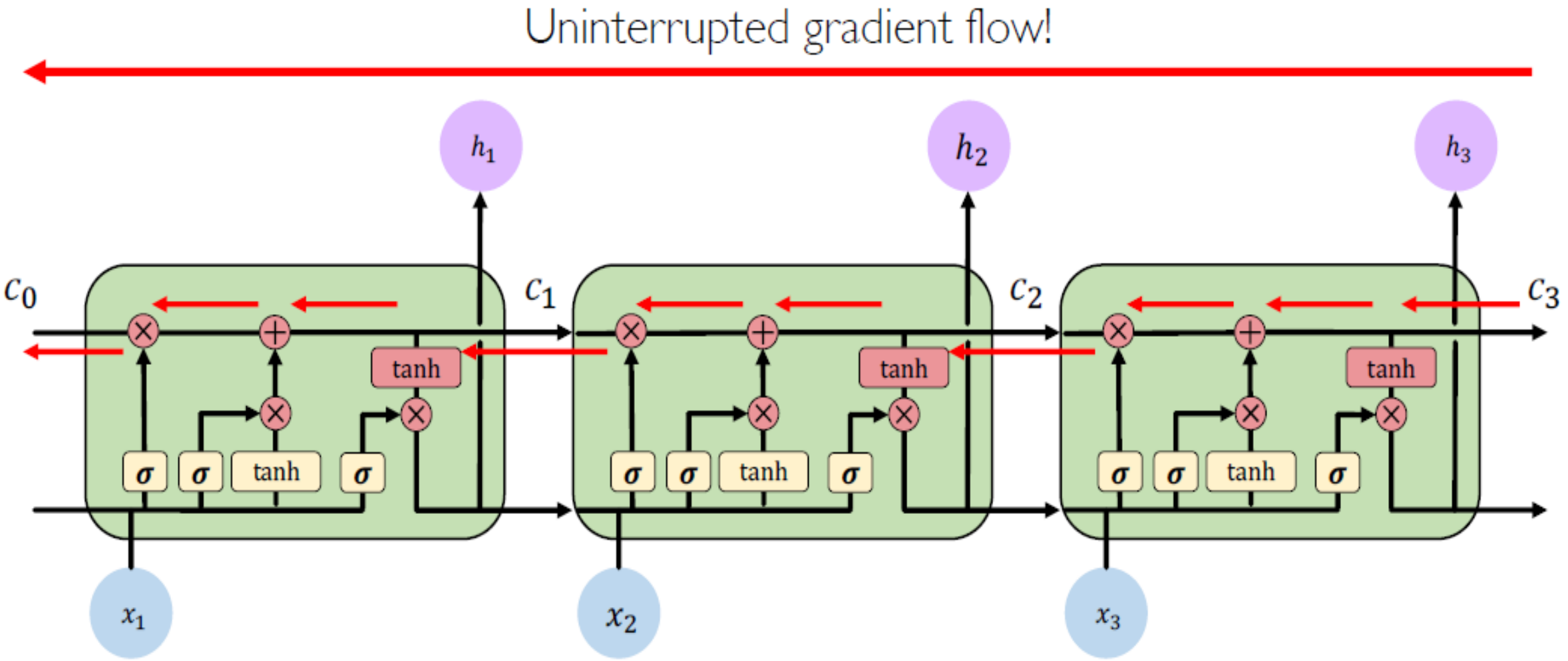


LSTM gradient flow

Backpropagation from C_t to C_{t-1} requires only elementwise multiplication!
No matrix multiplication → avoid vanishing gradient problem.



LSTM gradient flow



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**

LSTM Cells - Takeaways

- In short, an LSTM cell can learn to:
  - Recognize an important input (that's the role of the input gate).
  - Store it in the long-term state, preserve it for as long as it is needed (that's the role of the forget gate).
  - Extract it whenever it is needed.
- This explains why these cells have been amazingly successful at capturing long-term patterns in time series, long texts, audio recordings, and more.

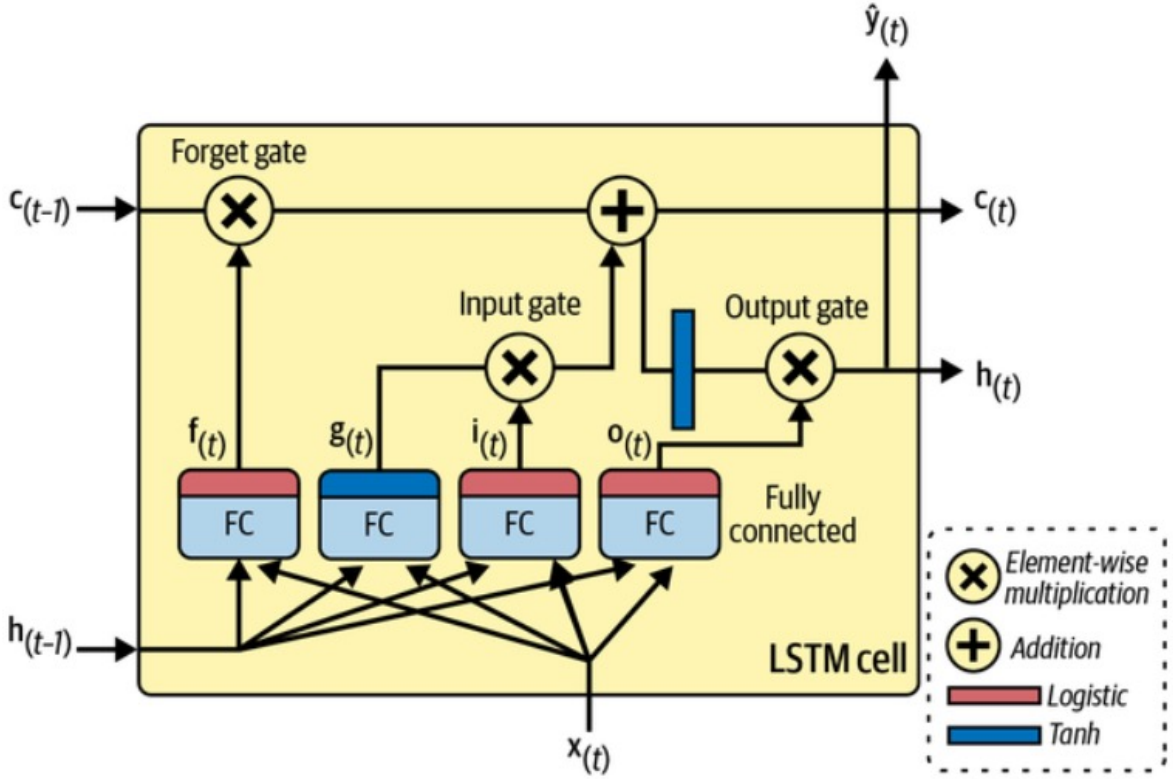


Figure 15-12. An LSTM cell

LSTM Cells – How They Operate

- The key idea is that the network can learn what to store in the long-term state, what to throw away, and what to read from it.
- As the long-term state c(t–1) traverses the network from left to right, you can see that it first goes through a forget gate, dropping some memories, and then it adds some new memories via the addition operation (which adds the memories that were selected by an input gate).
- At each time step, some memories are dropped and some memories are added.
- Moreover, after the addition operation, the long-term state is copied and passed through the tanh function, and then the result is filtered by the output gate. This produces the short-term state h(t) (which is equal to the cell's output for this time step, y(t)).

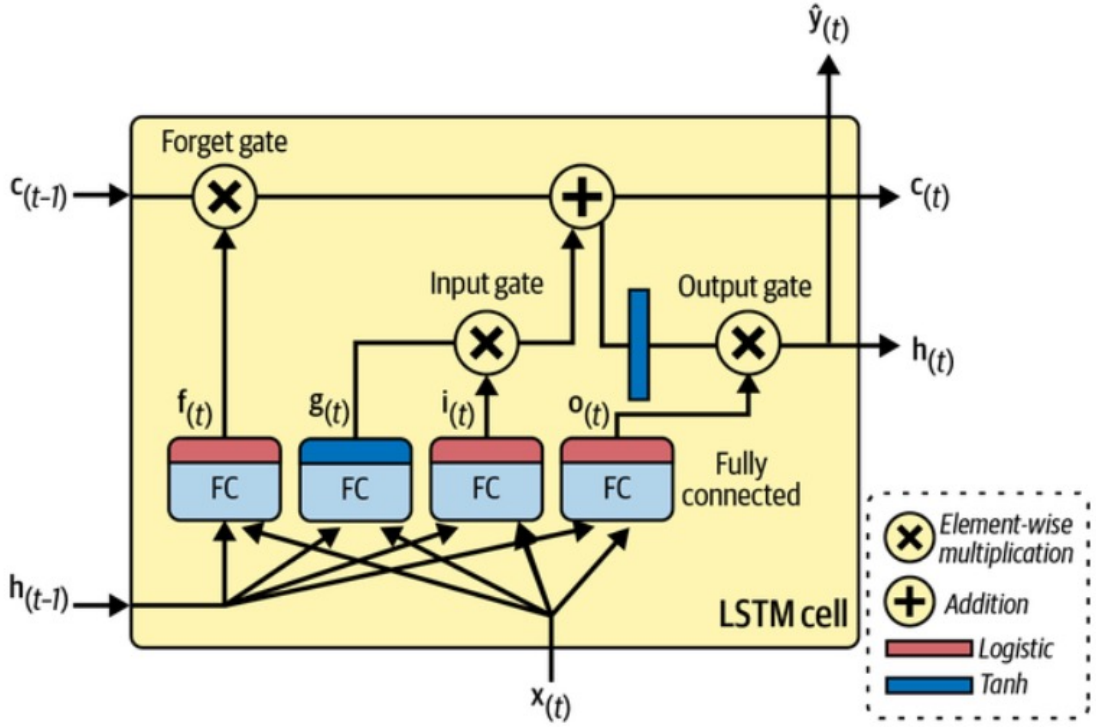


Figure 15-12. An LSTM cell

Gated Recurrent Unit (GRU) Cell

- The GRU cell is a ***simplified version*** of the LSTM cell
- These are the main simplifications:
  - Both state vectors are merged into a single vector h(t).
  - A single gate controller z(t) controls both the forget gate and the input gate. If the gate controller outputs a 1, the forget gate is open (= 1) and the input gate is closed (1 – 1 = 0). If it outputs a 0, the opposite happens.
  - In other words, whenever a memory must be stored, the location where it will be stored is erased first. This is actually a frequent variant to the LSTM cell in and of itself.
  - There is no output gate; the full state vector is output at every time step. However, there is a new gate controller r(t) that controls which part of the previous state will be shown to the main layer (g(t)).

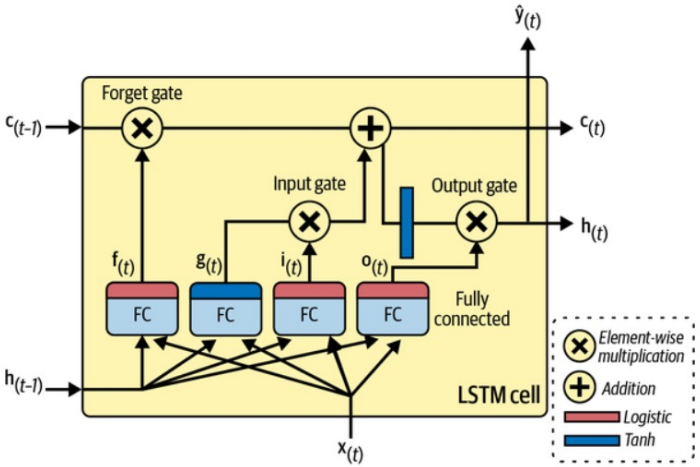


Figure 15-12. An LSTM cell

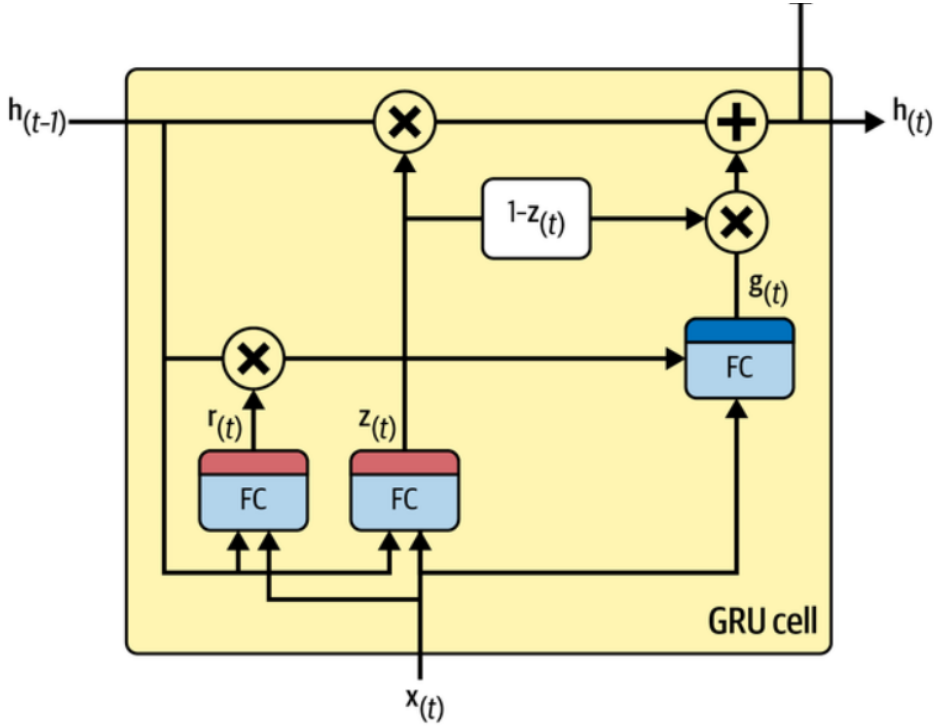


Figure 15-13. GRU cell

LSTMs and GRUs

- LSTM and GRU cells are one of the main reasons behind the success of RNNs.

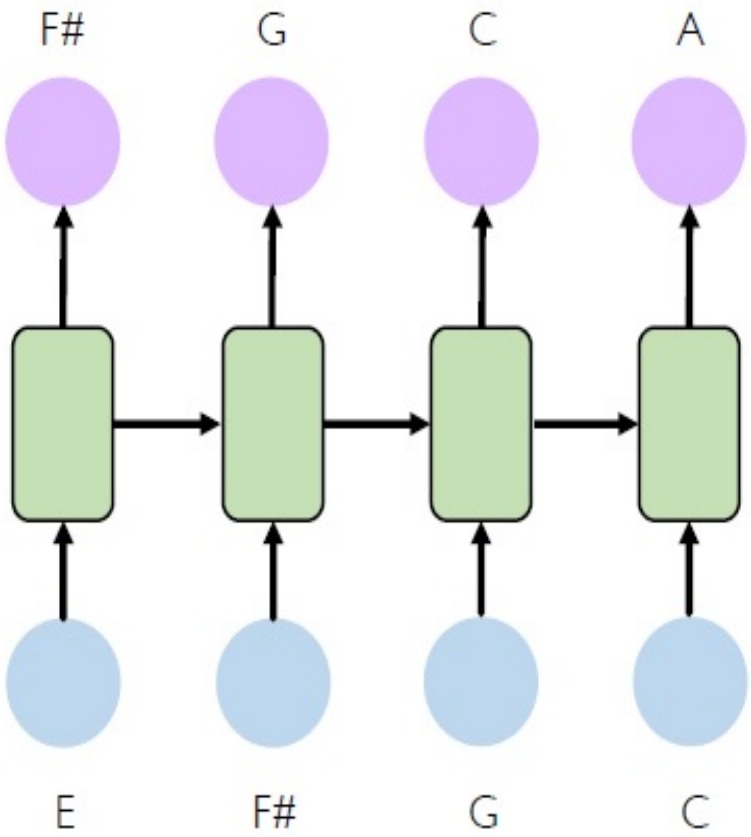
- Yet while they can tackle much longer sequences than simple RNNs, they still have a fairly limited short-term memory, and they have a hard time learning long-term patterns in sequences of 100 time steps or more, such as audio samples, long time series, or long sentences.

THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

RNN Applications

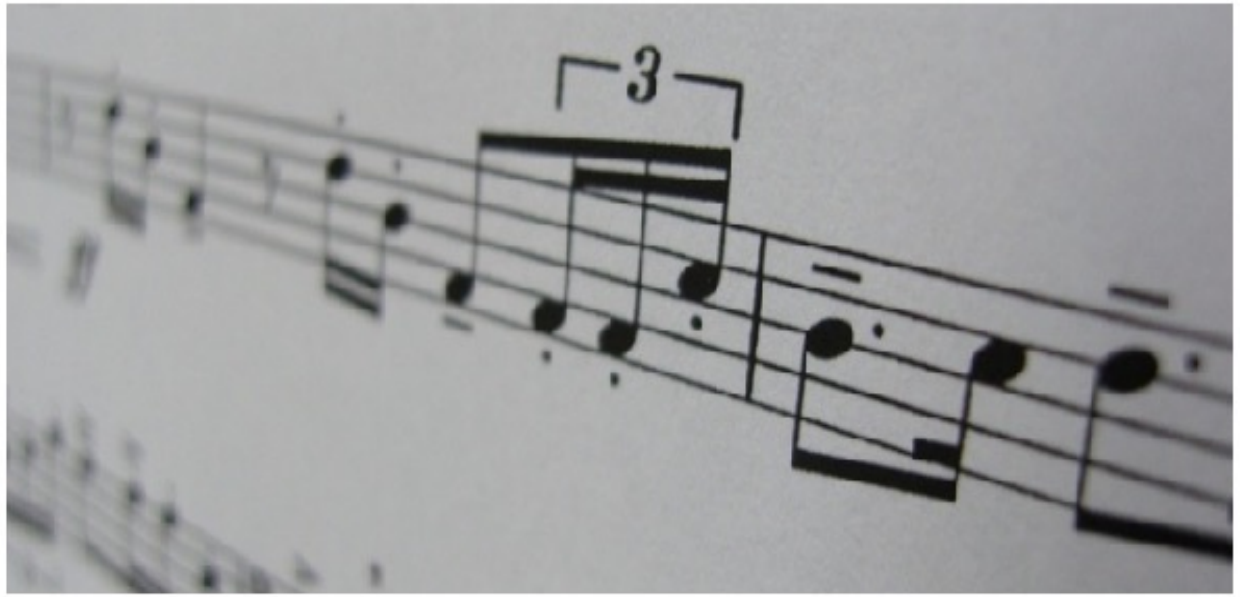
MIT 6.S191 – Deep Sequence Modeling, Introtodeeplearning.com

Example task: music generation



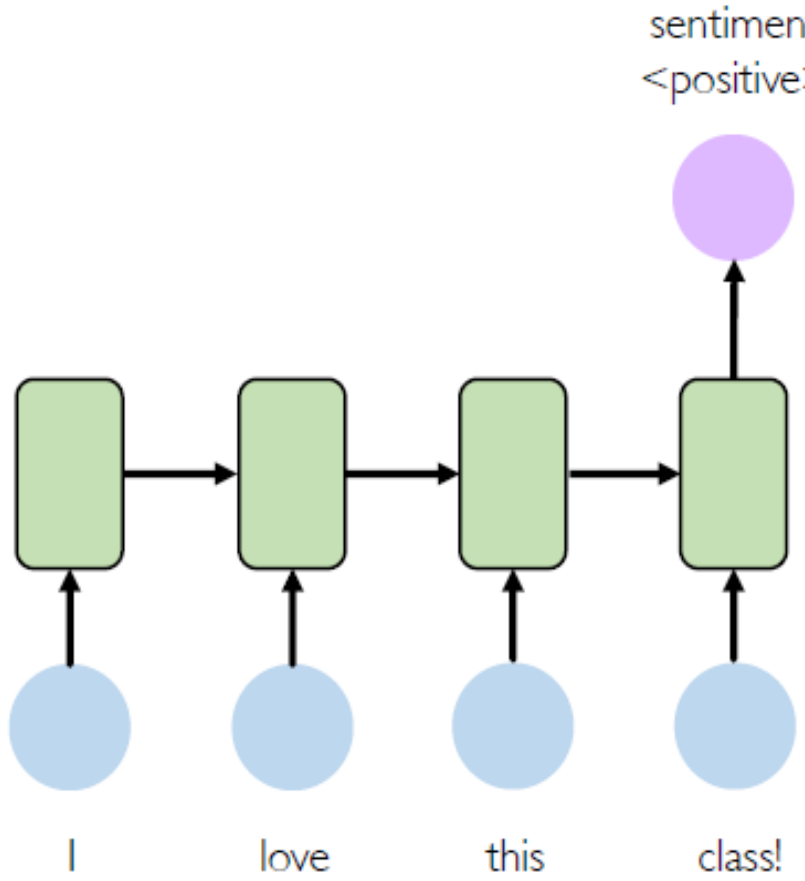
Input: sheet music

Output: next character in sheet music



Adapted from H. Suresh, 6.S191 2018

Example task: sentiment classification

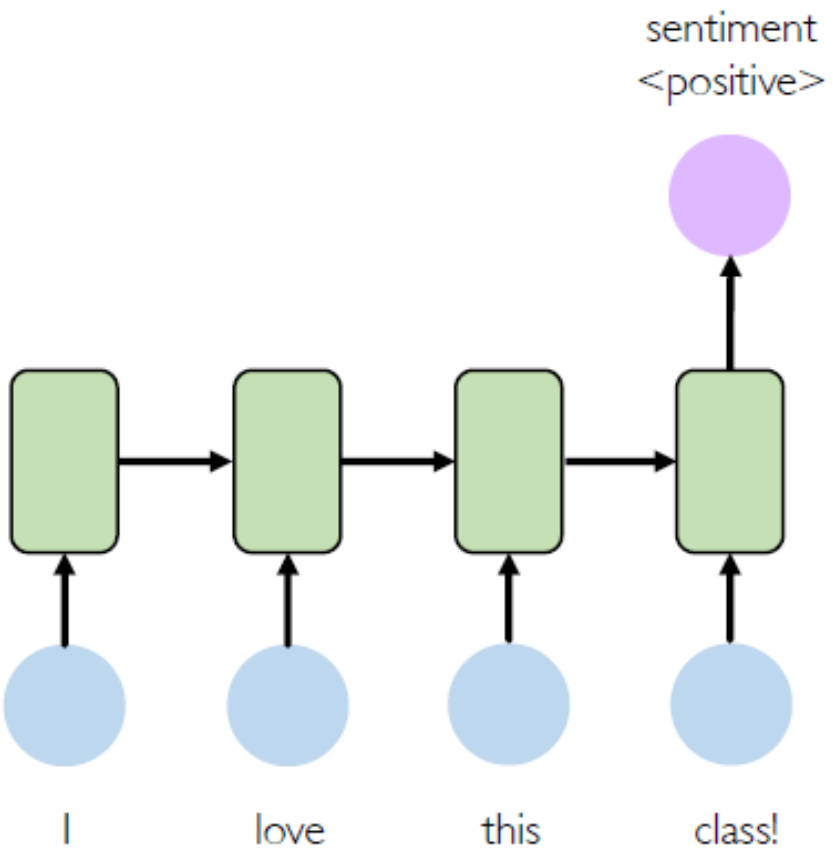


Input: sequence of words

Output: probability of having positive sentiment

Adapted from H. Suresh, 6.S191 2018

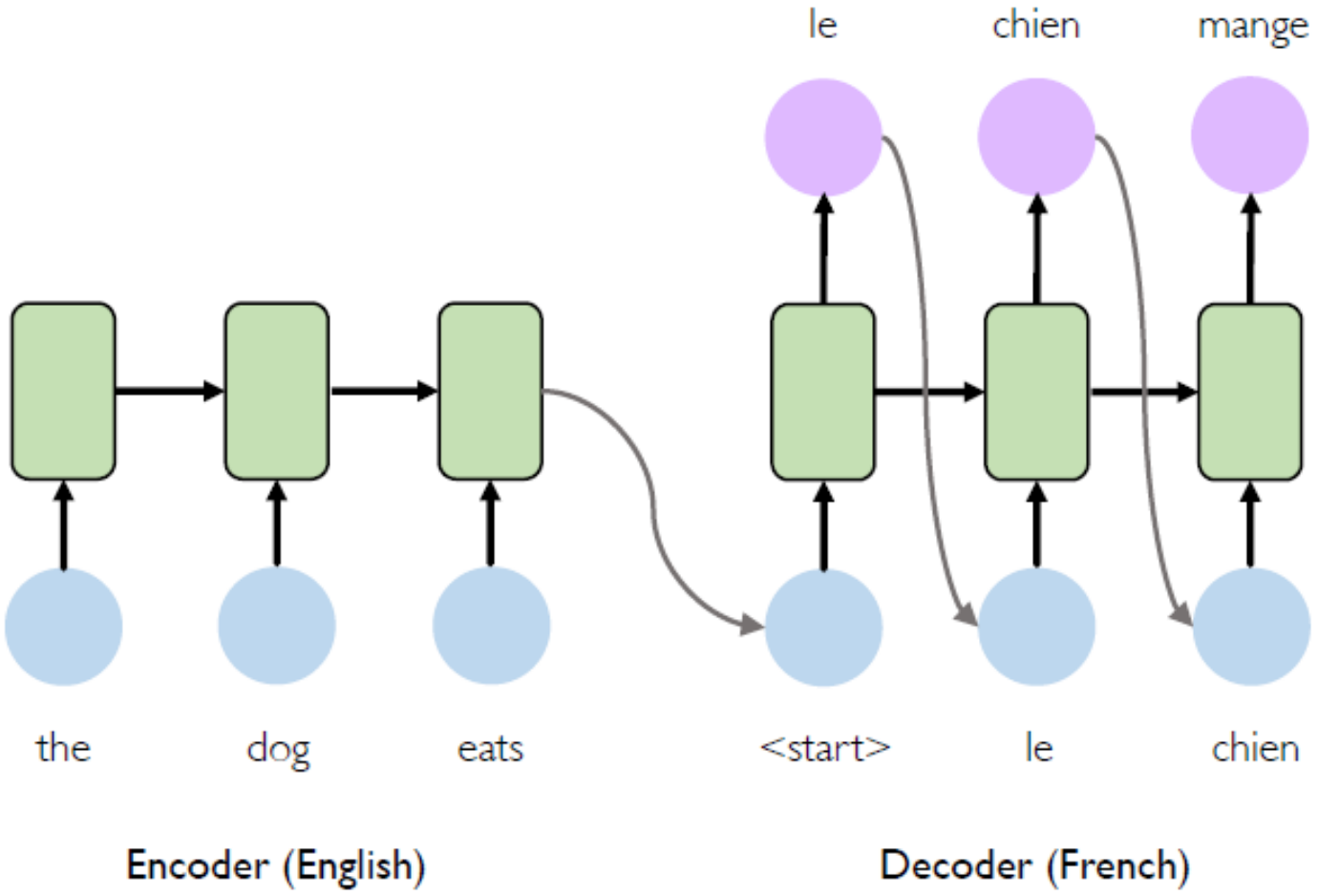
Example task: sentiment classification



Adapted from H. Suresh, 6.S191 2018

Tweet sentiment classification

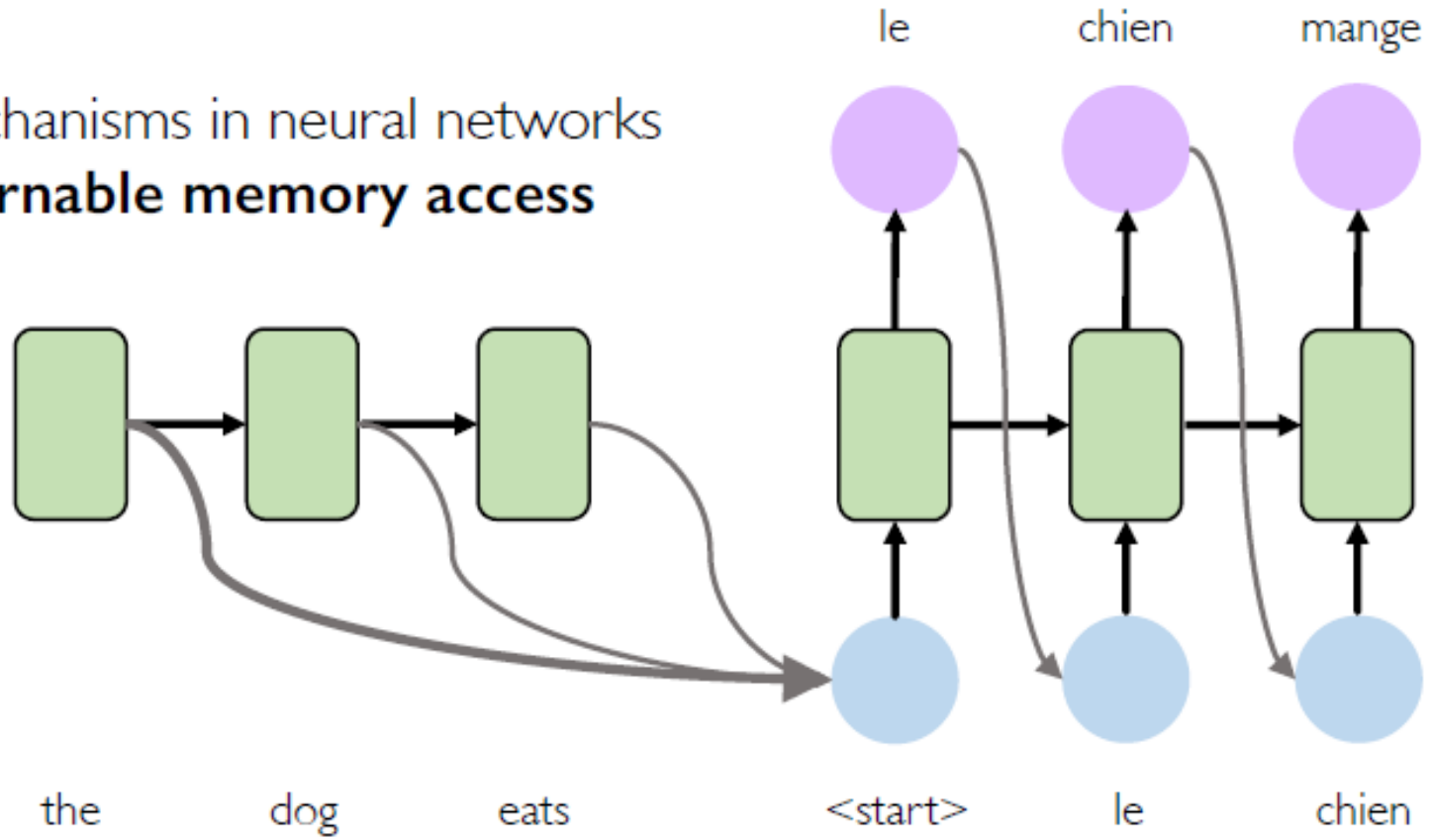
Example task: machine translation



Adapted from H. Suresh, 6.S191 2018

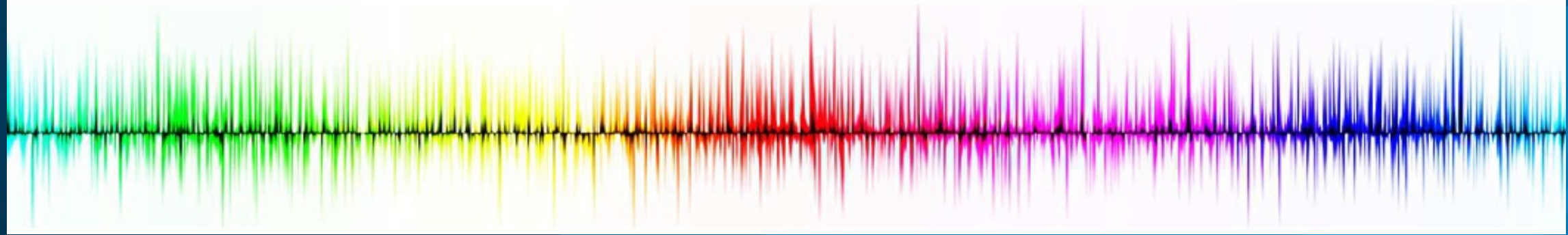
Attention mechanisms

Attention mechanisms in neural networks provide **learnable memory access**



Recurrent neural networks (RNNs)

1. RNNs are well suited for **sequence modeling** tasks
2. Model sequences via a **recurrence relation**
3. Training RNNs with **backpropagation through time**
4. Gated cells like **LSTMs** let us model **long-term dependencies**
5. Models for **music generation**, classification, machine translation



THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

Homework Overview

Professor Hamza F. Alsarhan
SEAS 8505

This Week

- Reading:
  - Chapter 15 in the textbook
- HW #6 (Run/Write Python Script in Google Colab first, and then answer the homework questions)
- Discussion #6

- Reminder: No extensions provided. Start assignments early!

Next Steps

- Come to office hours with any questions you may have.
- Work on your HW and Discussion and submit them by 9:00 am ET on Saturday.
- See you next class!

Thank you!