Deep Sequence Models

Anisio Lacerda
Rodrygo Santos

Recap

Tokenization

From words to embeddings

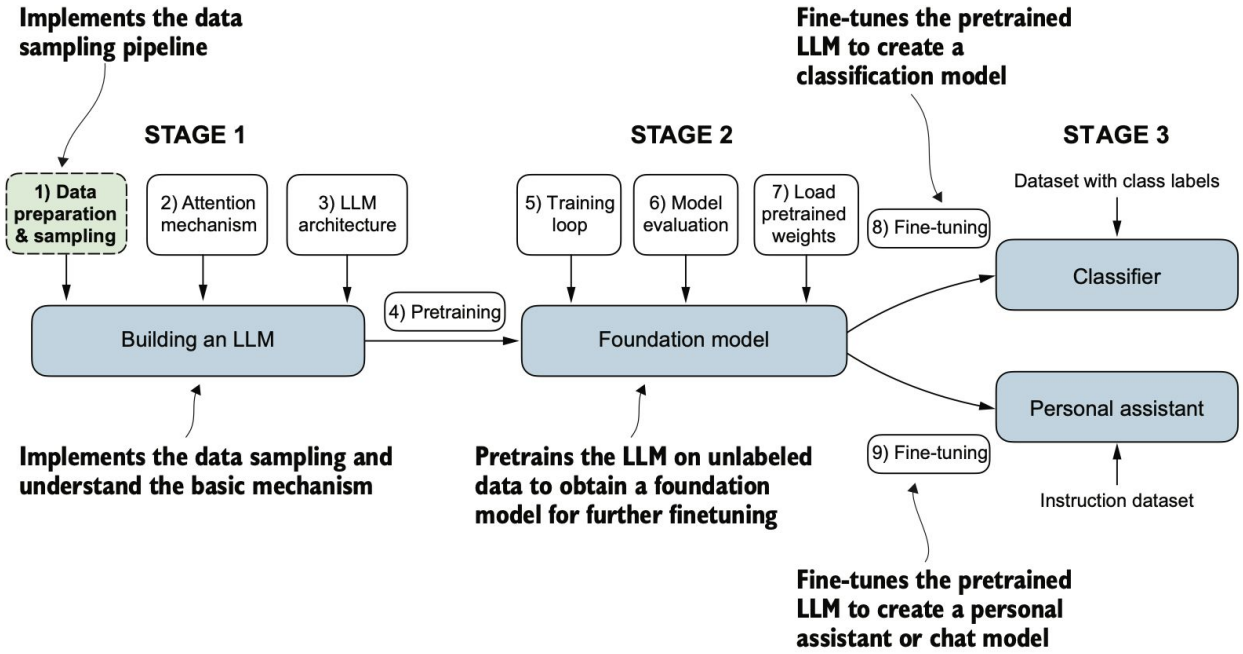


Figure 2.1 The three main stages of coding an LLM. This chapter focuses on step 1 of stage 1: implementing the data sample pipeline.

Today

Sequence models and language models

Recurrent Neural Networks (RNN)

GRU and LSTM

Brief intro to Attention

Sequence models and language models

Working with Sequences

Traditionally,

Collect observation pairs `(xi,yi) ~ P(X,Y)` for training.

Estimate `y | x ~ P(y | x)` for unseen `x' ~ P(x)`.

Assume instances are independent and identically distributed (IID).

Examples

Images & objects.

Regression/classification problems.

> The order of the data did not matter.

Fundamentals of Sequence Modeling

The Independence Assumption Problem

Within a sequence, adjacent elements are highly correlated.

Dependent Random Variables

Working with Sequences

We assume that **entire sequences** are **sampled independently**.

We **cannot** assume that **data arriving** at **each time step** are **independent** of each other. E,g,:

Entire documents

> **Words** that **likely** to **appear later depend** heavily on **words occurring earlier**.

Patient trajectories

> **Medicine** a **patient** is likely to receive on the **10th** day of a hospital **visit depends** heavily on what **transpired** in the **previous nine days**.

Working with Sequences

No surprise, we **assume** such **dependence** to model them as a sequence.

We do **not** require **independence** of our **sequences**.

We require only that the **sequences** are **sampled** from some **fixed** underlying **distribution** over entire sequences.

This allows tackling phenomena as

documents looking **different** at the **beginning** than at the **end**;

patient status evolving either towards recovery or death;

customer taste evolving in **predictable** ways over the **course** of continued **interaction** with a recommender system.

Fundamentals of Sequence Modeling

The Independence Assumption Problem

Within a sequence, adjacent elements are highly correlated.

The Mathematical Framework

– Seq2Seq – Autoregressive

Sequence-to-Sequence Modeling

Aim to learn a mapping from input sequences $X = (x_1, \ldots, x_m)$ to output sequences $Y = (y1, \ldots, yn)$ where m and n may differ.

The joint probability we seek to model is

$$P(Y \mid X) = P(y_1, \ldots, y_n \mid x_1, \ldots, x_m)$$

Autoregressive Modeling

The **goal** is to **estimate** the conditional probability $P(x_t \mid x_1, \ldots, x_{t-1})$.

This captures the essence of language modeling

> we **predict** the **next token** given all **previous tokens** in a **sequence**.

The **challenge** lies in the fact that the **conditioning set grows with time**, creating a variable-length input problem.

> although long sequences are available, it **may not be necessary to look back so far** in the history to predicting the feature.
>
> condition on some **window** of length τ and only use $x_{t-1}, \ldots, x_{t-\tau}$ observations.
>
> the **number of arguments is always the same**, at least for $t > \tau$.

Fundamentals of Sequence Modeling

The Independence Assumption Problem

Within a sequence, adjacent elements are highly correlated.

The Mathematical Framework

– Seq2Seq – Autoregressive

The Variable Length Challenge

A key challenge in sequence modeling is handling variable-length inputs and outputs.

Temporal Dependencies and Memory

Sequence modeling requires maintaining information about past events to make predictions about future events.

Temporal Dependencies and Memory

Consider the sentence: *The cat, which had been sleeping peacefully in the warm sunlight streaming through the window, suddenly woke **up***

To predict ***up*** correctly, the model must

remember that the subject is *cat*

the verb is *woke*

Temporal Dependencies and Memory

Several dependencies:

Short-range dependencies: **adjacent words** often have strong **syntactic** and **semantic** relationships

- articles precede nouns, adjectives modify nearby nouns, and auxiliary verbs appear near main verbs.

Long-range dependencies: **subjects** and **verbs** may be **separated** by arbitrary amounts of **intervening text**.

- pronouns may refer to entities mentioned much earlier in the discourse, thematic elements may span entire documents.

Hierarchical dependencies: language exhibits **nested structure**, with phrases embedded within clauses, clauses within sentences, and sentences within paragraphs.

- capturing these dependencies requires sophisticated memory mechanisms.

Fundamentals of Sequence Modeling

The Independence Assumption Problem

Within a sequence, adjacent elements are highly correlated.

The Mathematical Framework

– Seq2Seq – Autoregressive

The Variable Length Challenge

A key challenge in sequence modeling is handling variable-length inputs and outputs.

Temporal Dependencies and Memory

Sequence modeling requires maintaining information about past events to make predictions about future events.

The Alignment Problem

Which parts of the input sequence correspond to which parts of the output sequence.

The Alignment Problem

Which parts of the input sequence correspond to which parts of the output sequence

Rarely one-to-one and may involve complex reordering patterns.

Consider our translation example: **"I love you"** → **"Te amo."** The alignment is:

"you" → "te" (reordered to the beginning)

"I love" → "amo" (combined into a single conjugated verb)

The Alignment Problem

Traditional approaches to Seq2Seq modeling **attempted** to solve the **alignment problem** implicitly by **compressing** all input information into a **fixed-size representation**.

This creates **bottlenecks** that **attention mechanisms** were designed to **overcome**

Alignment reflects how **meaning** is **encoded** and **transformed** across different **representational systems**.

Understanding how **neural networks** learn to **solve alignment** problems provides **insights** into **their capacity** for abstract **reasoning** and **symbolic manipulation**.

Language Models

Markov models and n-grams for language modeling

Imagine you're trying to **predict** the **next word someone** will **say**.

An **n-gram model** works by assuming the person has a very short memory

A **bigram model** (n=2) assumes the next word only depends on the 1 previous word, i.e., the person only remembers the very last word they heard

A **trigram model** (n=3) assumes the next word only depends on the 2 previous words, i.e., their memory is slightly better

> **Core idea**
>
> **Markov assumption** – you only need to look at a fixed, recent history.

The Problem of Short Memory

What if the **context** you need is **further back**?

Consider the sentence

> *The children who live down the street and play in the park every afternoon **are**…*

A trigram model (n=3) trying to predict the next word only looks at the two previous words: *every afternoon*

Based on that context, it might predict *is* or *was*.

> *It has no memory of the subject of the sentence, "children", which is much further back.*
>
> *To correctly predict "are," the model needs to see "children".*

Let's increase the memory size: **13-gram model could see "children".**

The Exponential Cost of a Better Memory

Increasing "memory" (n) causes an explosive, **exponential growth** in the number of things it has to remember.

Imagine vocabulary (V) of just **10 words**.

For a bigram model (n=2)

The model needs to store the probability of every words following every other word.

The number of possible combinations is $|V|*|V| = |V|^2 = 10^2 = 100$.

For a trigram model (n=3)

The model needs to store the prob. of every word following every two-word sequence.

The number of possible combinations is $|V|*|V|*|V| = |V|^3 = 10^3 = 1000$.

For a 4-gram model (n=4)

The number of possible combinations is $|V|*|V|*|V|*|V| = |V|^4 = 10^4 = 10000$.

The Exponential Cost of a Better Memory

Pattern: the number of parameters the model needs to store is $|V|^n$

Imagine vocabulary (V) of just **50,000 words**.

For a bigram model (n=2)

parameters: $|V|*|V| = |V|^2 = 50{,}000^2 = 2.5$ billion.

For a trigram model (n=3)

parameters: $|V|*|V| = |V|^3 = 50{,}000^3 = 125$ trillion.

For a 4-gram model (n=4)

parameters: $|V|*|V| = |V|^4 = 50{,}000^4 = 6.25$ quadrillion.

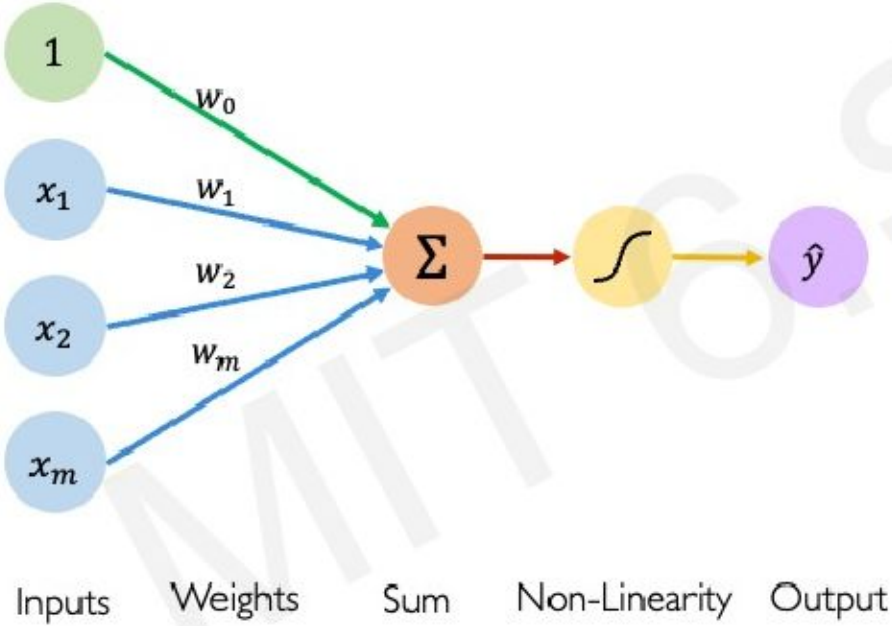
Increasing n is computationally infeasible

Storage – you may *run out of memory* to store the massive # of params.

Data sparsity – you would need an *huge amount of text* to ever *see* most of these *combinations*, so you couldn't calculate reliable probabilities for them anyway.

Neurons with Recurrence

The Perceptron: Forward Propagation



Output

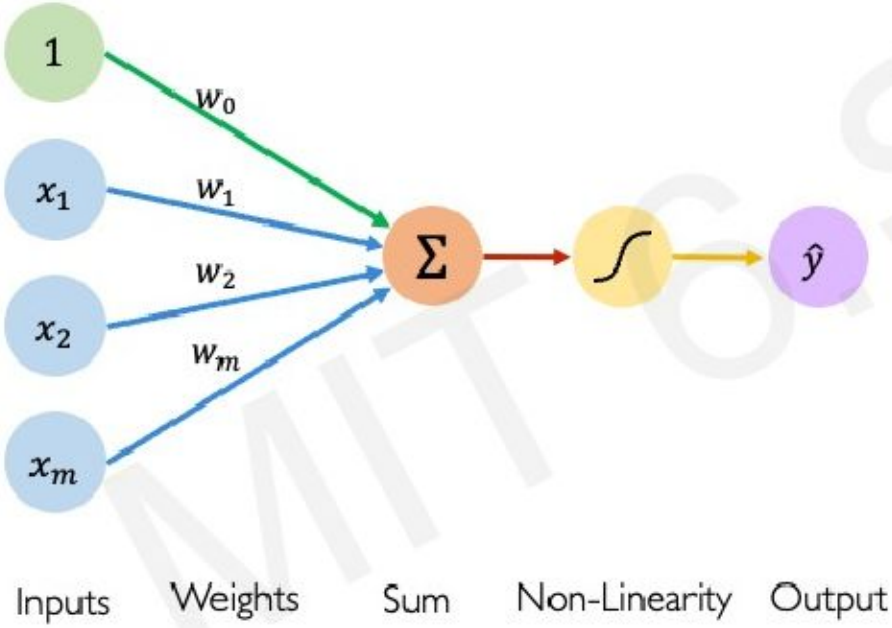
Linear combination of inputs

$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i\, w_i\right)$$

Non-linear activation function

Bias

The Perceptron: Forward Propagation

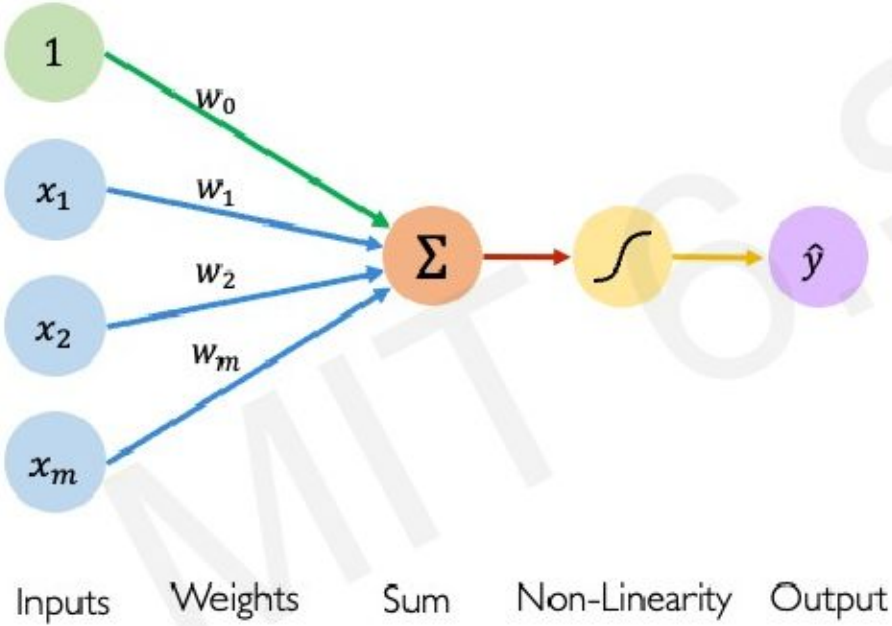


$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i\, w_i\right)$$

$$\hat{y} = g\left(w_0 + \boldsymbol{X}^T \boldsymbol{W}\right)$$

where: $\boldsymbol{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$ and $\boldsymbol{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$

The Perceptron: Forward Propagation

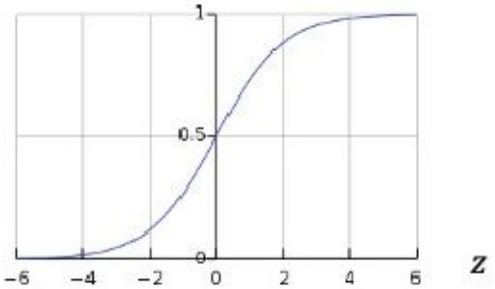


Activation Functions

$$\hat{y} = g\left(w_0 + \boldsymbol{X}^T \boldsymbol{W} \right)$$

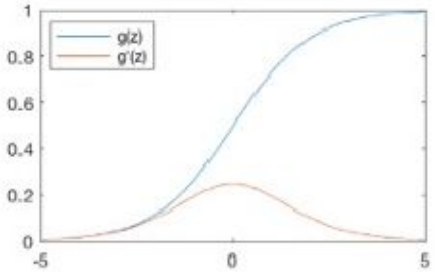
- Example: sigmoid function

$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$



Common Activation Functions

Sigmoid Function

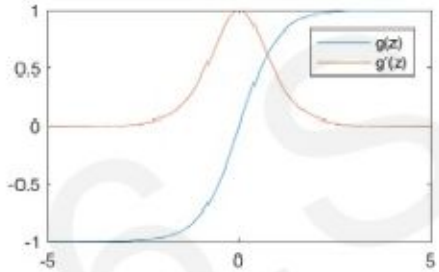


$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$

`tf.math.sigmoid(z)`

Hyperbolic Tangent

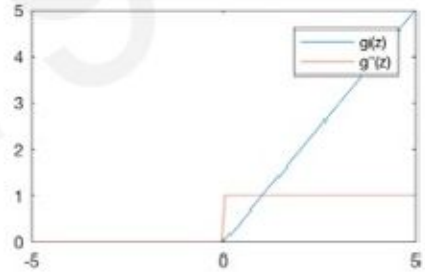


$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

`tf.math.tanh(z)`

Rectified Linear Unit (ReLU)



$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

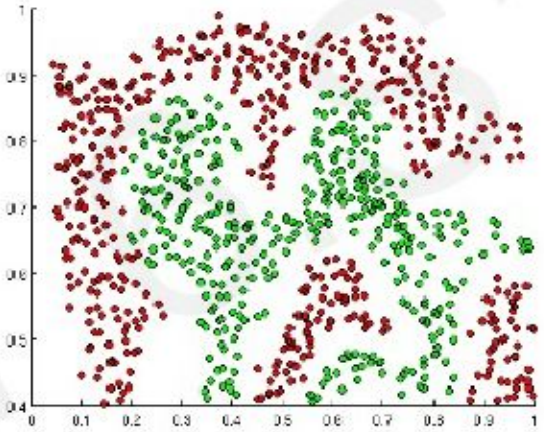
`tf.nn.relu(z)`

TensorFlow code blocks

NOTE: All activation functions are non-linear

Importance of Activation Functions

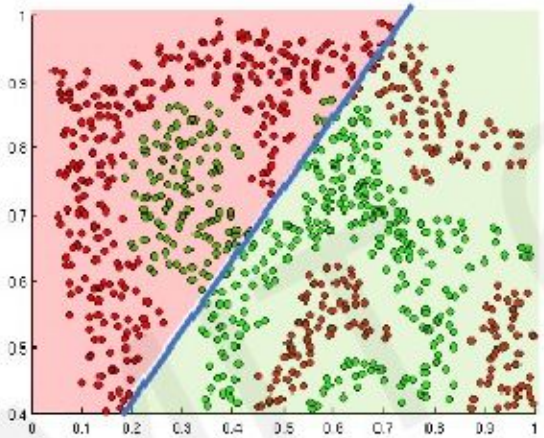
The purpose of activation functions is to ***introduce non-linearities*** *into the network*



What if we wanted to build a neural network to distinguish green vs red points?

Importance of Activation Functions

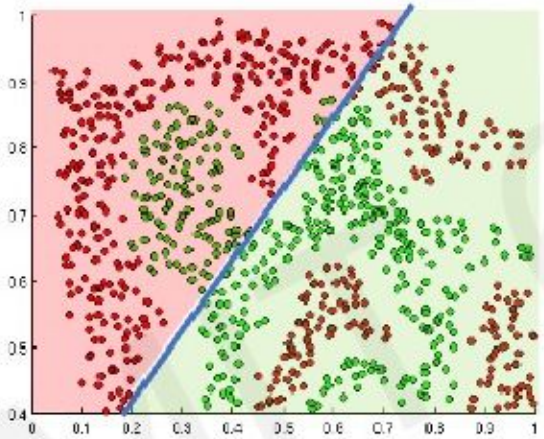
*The purpose of activation functions is to **introduce non-linearities** into the network*



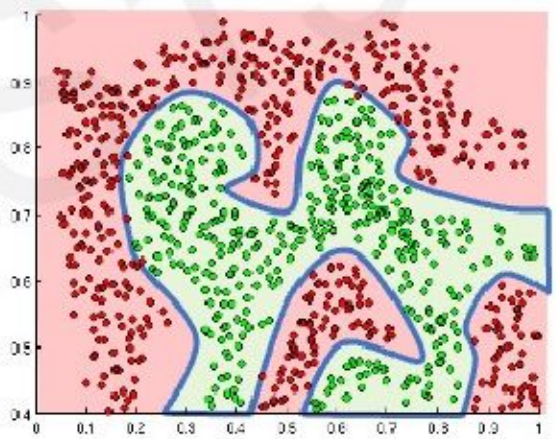
Linear activation functions produce linear decisions no matter the network size

Importance of Activation Functions

*The purpose of activation functions is to **introduce non-linearities** into the network*

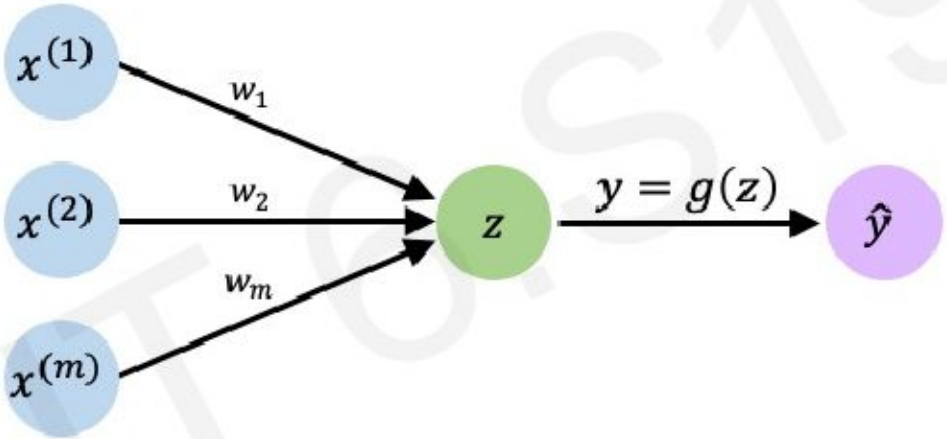


Linear activation functions produce linear decisions no matter the network size

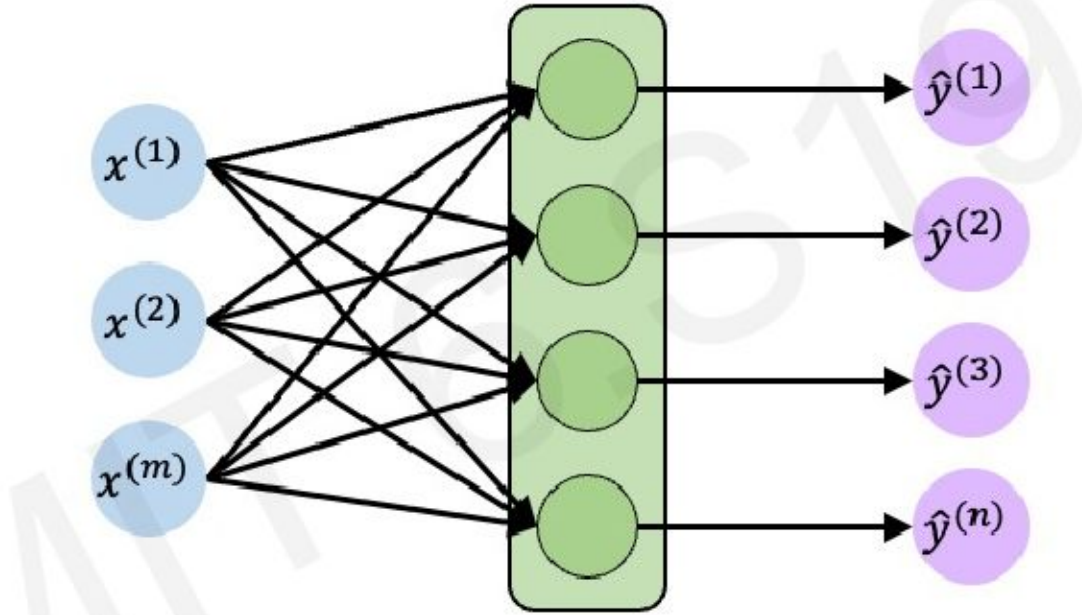


Non-linearities allow us to approximate arbitrarily complex functions

The Perceptron Revisited



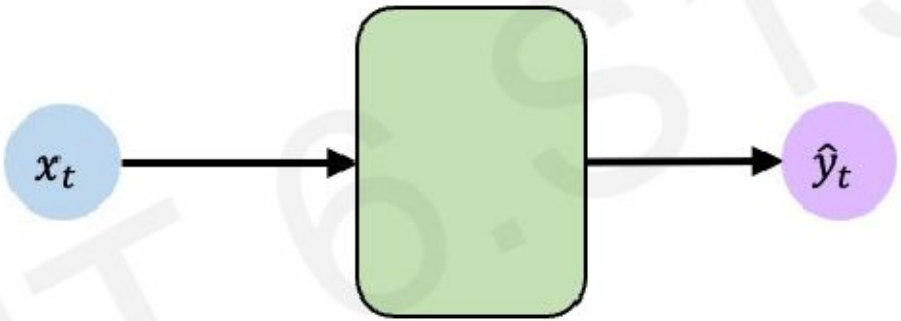
Feed-Forward Networks Revisited



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

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

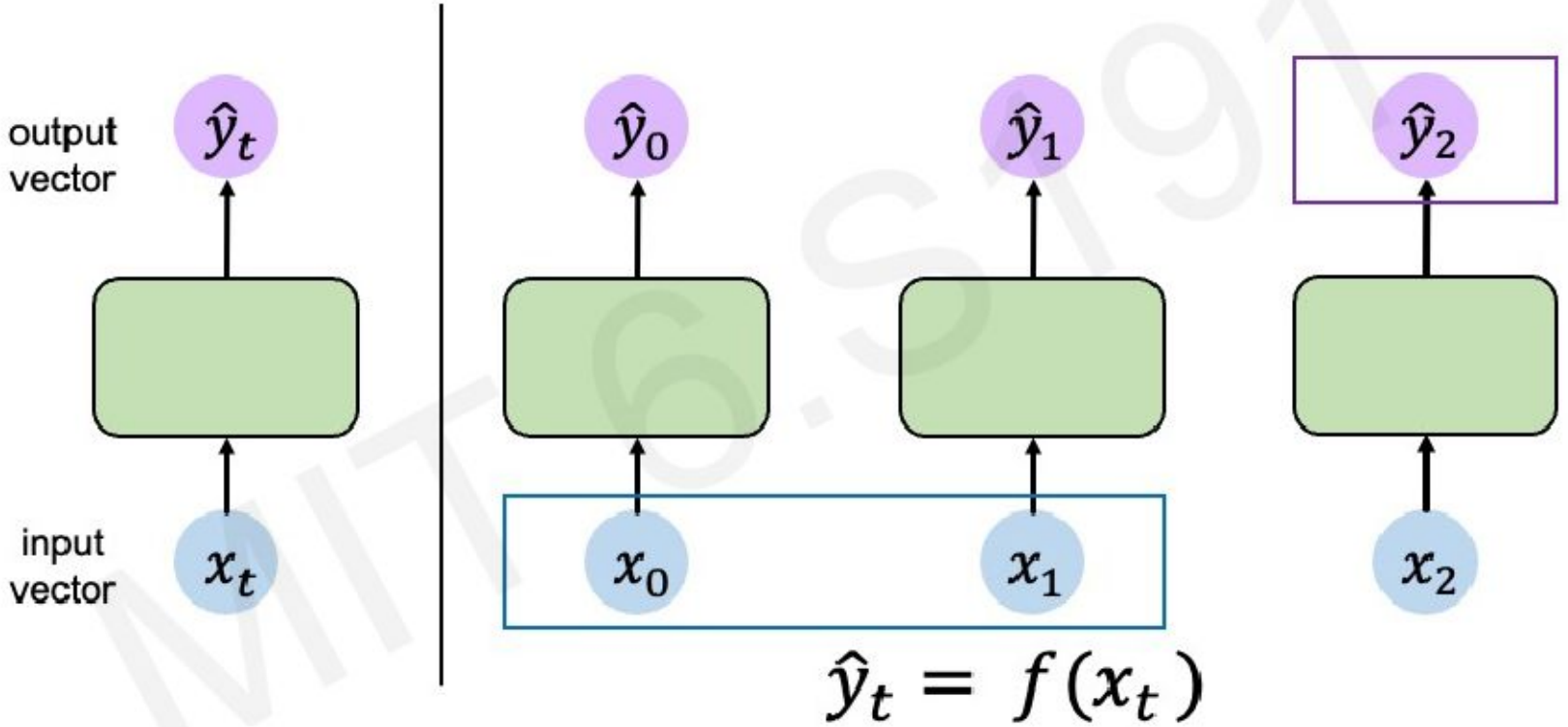
Feed-Forward Networks Revisited



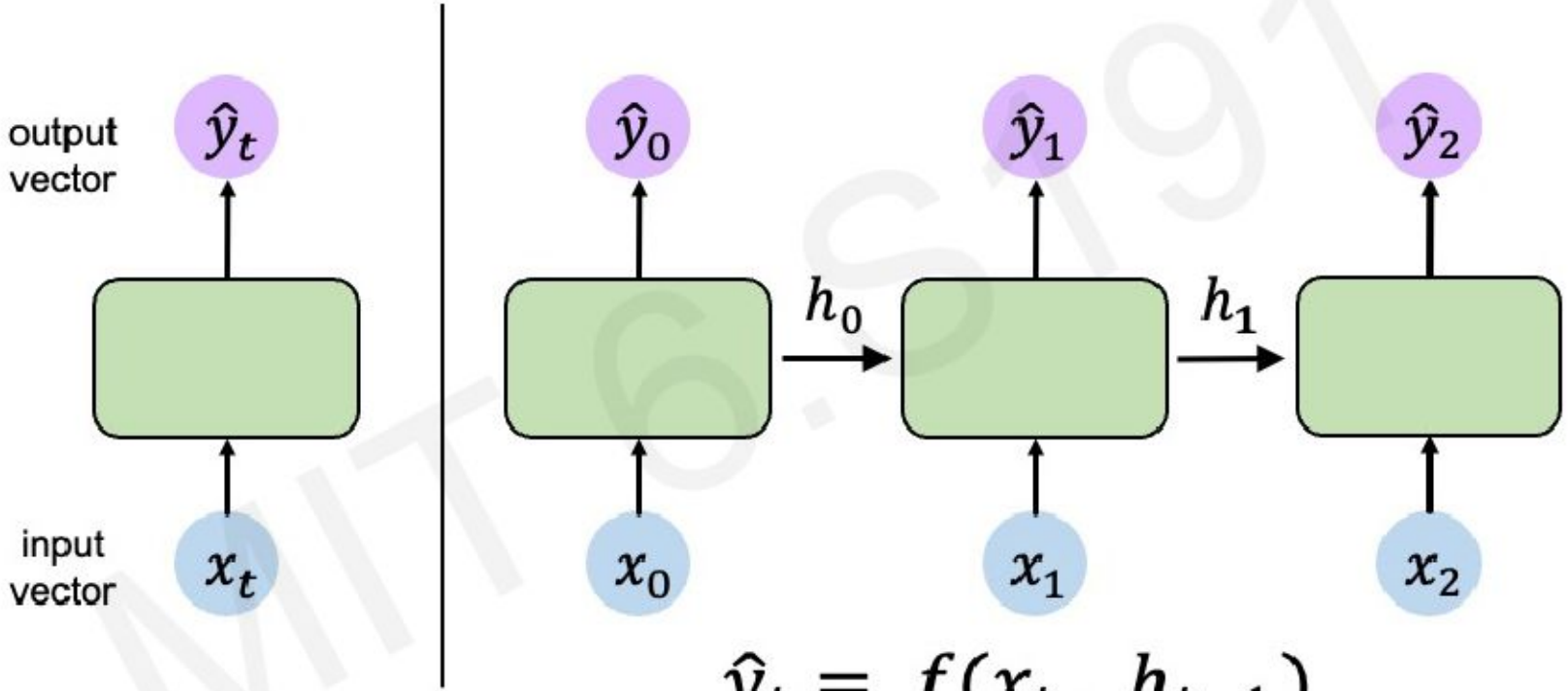
$$\boldsymbol{x_t} \in \mathbb{R}^m$$

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

Handling Individual Time Steps



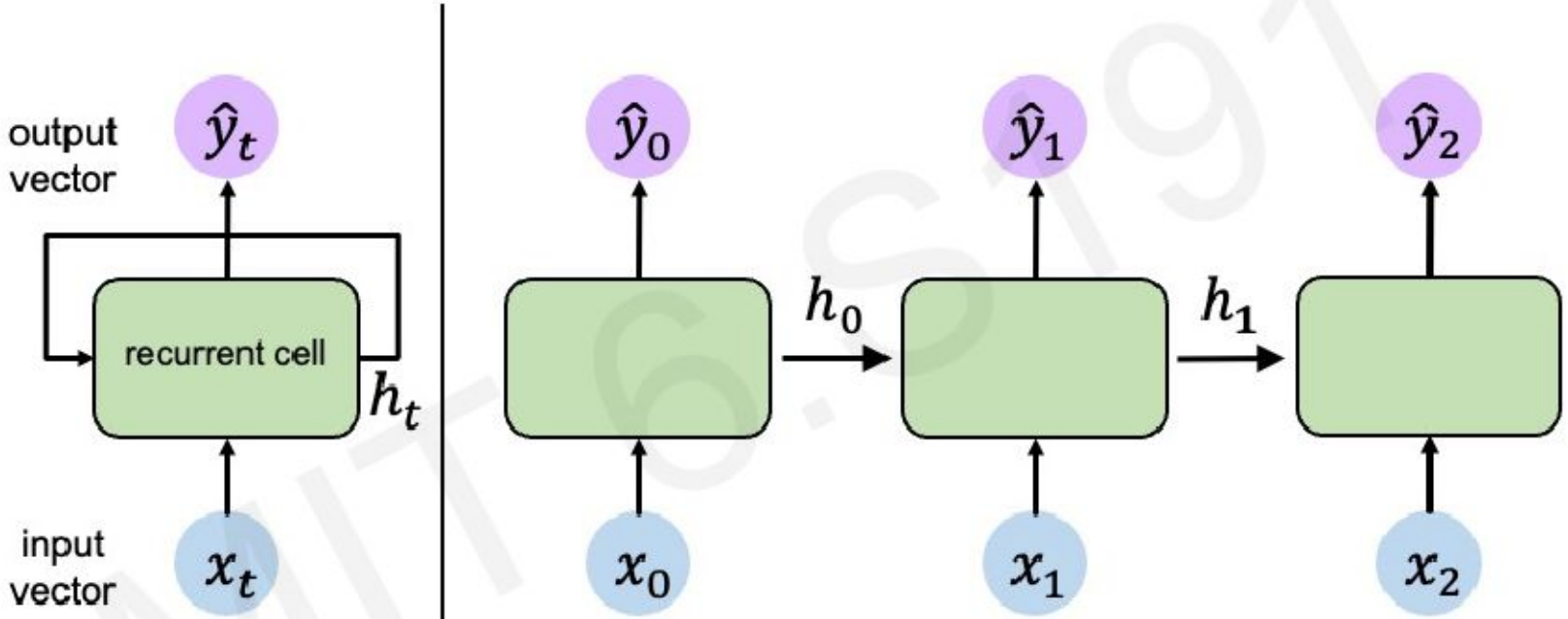
Neurons with Recurrence



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

output input past memory

Neurons with Recurrence

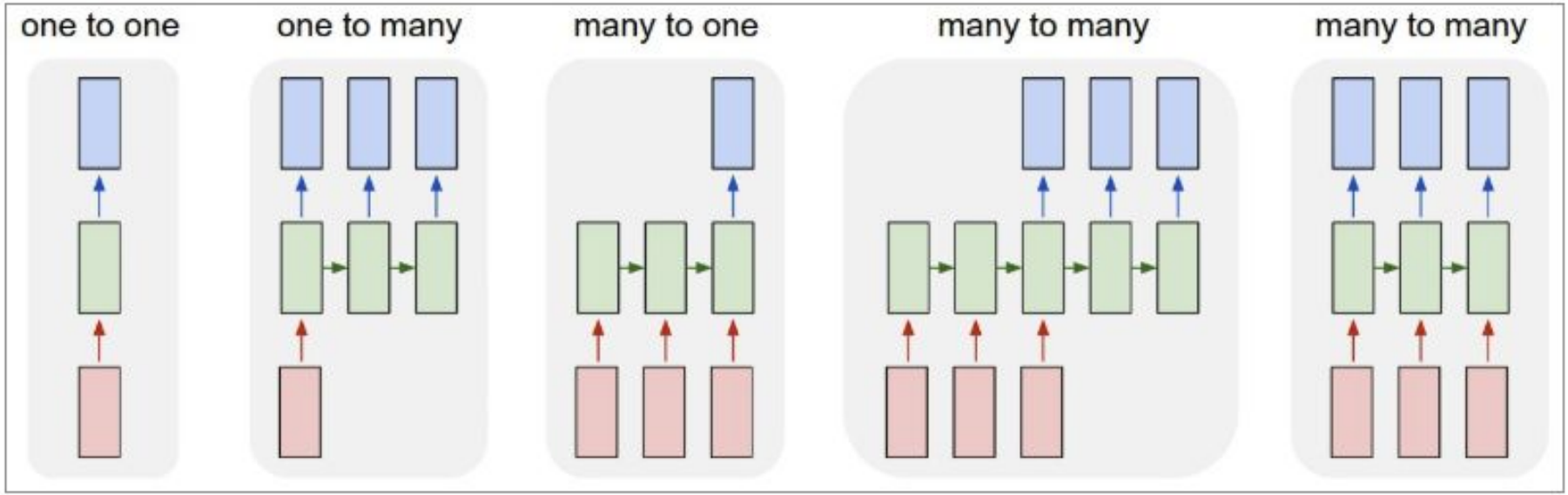


$$\hat{y}_t = f(x_t, h_{t-1})$$

output input past memory

Recurrent Neural Networks (RNNs)

Working with Sequences



(1) Vanilla model of processing without RNN.

(2) Sequence output (e.g., image captioning).

(3) Sequence input (e.g., sentiment analysis).

(4) Sequence input and output (e.g., Machine translation).

(5) Synced sequence input and output (e.g., Video classification, label each frame of the video).

The Architectural Innovation

The **key insight** behind **RNNs** is the introduction of a **hidden state h_t** that servers as the **network's memory**.

Rather than modeling $P(x_t \mid x_{t-1}, \ldots, x_{t-n+1})$, we use a latent variable model

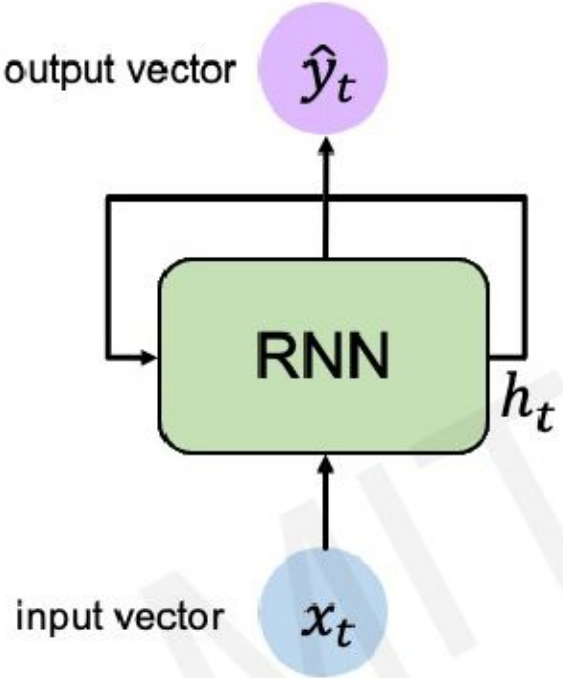
$$P(x_t \mid x_{t-1}, \ldots, x_{t-n+1}) \approx P(x_t \mid h_{t-1}),$$

where h_{t-1} is a *hidden state* that stores the sequence information up to time step $t-1$.

The hidden state at time step t could be computed based on both the current input x_t and the previous hidden state h_{t-1}

$$\mathbf{h_t = f(x_t, h_{t-1})}$$

Recurrent Neural Networks (RNNs)



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

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

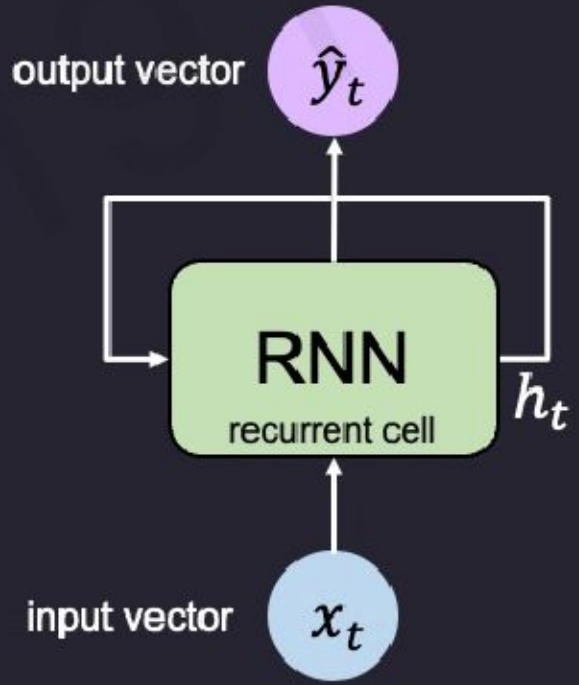
cell state | function with weights W | input | old state

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

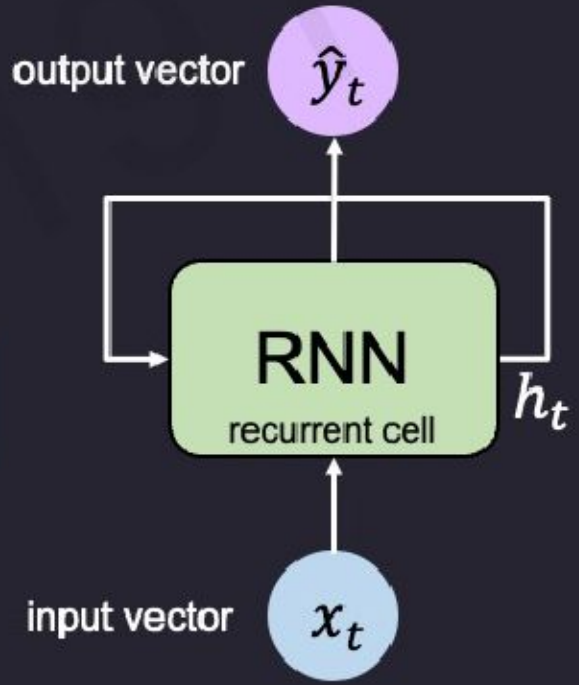
RNN Intuition

```
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!"
```



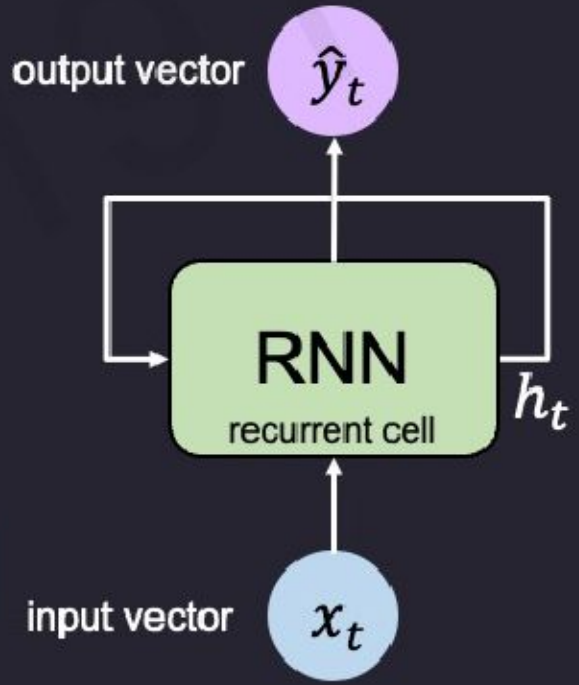
RNN Intuition

```
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!"
```

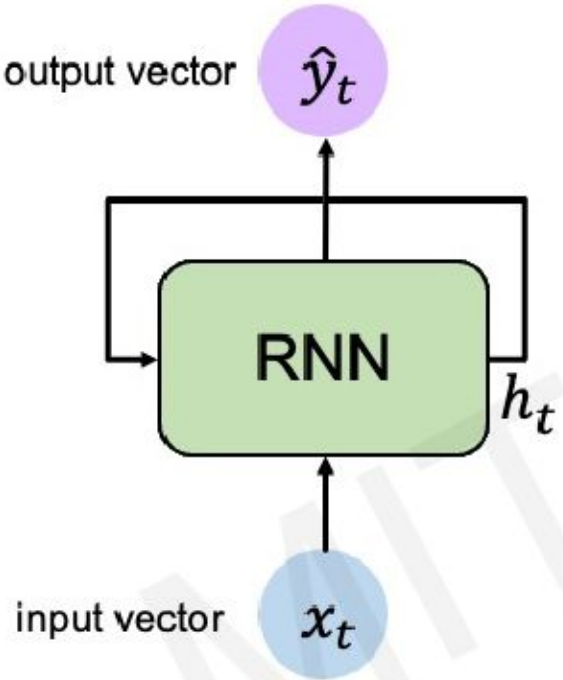


RNN Intuition

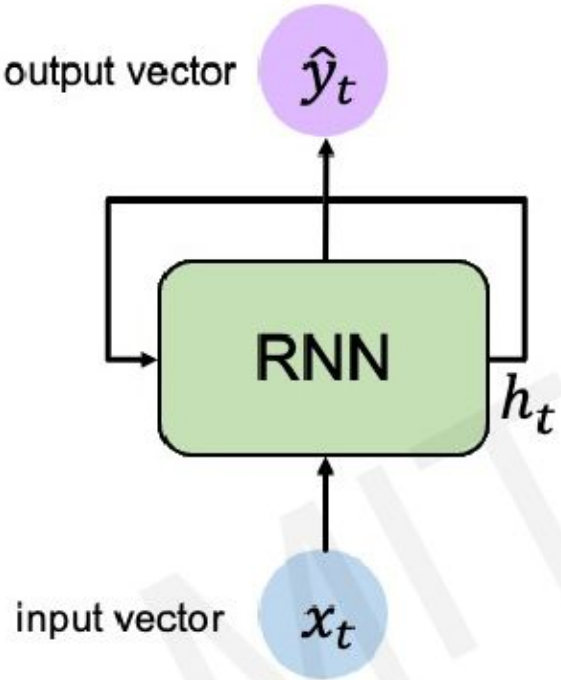
```
my_rnn = RNN()
hidden_state = [0, 0, 0, 0]

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



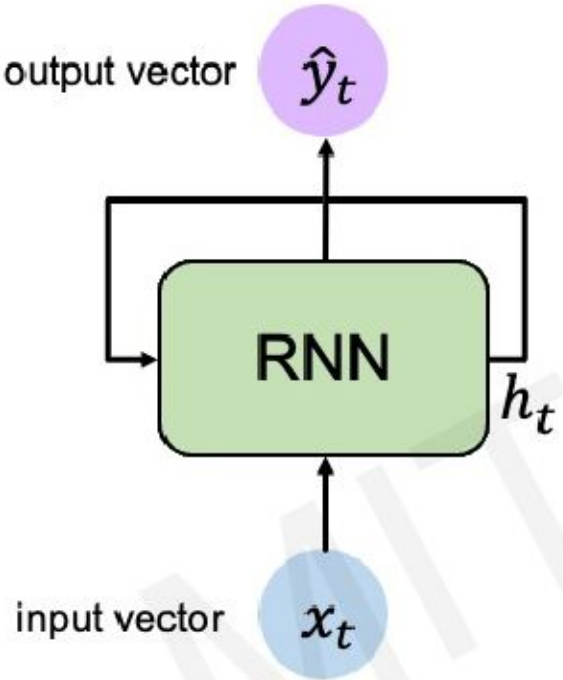
RNN State Update and Output



Input Vector

x_t

RNN State Update and Output



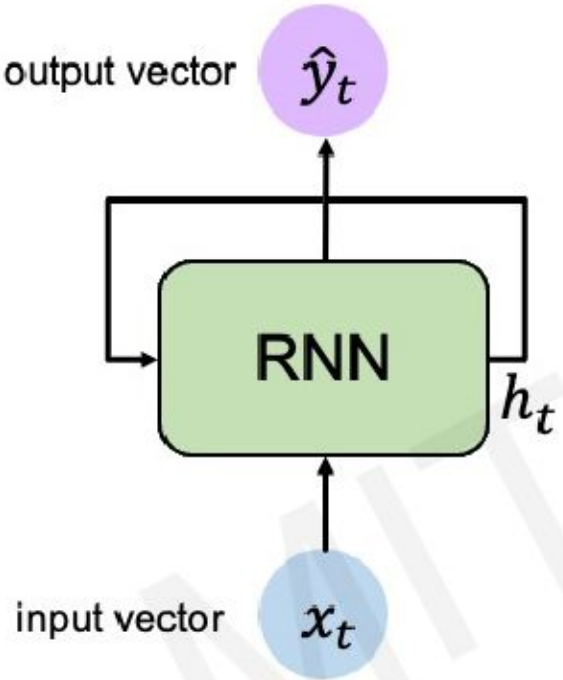
Update Hidden State

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

Input Vector

$$x_t$$

RNN State Update and Output



Output Vector

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

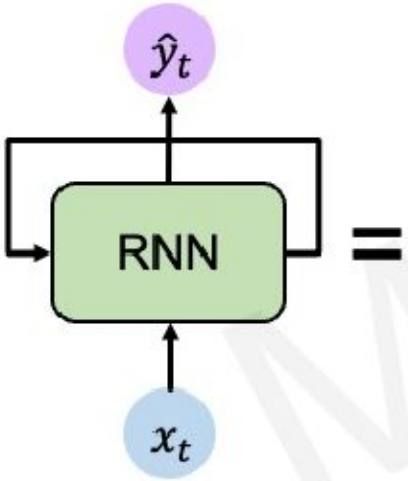
Update Hidden State

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

Input Vector

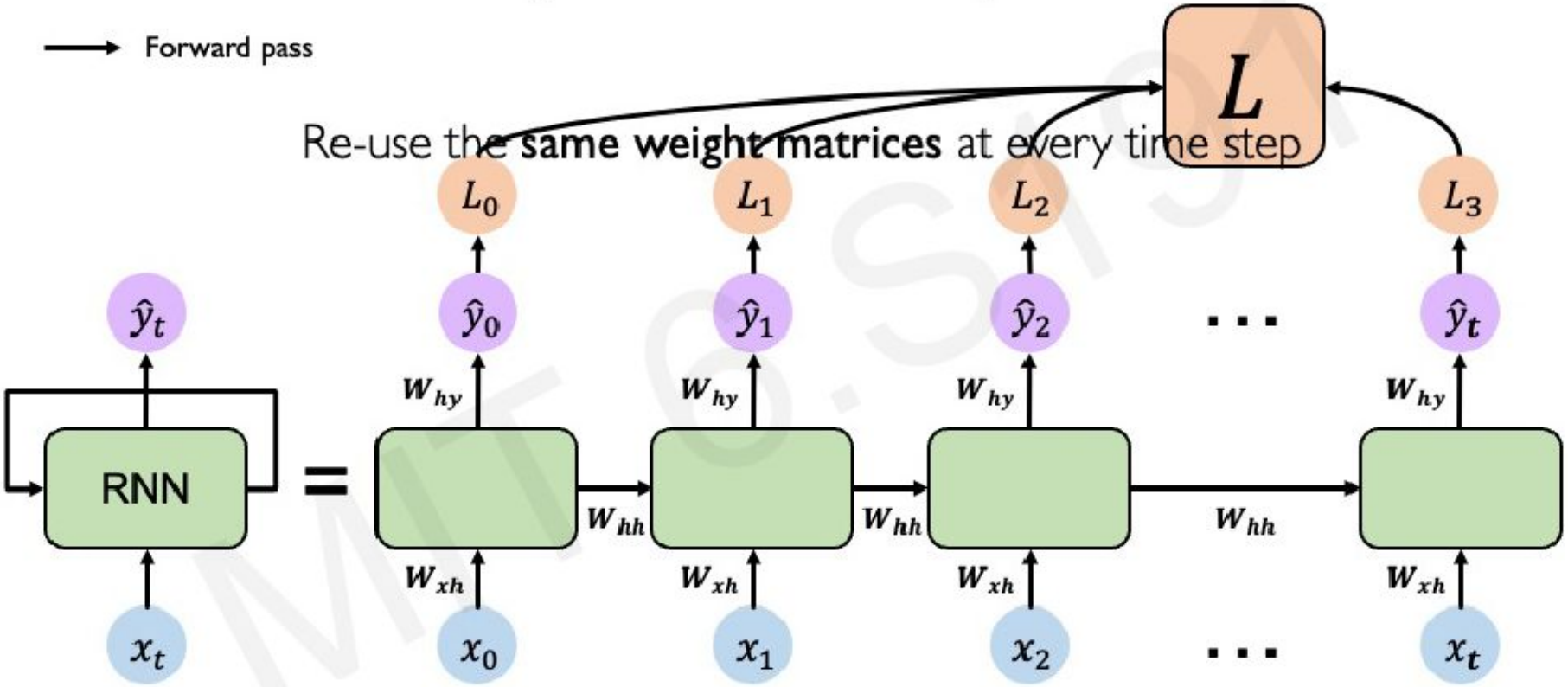
$$x_t$$

RNNs: Computational Graph Across Time



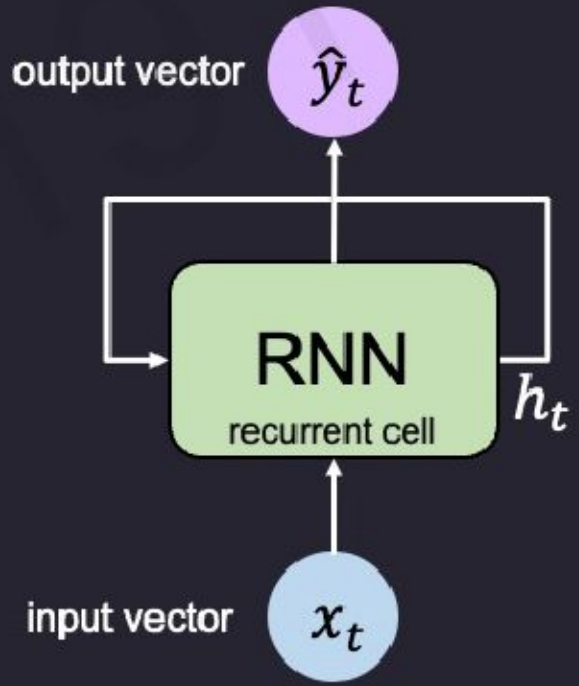
= Represent as computational graph unrolled across time

RNNs: Computational Graph Across Time



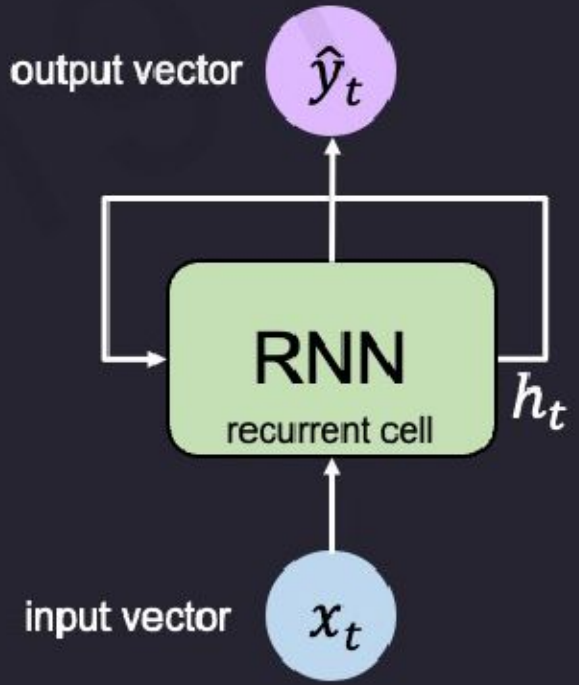
RNNs from Scratch

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

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

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

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

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

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



RNN Implementation in TensorFlow

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



A Sequence Modeling Problem: Predict the Next Word

A Sequence Modeling Problem: Predict the Next Word

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

A Sequence Modeling Problem: Predict the Next Word

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

given these words

A Sequence Modeling Problem: Predict the Next Word

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

given these words

predict the next word

A Sequence Modeling Problem: Predict the Next Word

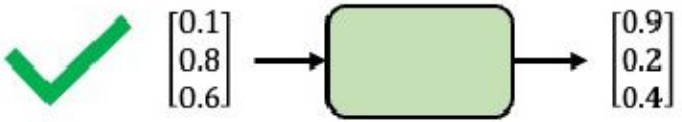
"This morning I took my cat for a walk."

given these words

predict the next word

Representing Language to a Neural Network

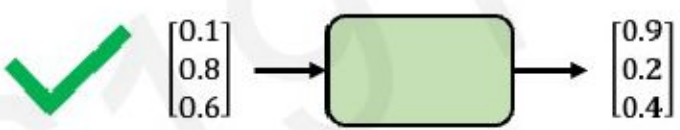
Neural networks cannot interpret words



Neural networks require numerical inputs

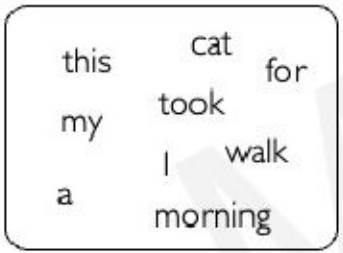
Encoding Language for a Neural Network

Neural networks cannot interpret words

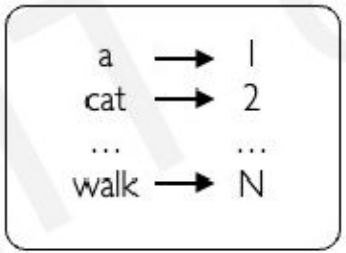


Neural networks require numerical inputs

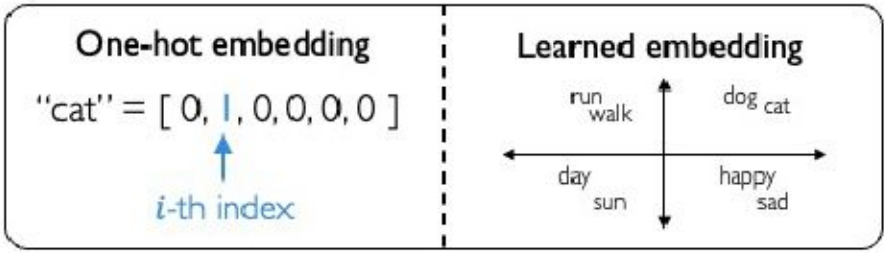
Embedding: transform indexes into a vector of fixed size.



1. Vocabulary:
Corpus of words



2. Indexing:
Word to index

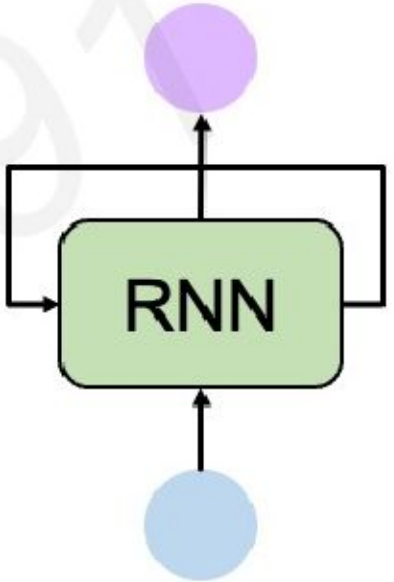


3. Embedding:
Index to fixed-sized vector

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

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.

Backpropagation Through Time (BPTT)

Loss Optimization

We want to find the network weights that ***achieve the lowest loss***

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\text{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\text{argmin}}\, J(\boldsymbol{W})$$

Loss Optimization

We want to find the network weights that ***achieve the lowest loss***

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\text{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\text{argmin}}\, J(\boldsymbol{W})$$

Remember:

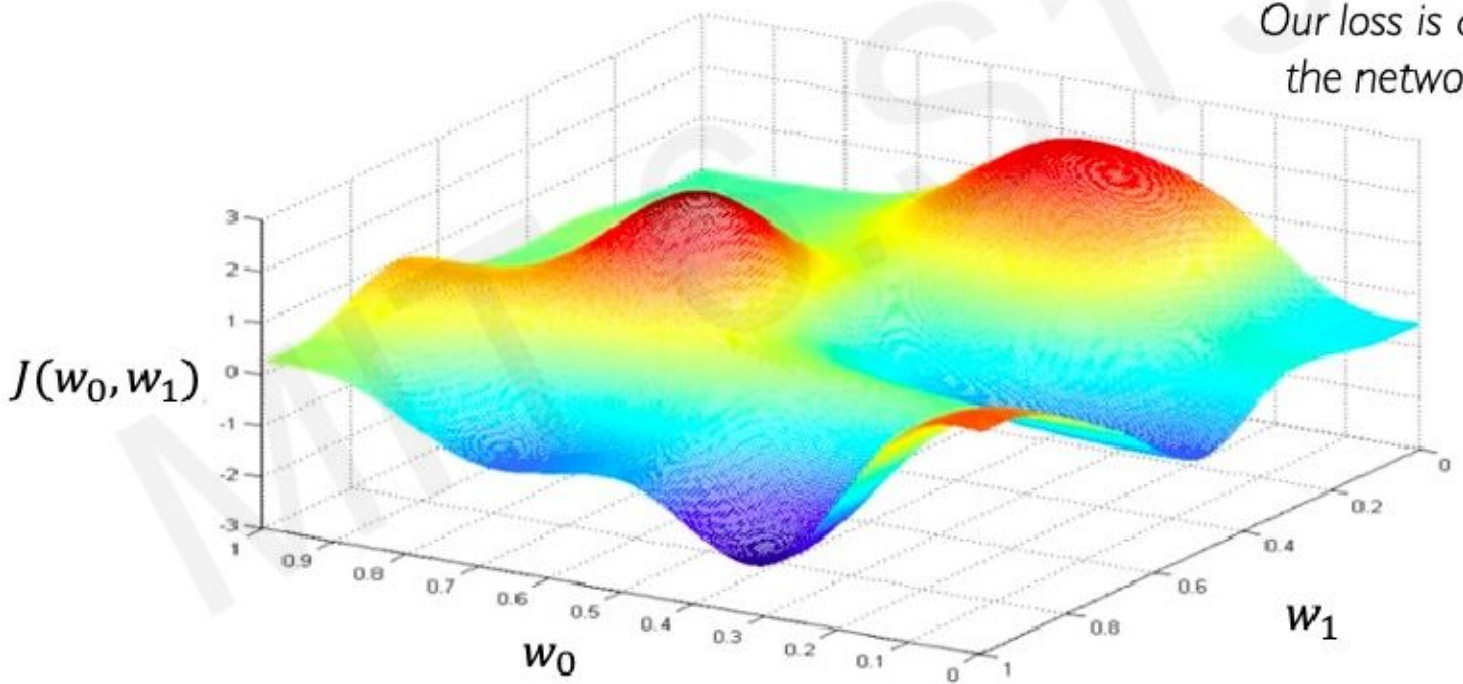
$$\boldsymbol{W} = \left\{\boldsymbol{W}^{(0)}, \boldsymbol{W}^{(1)}, \dots\right\}$$

Loss Optimization

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} J(\boldsymbol{W})$$

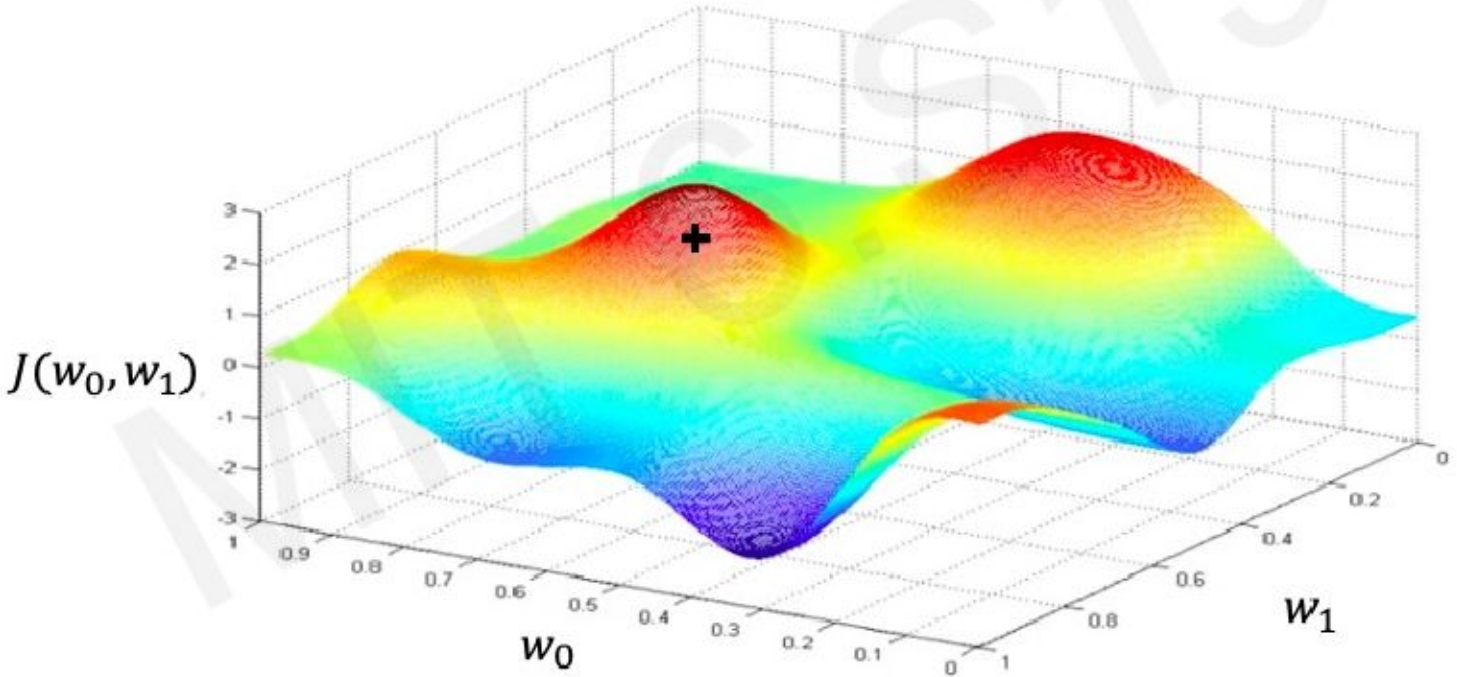
Remember:

Our loss is a function of the network weights!



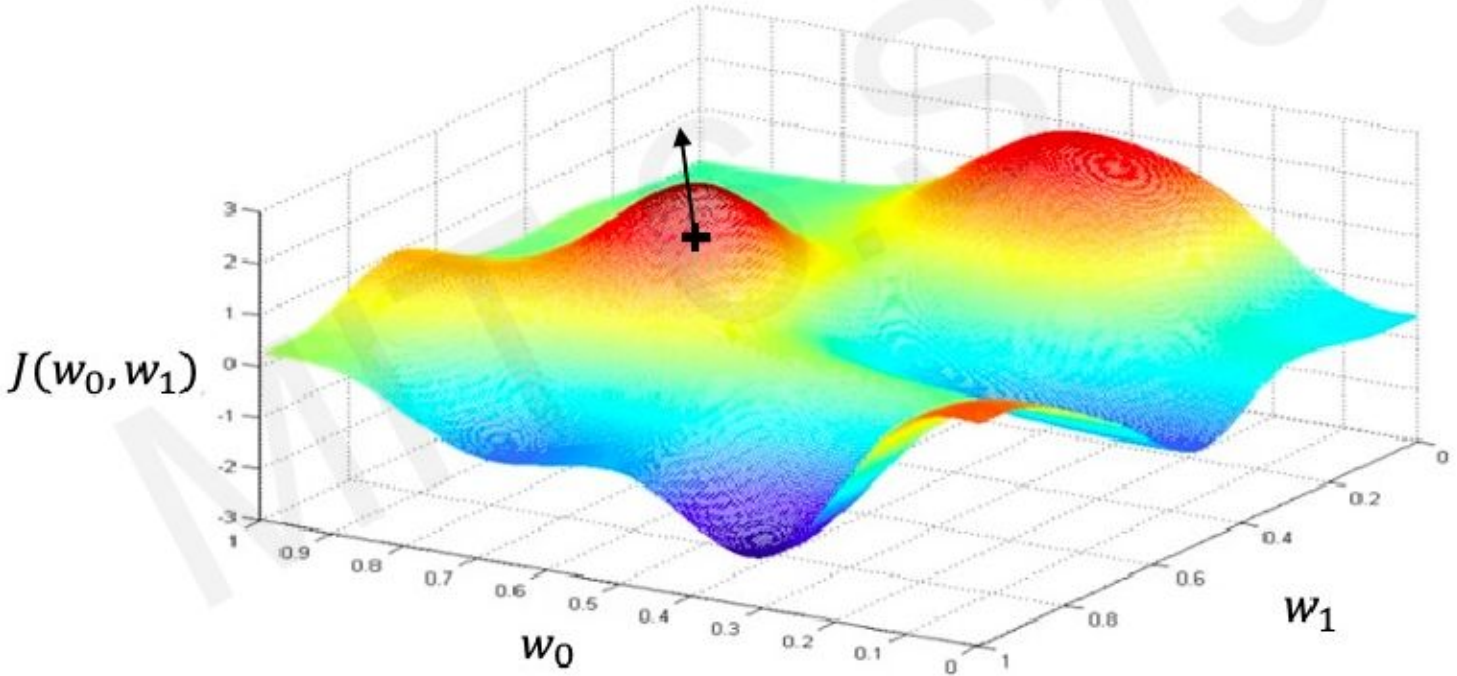
Loss Optimization

Randomly pick an initial (w_0, w_1)



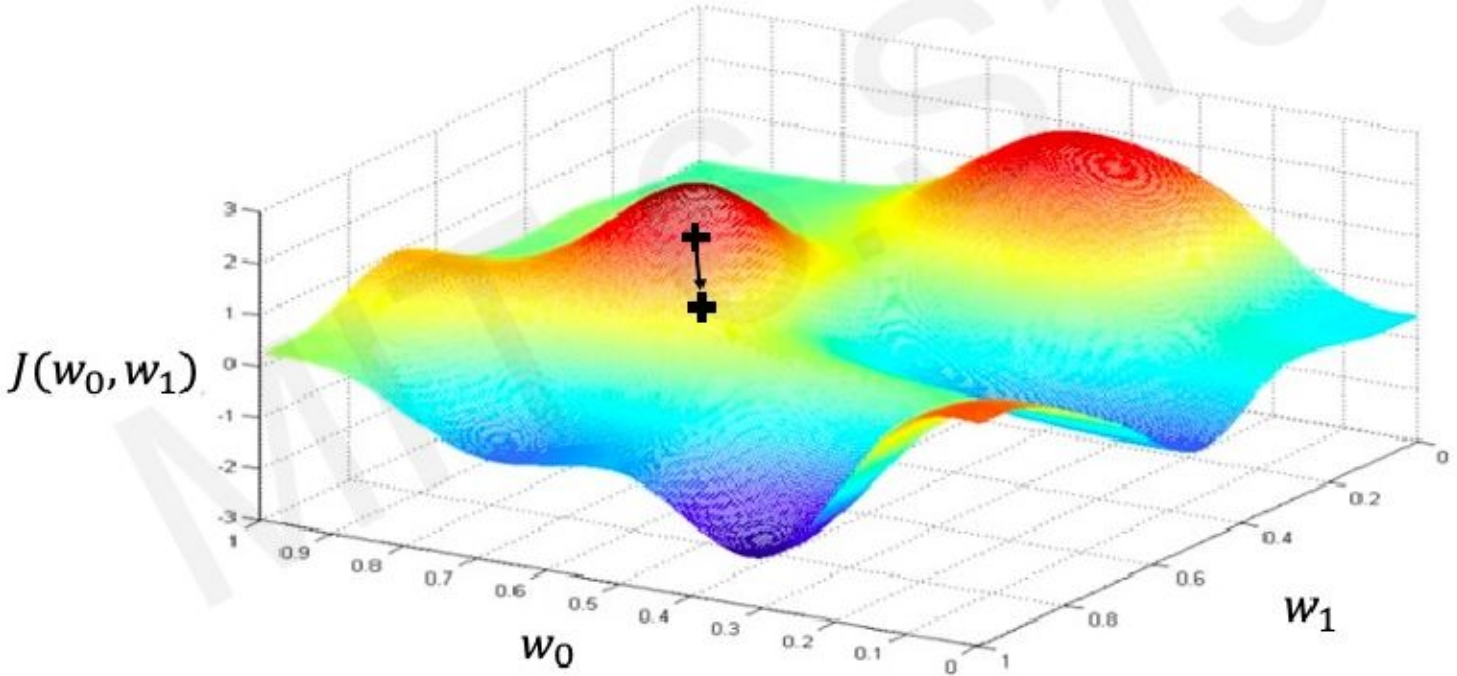
Loss Optimization

Compute gradient, $\frac{\partial J(W)}{\partial W}$



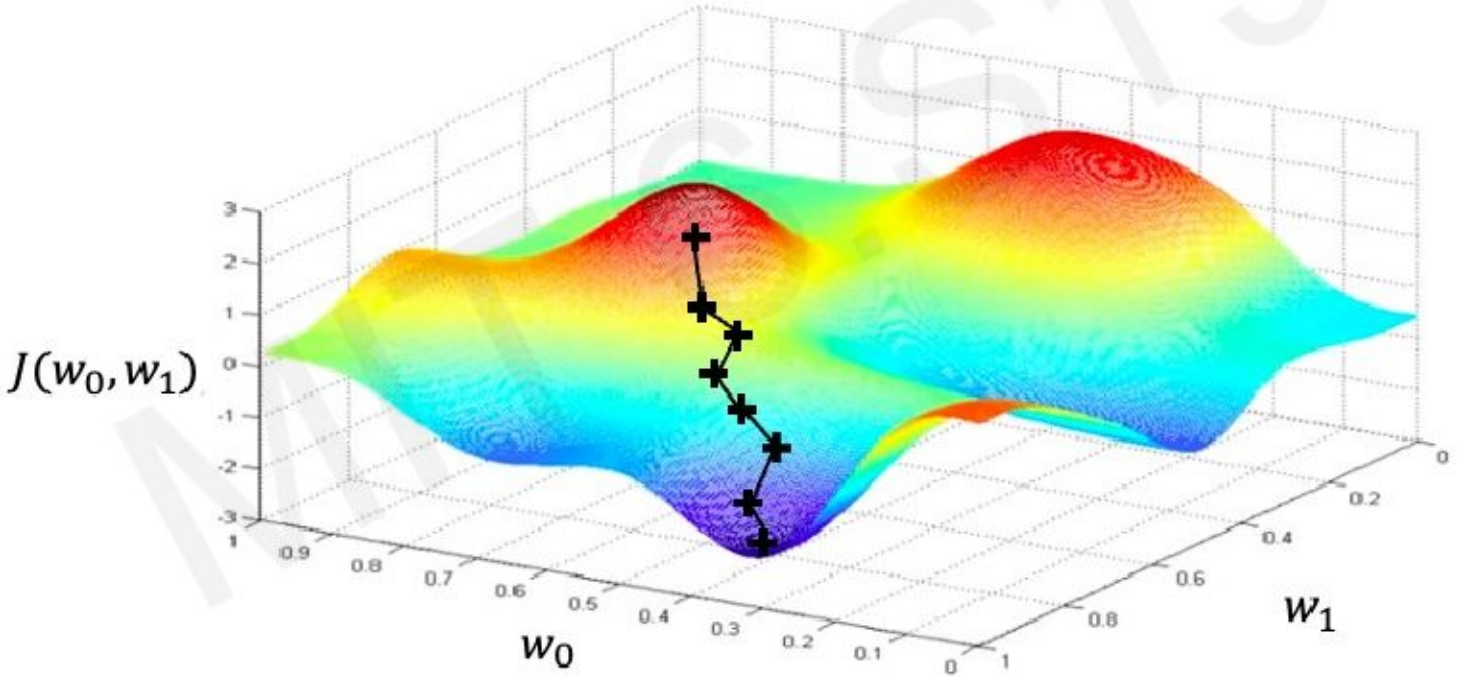
Loss Optimization

Take small step in opposite direction of gradient



Gradient Descent

Repeat until convergence

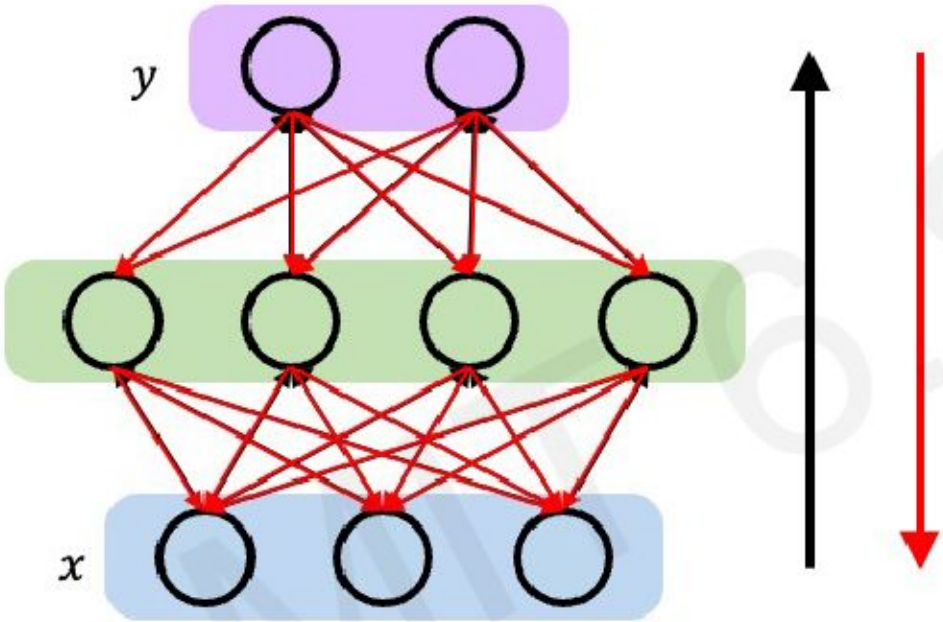


Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights

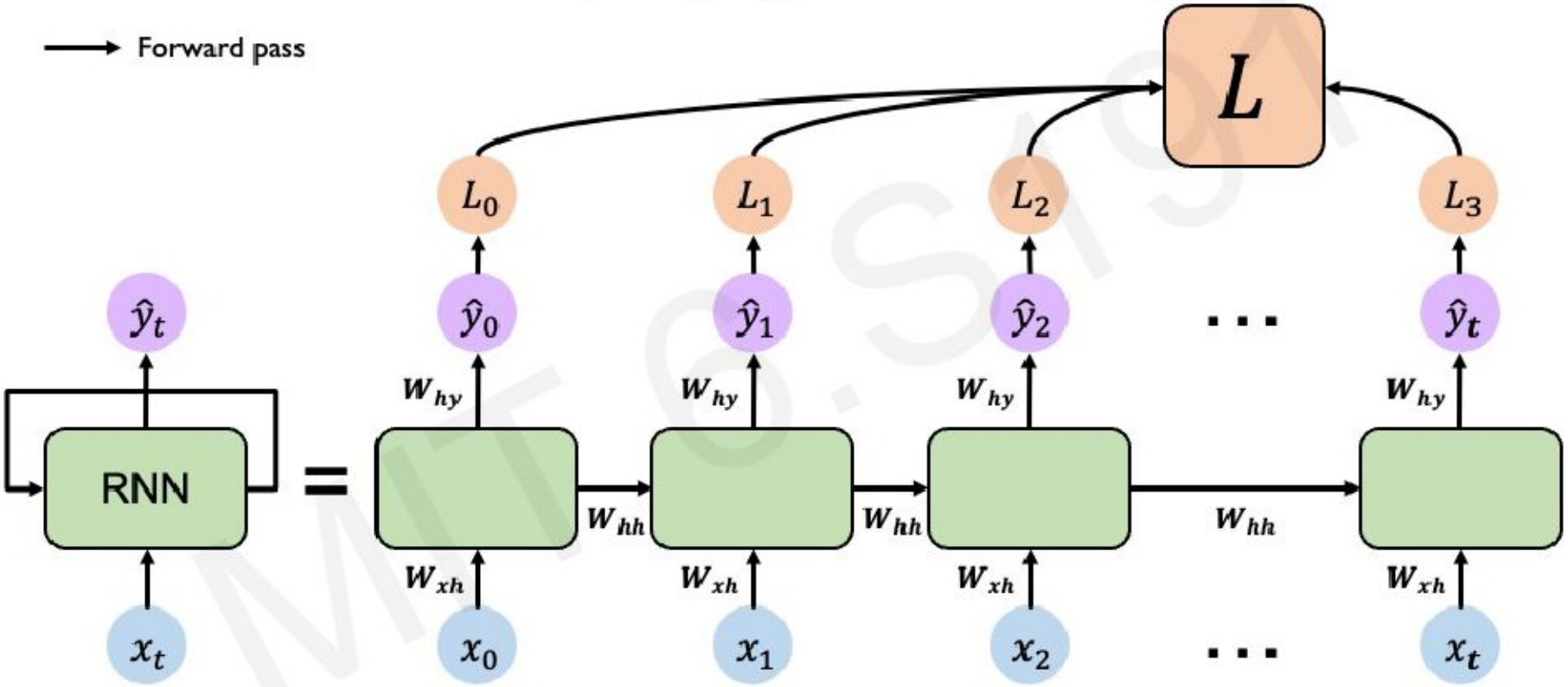
Recall: Backpropagation in Feed Forward Models



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



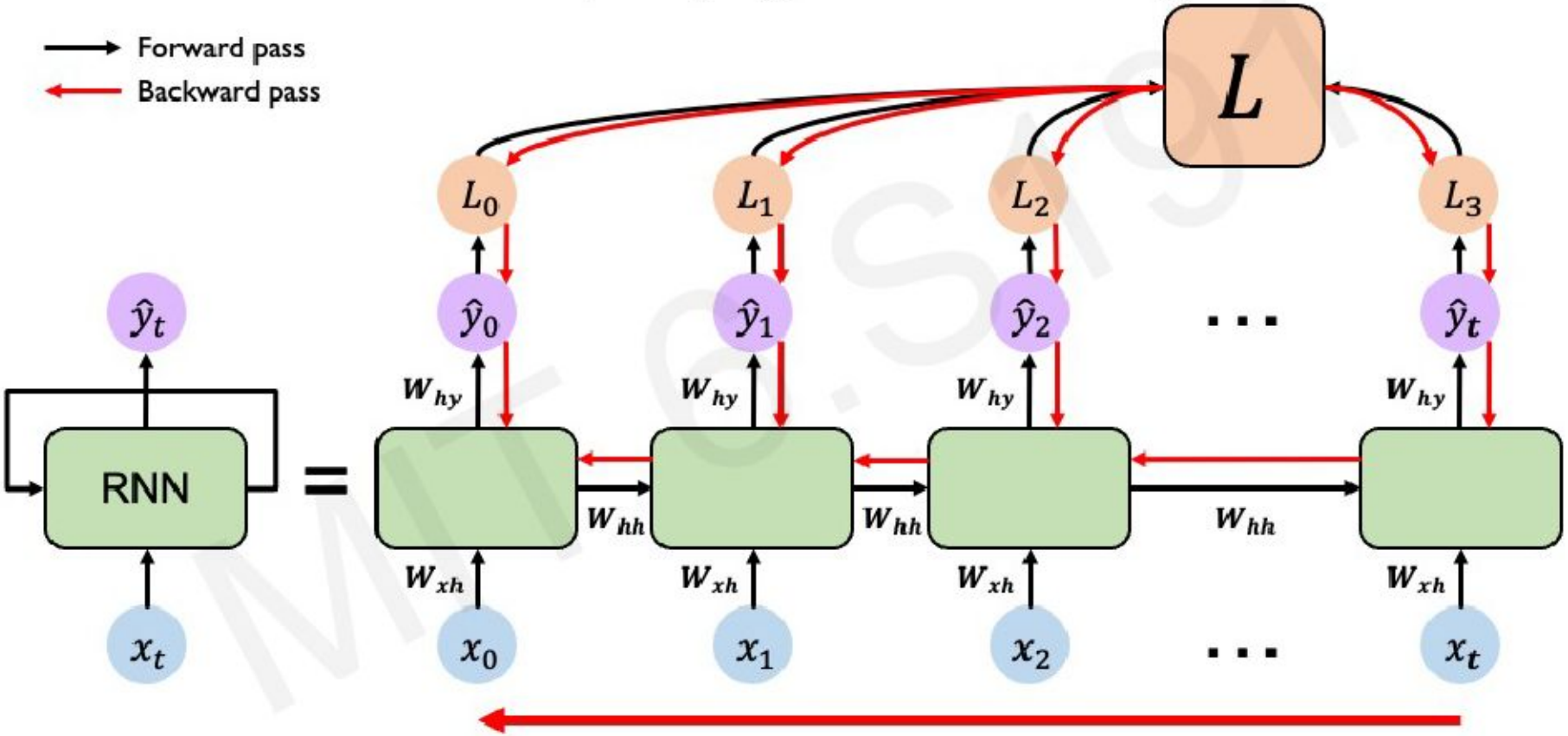
Forward Pass

The network **processes** the **input** sequence **one step at a time** from **start** to **finish**

At each time step, it **calculates** a **prediction** and a **hidden state**.

The prediction across all time steps are compared to the actual target values to **calculate a total error (loss) for the entire sequence**.

RNNs: Backpropagation Through Time



Backward Pass ("Through Time")

This is where the **"through time"** part **happens**.

The **total error** is **propagated backward** through the **unrolled network**, **starting** from the **last** time **step** and going all the **way back** to the **first**.

> At **each time step**, the algorithm **calculates** the **gradient** of the error with respect to the network's weights ***at that specific step***.

Machine Translation

Consider the task of **translating one word into two** (or any sequence into another of a different length)

I.e., **inputs** and **outputs** are of **varying lengths** that are **unaligned**.

Solution: design a **encoder-decoder architecture**.

Encoder ("reader"): takes a variable-length sequence as input.

Decoder ("writer"): acts as a conditional language model.

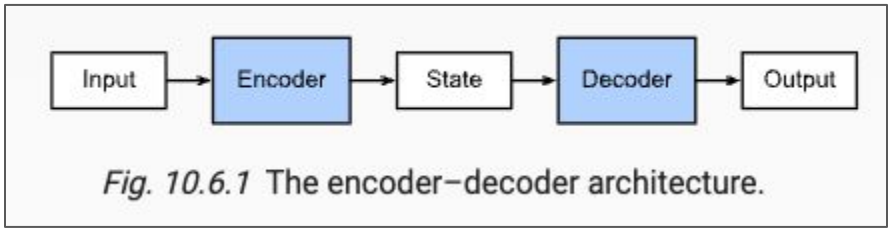


Fig. 10.6.1 The encoder–decoder architecture.

E.g., "cat" → "o gato"

Encoder: Reads "cat" and produces *context_vector/final hidden state*

Decoder (step 1): Takes *context_vector* and a <*start*> token as input. It predicts the word "o".

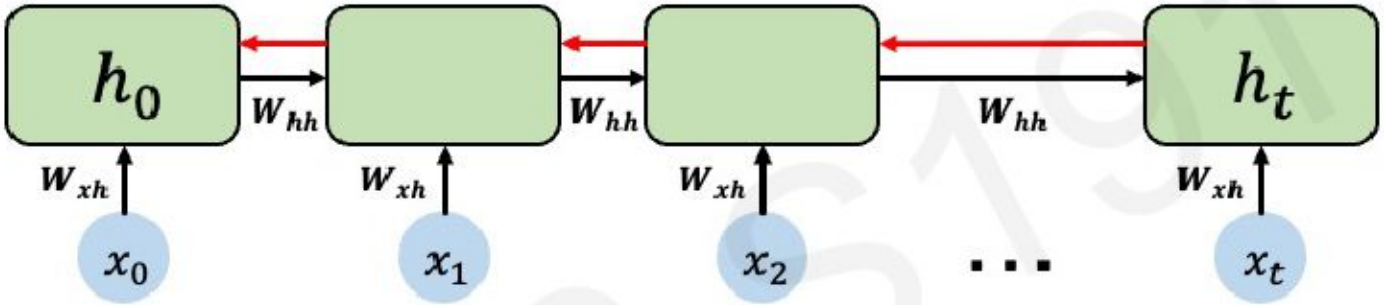
Decoder (step 2): Takes its own memory and the word "o" as input. It predicts the word "gato".

Decoder (step 3): Takes its memory and the word "gato" as input. It predicts and <*end*> token.

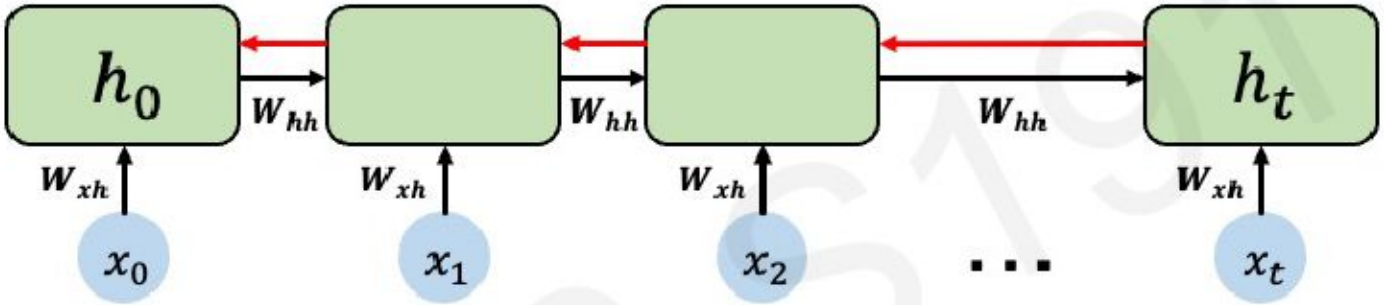
Final output: The model assembles the prediction: "o gato"

RNN tricks

Standard RNN Gradient Flow

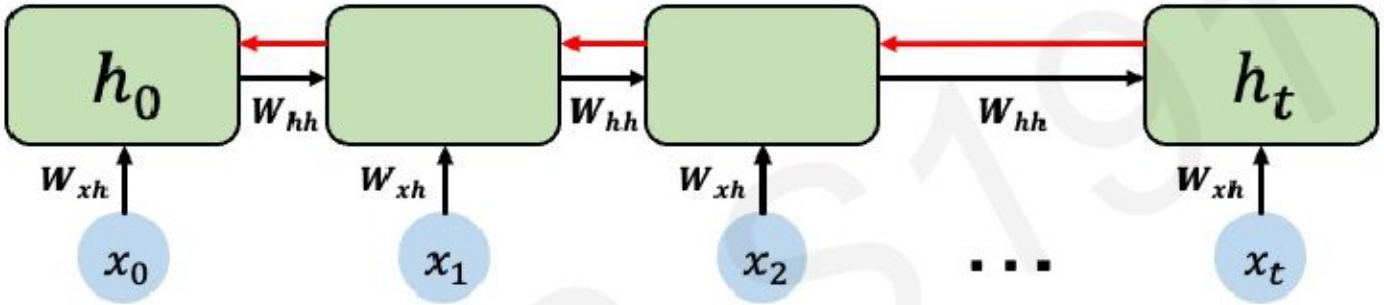


Standard RNN Gradient Flow



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

Standard RNN Gradient Flow: Exploding Gradients

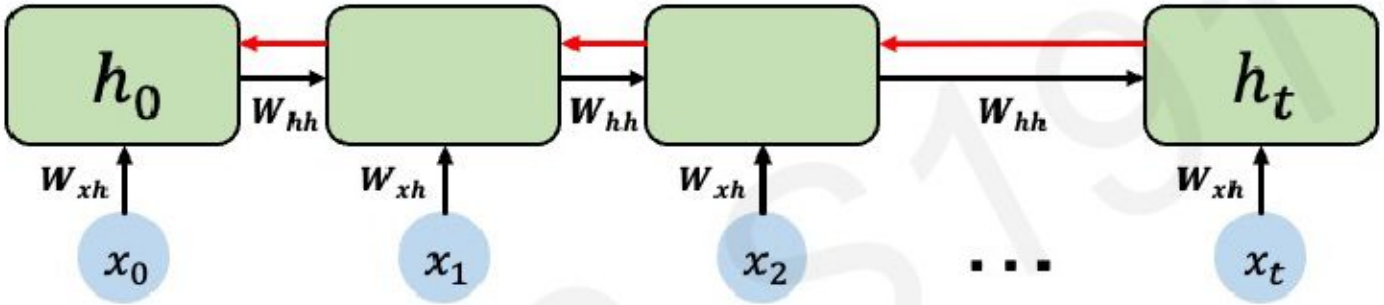


Computing the gradient wrt h_0 involves **many factors of W_{hh} + repeated gradient computation!**

Many values > 1:
exploding gradients

Gradient clipping to scale big gradients

Standard RNN Gradient Flow: Vanishing Gradients



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

Many values > 1:
exploding gradients

Gradient clipping to scale big gradients

Many values < 1:
vanishing gradients

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

The Problem of Long-Term Dependencies

Why are vanishing gradients a problem?

Multiply many **small numbers** together

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

Bias parameters to capture short-term dependencies

The Problem of Long-Term Dependencies

"The clouds are in the ___"

Why are vanishing gradients a problem?

Multiply many **small numbers** together

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

Bias parameters to capture short-term dependencies

The Problem of Long-Term Dependencies

Why are vanishing gradients a problem?

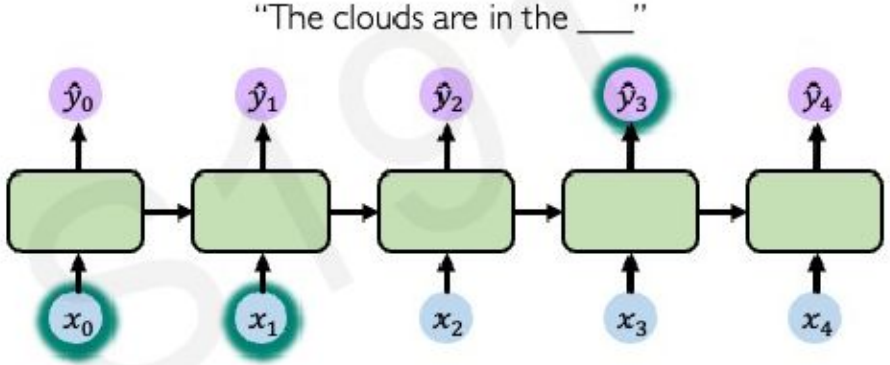
Multiply many **small numbers** together

↓

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

↓

Bias parameters to capture short-term dependencies



The Problem of Long-Term Dependencies

Why are vanishing gradients a problem?

Multiply many **small numbers** together

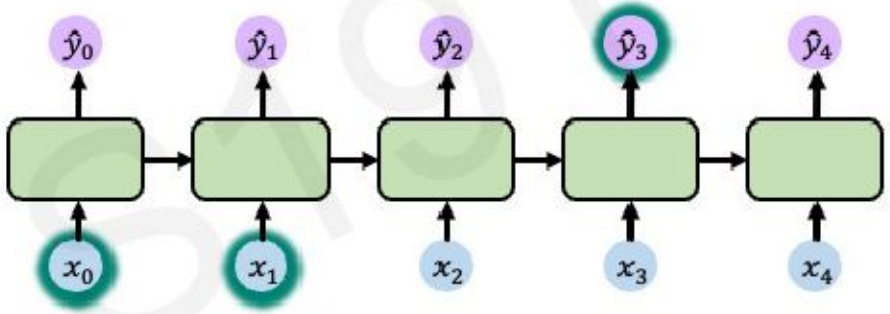
↓

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

↓

Bias parameters to capture short-term dependencies

"The clouds are in the ___"



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

The Problem of Long-Term Dependencies

Why are vanishing gradients a problem?

Multiply many **small numbers** together

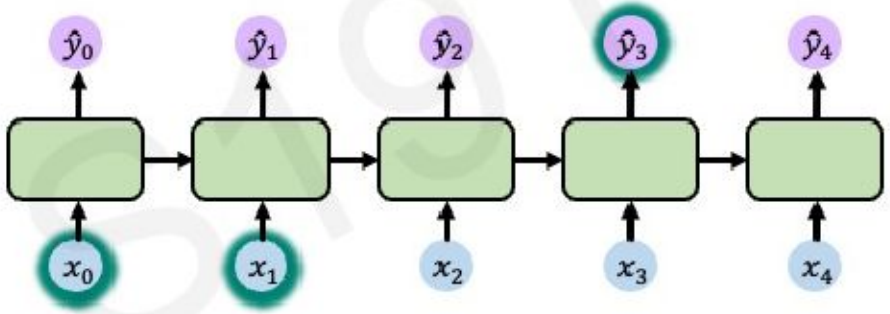
↓

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

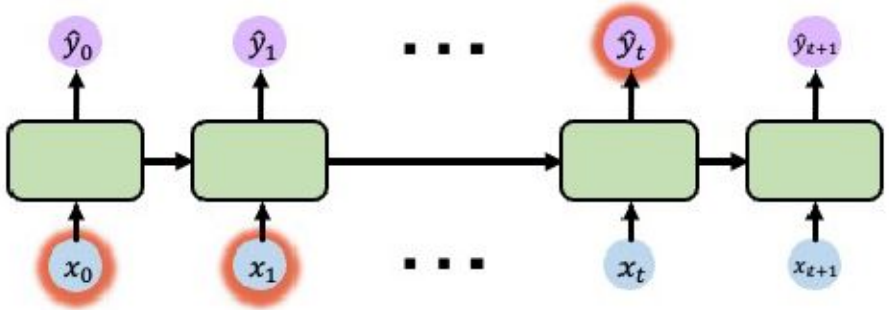
↓

Bias parameters to capture short-term dependencies

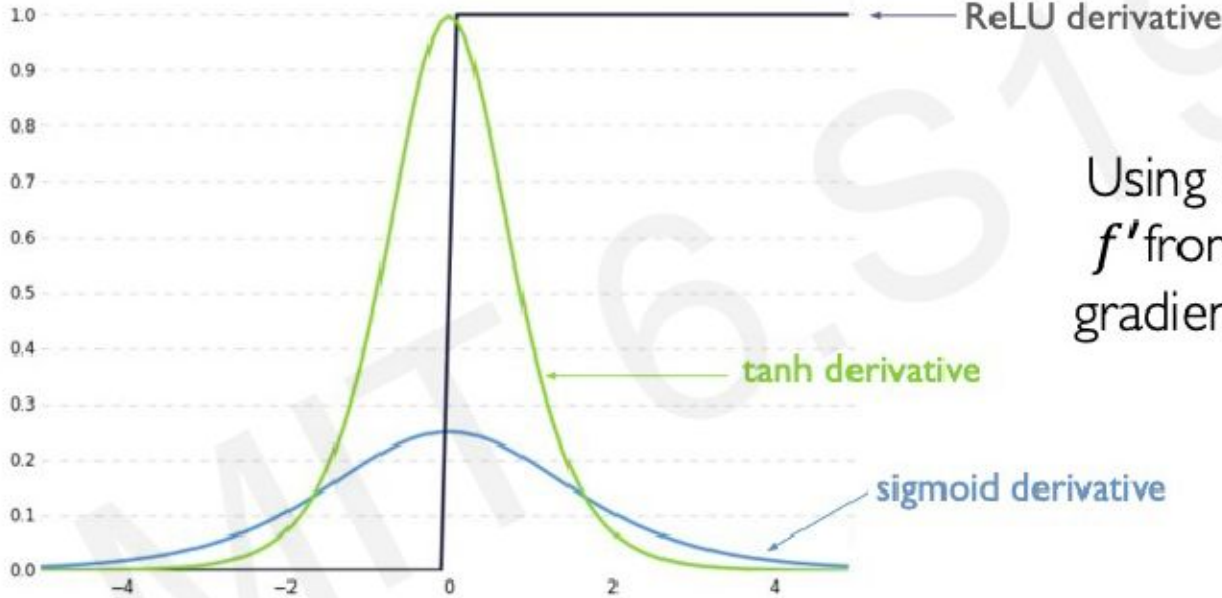
"The clouds are in the ___"



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



Trick #1: Activation Functions



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

Trick #2: Parameter Initialization

Initialize **weights** to identity matrix

Initialize **biases** to zero

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

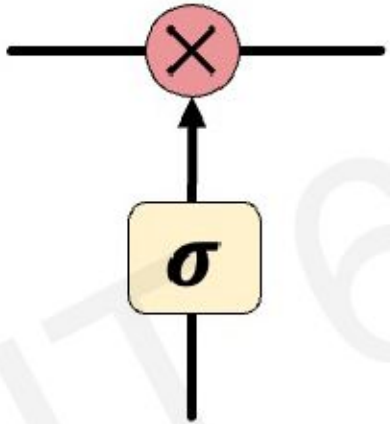
This helps prevent the weights from shrinking to zero.

Trick #3: Gated Cells

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

Pointwise multiplication

Sigmoid neural net layer



gated cell

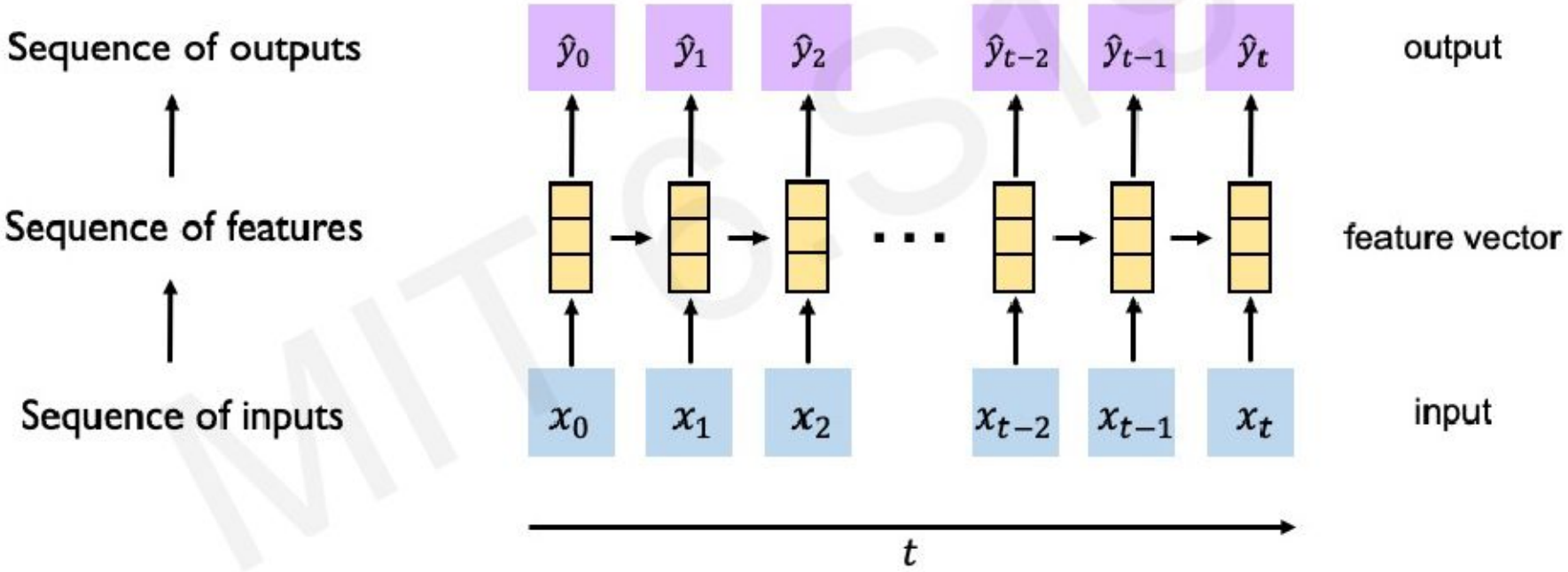
LSTM, GRU, etc.

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.

Goal of Sequence Modeling

RNNs: recurrence to model sequence dependencies

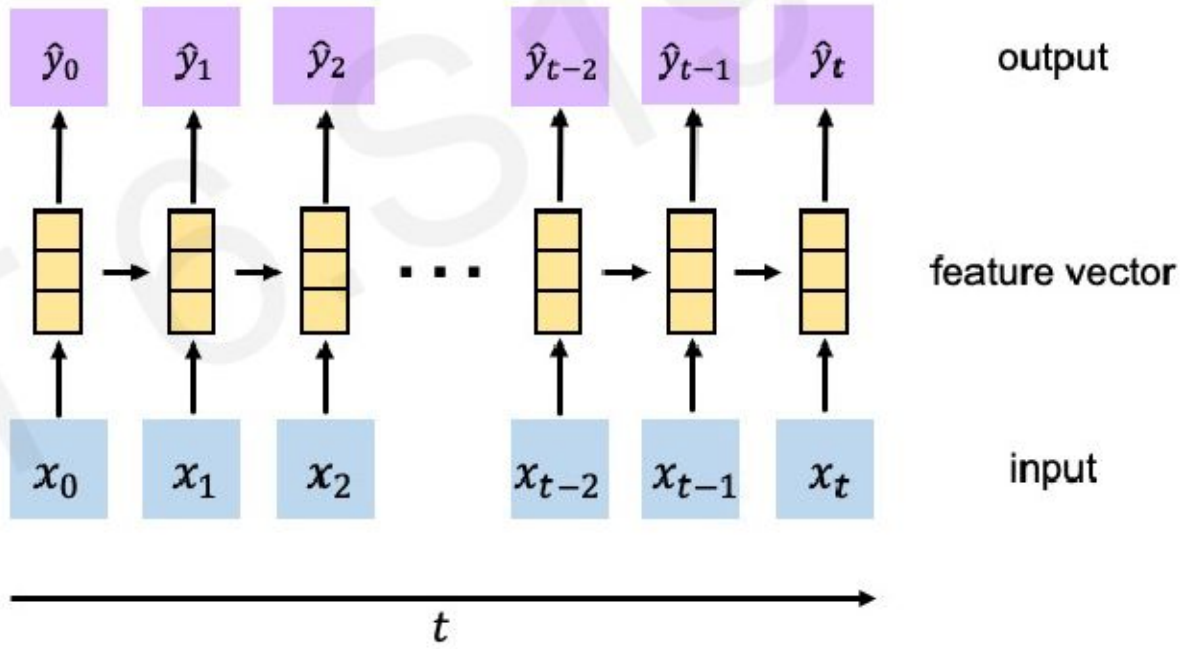


Goal of Sequence Modeling

RNNs: recurrence to model sequence dependencies

Limitations of RNNs

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

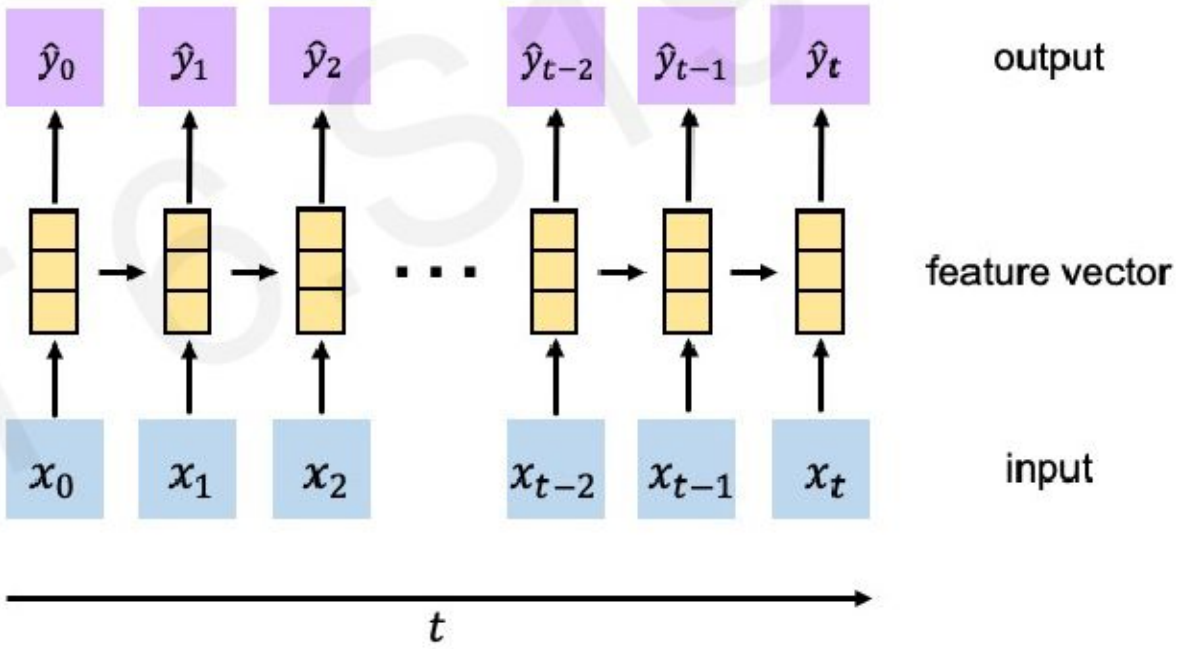


Goal of Sequence Modeling

Can we eliminate the need for recurrence entirely?

Desired Capabilities

- Continuous stream
- Parallelization
- Long memory

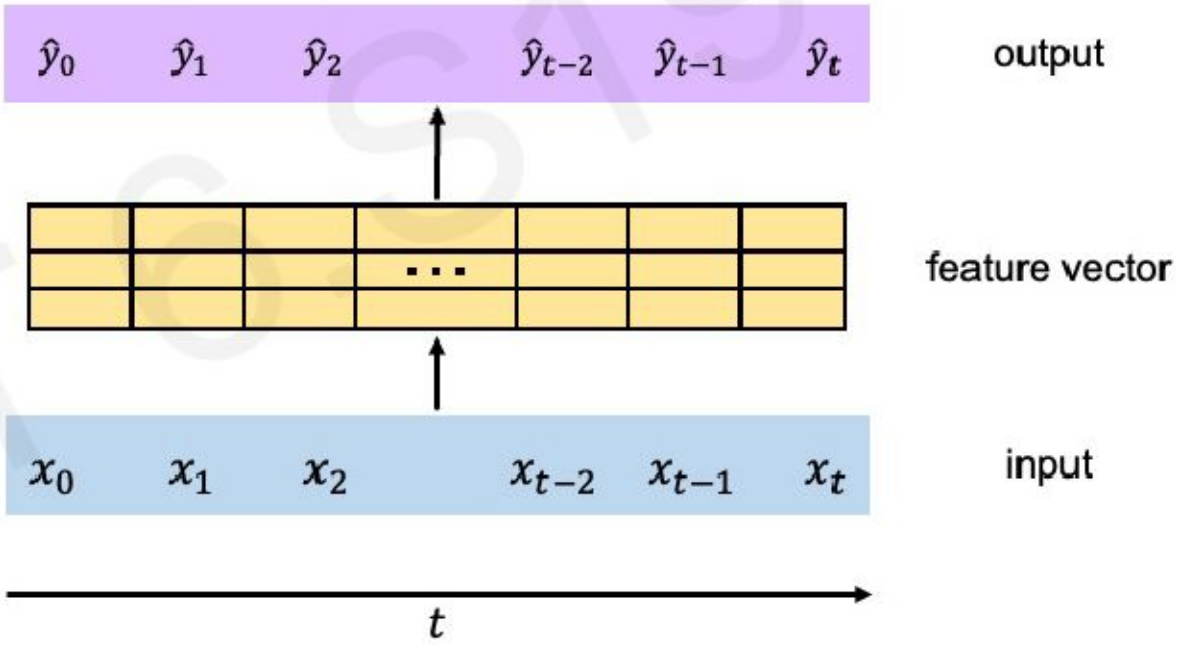


Goal of Sequence Modeling

Can we eliminate the need for recurrence entirely?

Desired Capabilities

- Continuous stream
- Parallelization
- Long memory



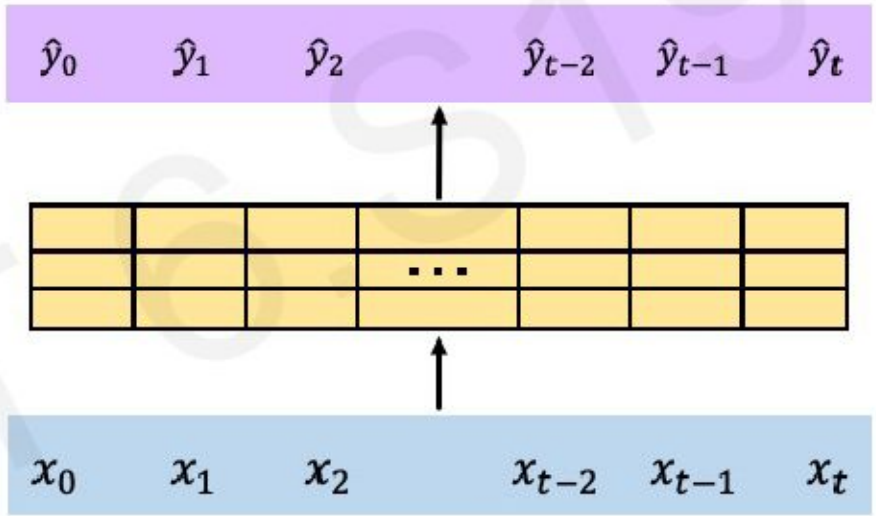
Goal of Sequence Modeling

Idea 1: Feed everything into dense network

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

Idea: Identify and attend to what's important

Can we eliminate the need for recurrence entirely?



Applications

Language modeling

RNNs could generate coherent text by predicting the next word in a seq.

They struggle with maintaining coherence over longer passages due to the vanishing gradient problem.

Speech Recognition

RNNs proved effective for processing audio sequences, where local dependencies are often more important than very long-range dependencies.

The Legacy of RNNs

Sequential Processing Paradigm – RNNs established the paradigm of processing sequences step-by-step while maintaining state across time.

Weight Sharing – The concept of sharing parameters across time steps became a fundamental principle in sequence modeling.

Hidden State Representation – The idea of maintaining compressed representation of sequence history influenced all subsequent architectures.

Recurrent Connections – While modern architectures like Transformers avoid recurrence, the concept of allowing information to flow between different positions in a sequence remains central to sequence modeling.

UFMG UNIVERSIDADE FEDERAL DE MINAS GERAIS

Coming next…

Attention Mechanisms

Rodrygo L. T. Santos
rodrygo@dcc.ufmg.br