Summer School – Speaking Data: dato e comunicazione nell'era dell'Intelligenza Artificiale Generativa

Architettura dell'AI Generativa

Mauro Bruno (Istat)

May 27, 2025

Introduction to Artificial Neural Networks

1 Context

- Context
- Neural Networks (core concepts)
- Recurrent Neural Networks
- Language Models

What are we going to learn today!

1 Context

The lecture will be divided into sections:

- **Artificial Neural Networks**.
- **Transformer-based Architectures**.

Each section will provide a brief overview of the core concepts, the most important papers, and code snippets to help you better understand the concepts.

The training material is available at:
`https://github.com/istat-methodology/speaking-data`

Handwritten digits

1 Context

Most people can easily recognize the digits shown in the figure. Yet, this ease is deceptive.

In each hemisphere of the brain, humans have a **primary visual cortex containing approximately 140 million neurons**, with tens of billions of connections between them.

Vision involves a series of visual cortices that carry out **progressively more complex image processing**.

Figure: Handwritten digits

Handwritten digits: rule based approach

1 Context

What if we try writing a character recognition program (using Python)?

The difficulty of visual recognition becomes evident when we try to write a program to recognize digits. **What is easy for a human suddenly becomes extremely difficult for a program.**

Simple intuitions on how we recognize shapes - *"a 9 has a loop on top and a vertical stroke at the bottom right"* - turn out to be not so simple to express algorithmically.

```python
# Define a function to classify handwritten digits using classical programming
def classify_digit(image):
    # Image is a 28x28 matrix of pixel values (0-255)

    # Example of manual feature extraction based on pixel patterns
    # Check for patterns like vertical/horizontal lines, loops, etc.
    if check_vertical_line(image):
        if check_horizontal_line(image):
            return 8  # Looks like digit 8
        else:
            return 1  # Looks like digit 1
    elif check_loops(image):
        return 0  # Looks like digit 0
    elif check_diagonal_line(image):
        return 7  # Looks like digit 7
    else:
        # Complex conditions for other digits
        # Continue adding many more rules for all possible digit shapes
        return -1  # Unclassified

# Function to check vertical lines (for digit 1 or 8)
def check_vertical_line(image):
    # Look for strong vertical line in the middle of the image
    for row in range(28):
        if image[row][14] < 100:  # Arbitrary threshold for 'black' pixel
            return False
    return True
```

Figure: Rule-based digit recognition

Handwritten digits: neural networks

1 Context

Artificial Neural Networks (ANNs) tackle the problem differently: ANNs **learn from examples** instead of relying on hard-coded rules.

We feed the network a large number of **handwritten digits** (*training examples*), and the network **automatically learns** how to recognize them.

The more examples it sees, the better it gets - this is called **supervised learning**.

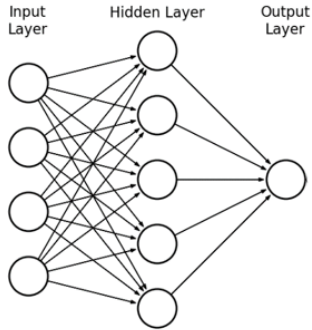


Figure: Multi-Layer Perceptron

Representation Matters

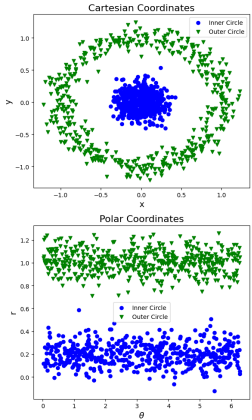
1 Context

What seems complex in one representation may become simple in another.

In the first plot, the two classes (circles and triangles) are mixed in Cartesian coordinates (x, y), making separation difficult.

By transforming the same data into Polar coordinates (r, θ), the classes become **trivially separable**.

This is what neural networks do: they **learn new representations** that make difficult tasks easier.

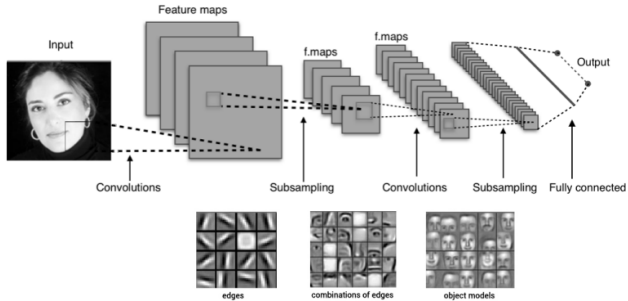


Neural Networks Learn Representations

1 Context

Neural networks work by **stacking layers of transformations**, building **increasingly abstract representations**.

Starting from raw pixels, they learn to detect *edges*, then *parts*, and finally recognize complex objects like *cars, persons, animals.*



Introduction to Artificial Neural Networks

2 Neural Networks (core concepts)

- Context
- Neural Networks (core concepts)
- Recurrent Neural Networks
- Language Models

From perception to models

2 Neural Networks (core concepts)

How can we model perception?

In biology, perception relies on neurons. Each neuron receives signals through the **dendrites**, processes them in the **soma**, and sends an output signal through the **axon** to the **synapse** — firing only if the signal is strong enough.

In machine learning, we use a simplified version: the **artificial neuron**. This is the basic building block of neural networks.

Let's now see how we can *model* this simplified neuron using maths.

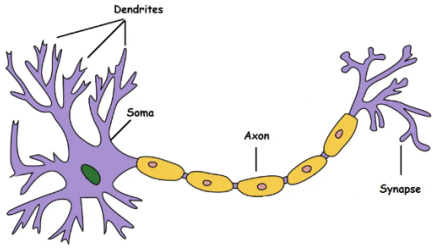


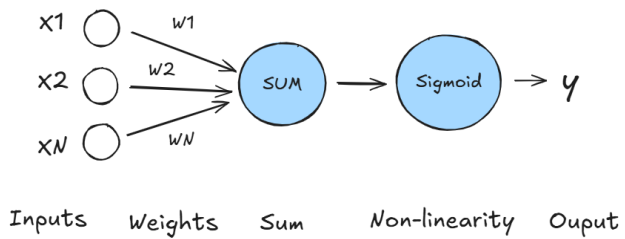
Figure: Biological neuron

The perceptron: modeling an artificial neuron

A perceptron receives **multiple inputs**, multiplies each input by a **weight**, and adds a **bias**. Then it applies a **non-linear** function, typically the **sigmoid**, to produce an output $\hat{y}$:

$$\hat{y} = g\left(\sum_{i=1}^{N} w_i x_i + b\right)$$

where g is the **activation function**, w_i are the **weights**, and b is the **bias**.



Importance of activation functions

The activation function g transforms the output of a neuron and introduces **non-linearity** into the network.

Without g, no matter how many layers we add, the network still behaves like a **linear model**.

Non-linear activation functions allow neural networks to approximate **complex functions**. *ReLU* is often used in hidden layers, *sigmoid* in binary classification.

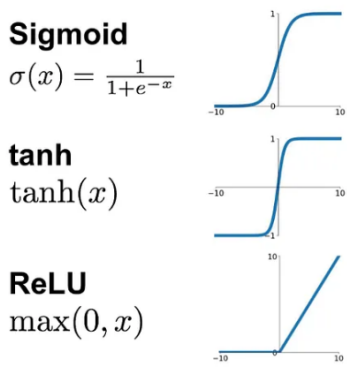
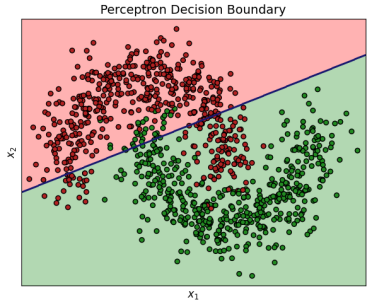
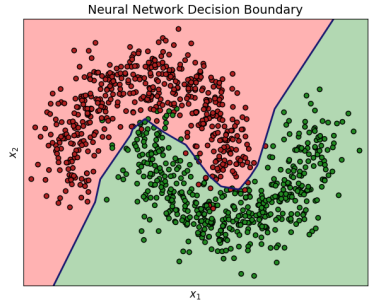


Figure: Activation functions

Importance of activation functions





Linear activation functions produce *linear classifiers* no matter the network size

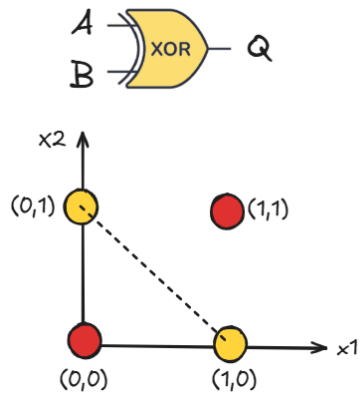
Non-linear elements allow the network to **approximate arbitrarily complex functions**

Linear separability: the perceptron's limitation

A single-layer perceptron can only solve problems where the data is **linearly separable** - meaning it can be split by a *straight line* (or a hyperplane in higher dimensions).

The **XOR** function is a classic example of a problem that is **not linearly separable**. No single straight line can separate the output classes.

A single perceptron cannot solve the XOR problem...

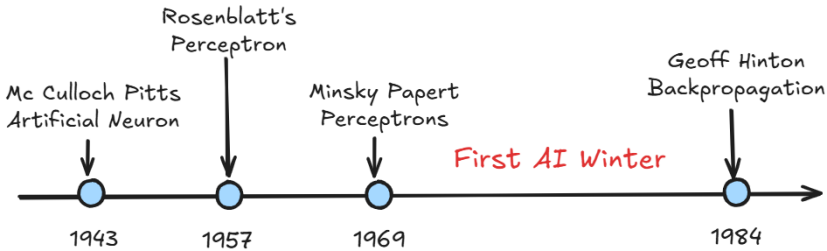


Early days of neural networks

1943: Artificial neuron was introduced by **McCulloch and Pitts**.

1957: **Rosenblatt** proposed the Perceptron.

1969: **Minsky and Papert** showed Perceptron limitations, particularly its inability to solve problems like XOR, leading to the **first AI winter**



Section 2.1

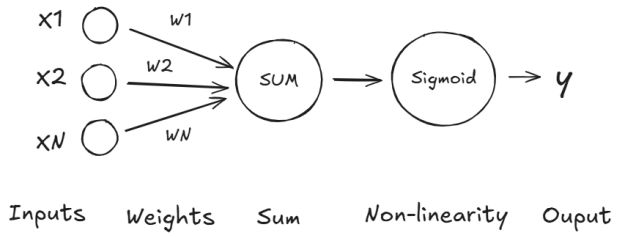
Building Neural Networks

The Perceptron: Theory (more formal)

A perceptron receives **multiple inputs**, multiplies each input by a **weight**, and adds a **bias**. Then, it applies a **non-linear function** to produce an output $\hat{y}$.

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

where $\mathbf{X}^\intercal \mathbf{W}$ is the **dot product** between inputs ($\mathbf{X}$) and weights ($\mathbf{W}$), g is the **activation function**, and w_0 is the **bias**.

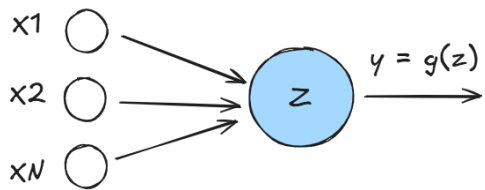


The Perceptron: Simplified View

We must simplify the diagram to introduce neural networks, i.e., models containing several neurons. To this aim, we will introduce z:

$$z = w_0 + \sum_{j=1}^{m} x_j w_j \implies \hat{y} = g(z)$$

We'll remove weights in the diagram and use only one circle to schematize dot product, bias, and non-linearity.



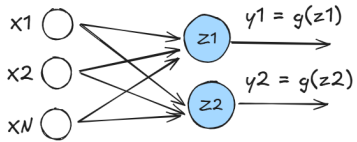
The Perceptron: Dense Neural Network

Suppose we want to implement a neural network that can classify an output with multiple modalities y_i, e.g., handwritten digits:

$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j w_{j,i} \implies \hat{y}_i = g(z_i)$$

In the network sketched in the diagram, all inputs are densely connected to all outputs (**dense neural network**).

In the sketched network, both neurons will process the same inputs, although their weights are different. Therefore, each neuron will produce a different output.



The Multi Layer Perceptron (MLP)

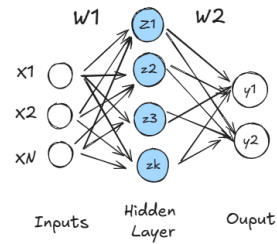
To solve more complex tasks, we can stack multiple layers of neurons. Let's consider a Neural Network with **one hidden layer**:

$$\text{Hidden Layer:} \quad z^{[1]} = W^{[1]}x + b^{[1]}, \quad a^{[1]} = g(z^{[1]})$$

$$\text{Output Layer:} \quad z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}, \quad \hat{y} = g(z^{[2]})$$

This architecture is known as a **Multi Layer Perceptron (MLP)** and is the foundation of modern deep learning.

Inputs are passed through deeper layers. Each layer extracts higher-level representations.



Recap and Coding Playground

2 Neural Networks (core concepts)

What have we learned?

- Neurons model how we process information — biologically and artificially.
- A **perceptron** computes a weighted sum and applies an activation.
- A **multi-layer perceptron (MLP)** stacks layers to solve complex problems.
- Non-linear **activation functions** are crucial to learn complex relations.

Now, let's start coding and implementing our first neural network!

Open the Colab Notebook

Section 2.2

Training Neural Networks

Learning by Example

2 Neural Networks (core concepts)

How do neural networks learn?

- Humans learn to recognize **patterns** - like cats and dogs - by seeing many **labeled** examples.
- **Supervised learning** follows the same idea: we show the network examples with the correct answers (labeled data).
- The network starts with *random weights*, makes predictions, and compares the output ($\hat{y}_i$) to the true labels (y_i).
- It then adjusts its internal parameters ($w_{j,i}$) to improve - this process is repeated many times.

Just like students learn from mistakes,
neural networks learn by **minimizing errors**.

Learning from Mistakes: the Loss Function

How do we measure how wrong the network is?

- The network computes a prediction $f(x^{(i)}; \mathbf{W})$ for each input $x^{(i)}$.
- The **loss function** $\mathcal{L}$ compares this prediction to the true label $y^{(i)}$.
- The loss quantifies the error: the further the prediction is from the actual value, the higher the loss.
- The goal is to **minimize the average loss** over all training examples:

$$\frac{1}{N} \sum_{i=1}^{N} \mathcal{L}\left(f(x^{(i)}; \mathbf{W}), y^{(i)}\right)$$

The loss is a way to tell the network: *"You're this far from the right answer."*

Binary Cross Entropy: Predicting Success

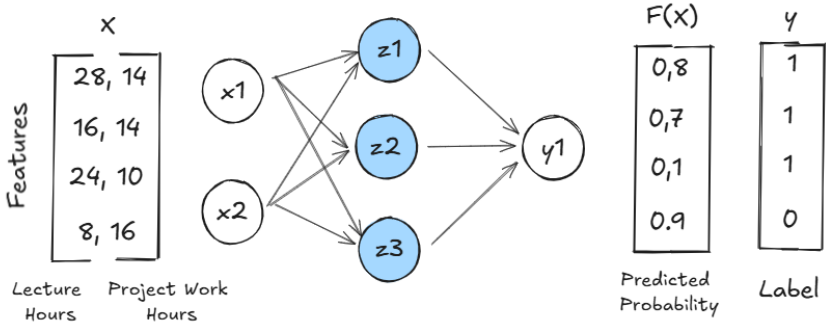
Example: Predicting whether a student will pass the exam ($y = 1$) or not ($y = 0$).

- The network outputs a probability $\hat{y} \in (0, 1)$.
- We want $\hat{y} \approx 1$ for passing the exam and $\hat{y} \approx 0$ for not passing.
- The **Binary Cross Entropy (BCE)** loss is:

$$\mathcal{L}(\hat{y}, y) = -\left[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})\right]$$

BCE is perfect when your goal is to predict the probability of success or failure.

Binary Cross Entropy: Predicting Success



Now, let's predict whether you'll pass the exam!

Open the Colab Notebook

Mean Squared Error: A Loss for Regression

Example: Predicting a student's final exam score (out of 30).

- The true score is $y = 26.0$, but the network predicts $\hat{y} = 23.5$.
- We want the prediction to be as close as possible to the actual score.
- The **Mean Squared Error (MSE)** is defined as:

$$\mathcal{L}(\hat{y}, y) = (\hat{y} - y)^2$$

MSE is ideal when you care about making accurate, continuous predictions.

Training: Learning by Optimization

Training a neural network means optimizing its weights to reduce the loss function $\mathcal{L}(\mathbf{W})$.

- The network starts with random weights $\mathcal{N}(0, \sigma^2)$.
- It makes predictions and computes the loss.
- Then it updates the weights to reduce the loss.
- Repeat this process for many examples

This process is called **training** and can be formalized as follows:

$$\min_{\mathbf{W}} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}\left(f\left(x^{(i)}; \mathbf{W}\right), y^{(i)}\right) \implies \min_{\mathbf{W}} J(\mathbf{W})$$

Training is like learning from feedback - over and over again.

Training: Gradient Descent

2 Neural Networks (core concepts)

How does the network know how to update the weights?

- We compute the **gradient** of the loss w.r.t. the weights.
- The gradient tells us the direction in which the loss increases.
- We move in the opposite direction to **minimize** the loss.
- Update rule:

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \cdot \frac{\partial J}{\partial \mathbf{W}}$$

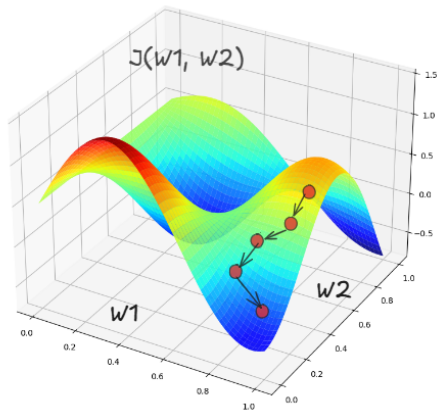
- η is the **learning rate**: how big each step is.

Think of it as walking downhill until you reach the bottom.

Gradient Descent: Visualizing the Loss

Training means finding the lowest point on the loss surface.

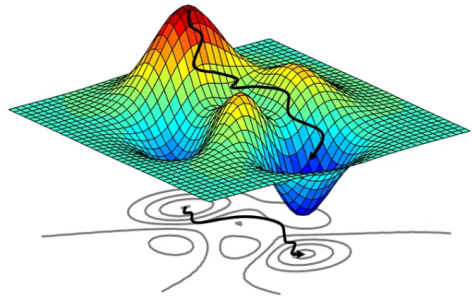
- The image shows a three-dimensional representation of the loss function, which depends on the weights $J(\mathbf{W}_1, \mathbf{W}_2)$.
- Gradient descent is like following the steepest slope downhill.
- Each step updates the weights $\mathbf{W}_1, \mathbf{W}_2$ to reduce the loss.



Gradient Descent: Summary

Gradient Descent Algorithm

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

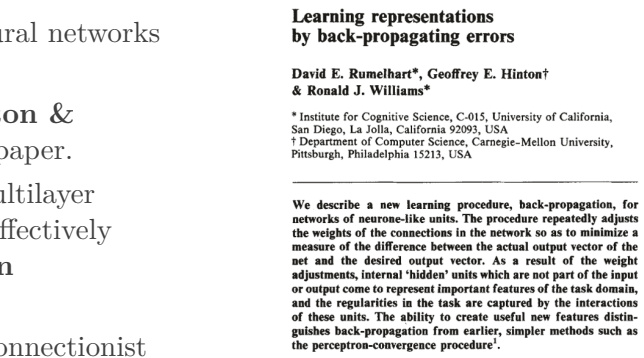


How do we compute the gradients? We need an efficient algorithm...

The Revival of Neural Networks

2 Neural Networks (core concepts)

- In the 1980s, interest in neural networks was declining (AI winter).
- In 1986, **Rumelhart, Hinton & Williams** published a key paper.
- They demonstrated that multilayer networks could be trained effectively using the **backpropagation algorithm**.
- This reignited research in connectionist models and laid the groundwork for modern deep learning.



Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton† & Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure[1].

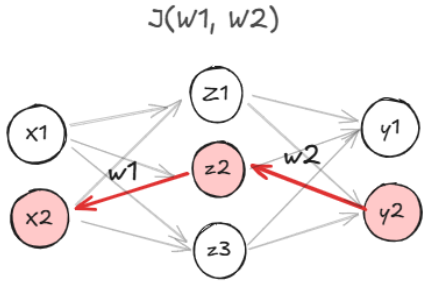
Figure: Learning representations by back-propagating errors - Nature 1986

Backpropagation: Overview

2 Neural Networks (core concepts)

How do we compute gradients in deep networks?

- Apply the **chain rule** to compute gradients layer by layer
- Propagate the error **backwards** through the network
- Backpropagation allows efficient computation of gradients in multilayer networks.



$$\frac{\partial J}{\partial w_2} = \frac{\partial J}{\partial y_2} \cdot \frac{\partial y_2}{\partial w_2} \qquad \frac{\partial J}{\partial w_1} = \frac{\partial J}{\partial y_2} \cdot \frac{\partial y_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial w_1}$$

Handwritten Digit Recognition

2 Neural Networks (core concepts)

- Input: 28×28 grayscale digit images
- Output: One of 10 digit classes (0-9)
- Activation: ReLU in hidden layers, Softmax in output
- Loss function: **Categorical Cross Entropy**

Let's teach a network to read handwritten digits!

Open the Colab Notebook

Introduction to Artificial Neural Networks

3 Recurrent Neural Networks

- Context
- Neural Networks (core concepts)
- Recurrent Neural Networks
- Language Models

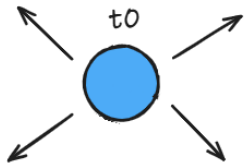
Section 3.1

Modeling Sequences

Sequences Matter: A Simple Question

3 Recurrent Neural Networks

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

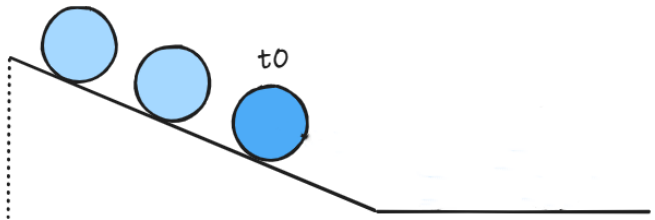


Without providing any prior information regarding the ball's history...

The task is almost impossible.

Sequences Matter: A Simple Question

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

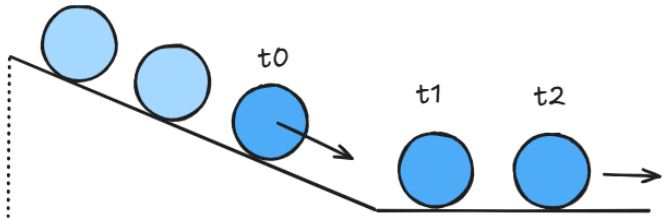


What if I provide the previous positions and the forces acting on the ball?

The task becomes feasible.

Sequences Matter: A Simple Question

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



The task is much easier: **given the previous positions of the ball, predict where the ball will go next**

What is a Sequence?

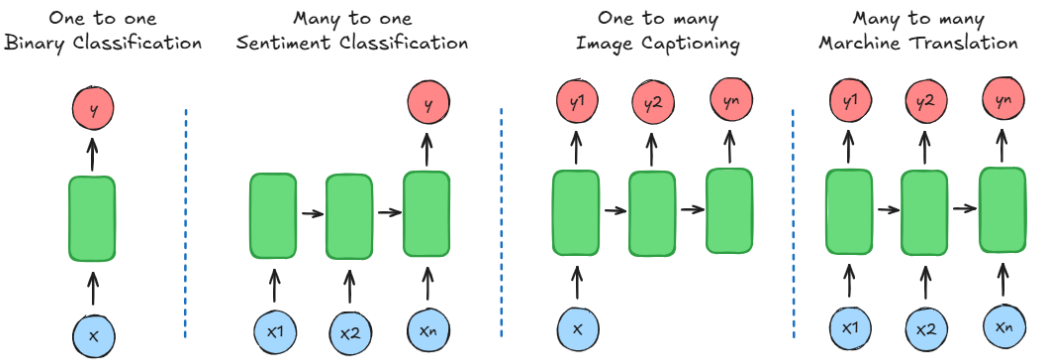
A sequence is an ordered set of elements where the *order* matters.

- **Text**: a sequence of words forming a sentence.
- **Speech**: a sequence of audio samples over time.
- **Video**: a sequence of frames capturing motion.
- **Sensor Data**: a sequence of readings from a sensor (e.g., heart rate, temperature).
- **Financial Data**: a sequence of prices or market trends over time.

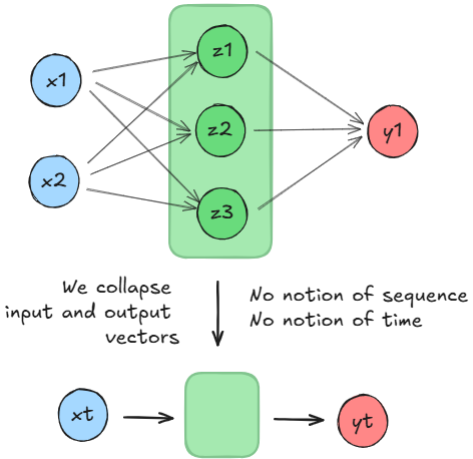
In all these cases, understanding the dependencies between elements in the sequence is key to solving real-world problems.

Sequence Modeling Applications

3 Recurrent Neural Networks



Feed-Forward Networks Revisited



In the simplified model (bottom) we map an input $\mathbf{x}_t \in \mathbb{R}^m$ to an output $\hat{\mathbf{y}}_t \in \mathbb{R}^n$.

Key limitations:

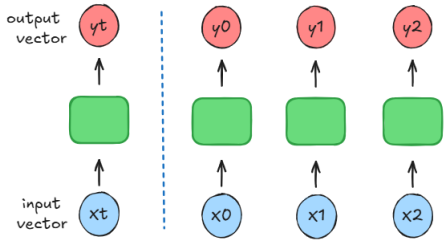
- Treats all inputs as if they happen at the **same time**.
- Lacks **memory** of previous inputs.
- Ignores the **order** of the data.

This means the model cannot connect the current output with any past input.

Introducing the Concept of Time

To handle sequences, we process inputs **step by step**, introducing the concept of **time** t:

- $x_0, x_1, \ldots, x_n$.
- $y_0, y_1, \ldots, y_n$.
- $\mathbf{y_t} = \mathbf{f}(\mathbf{x_t})$.



Limitation:

- Each time step is processed **independently**.
- Each prediction is made without considering previous inputs.

Predicting Stock Market Trends

3 Recurrent Neural Networks

Imagine this strategy:

- The model sees only the current stock price at time x_t.
- It decides whether to **hold or sell**, ignoring all past data $\mathbf{y_t} = \mathbf{f}(\mathbf{x_t})$.

Limitation:

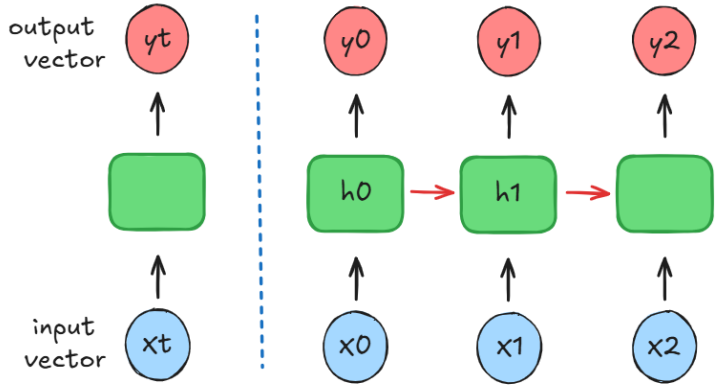
- No awareness of trends or volatility.
- A snapshot without historical context leads to **poor decisions**.

To make better decisions, the model needs to consider the price history...

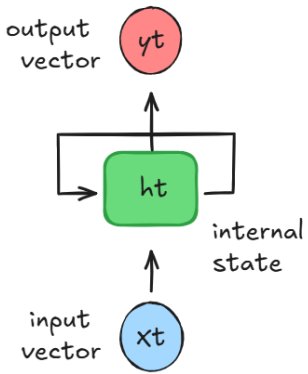
Introducing Recurrence

3 Recurrent Neural Networks

From Independent to Connected Steps



Neurons with Recurrence



Each output $\hat{\mathbf{y}}_t$ depends on:

- The current input $\mathbf{x}_t$
- The **internal state** $\mathbf{h}_{t-1}$ from the previous step. $\mathbf{h}_t$ connects the current output at earlier inputs.

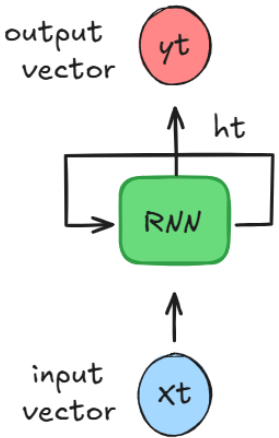
The network **remembers** what happened before through the state $\mathbf{h}_t$. This state introduces a notion of **memory** in our architecture.

$$\hat{\mathbf{y}}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1})$$

Section 3.2

Recurrent Neural Networks

Recurrent Neural Networks (RNNs)



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

$$\mathbf{h}_t = f_W(\mathbf{x}_t, \mathbf{h}_{t-1})$$

- $\mathbf{h}_t$: cell state at time t
- f_W: function with trainable weights W
- $\mathbf{x}_t$: input at time t
- $\mathbf{h}_{t-1}$: state from previous step

RNNs maintain a state $\mathbf{h}_t$ (**hidden state**) that gets updated as the sequence is processed.

Insight: Recurrence Beyond Neural Networks

Factorial	**RNN Hidden State Update**
$n! = n \times (n-1)!$	$\mathbf{h}_t = f_W(\mathbf{x}_t, \mathbf{h}_{t-1})$

Key Similarities:

- Both define the **current value** based on the **previous one**.
- Both apply the **same rule** repeatedly over steps.
- Both build up a result by **propagating state**.

Both rely on recurrence to build up results progressively.

RNN Intuition (Pseudocode)

Recurrent Neural Network (pseudo code)

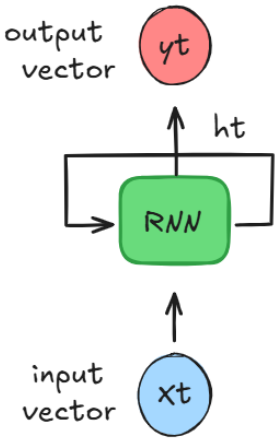
```
[ ]  # Initialize the RNN model
     rnn = RNN()

     # Initialize the hidden state
     hidden_state = initial_state()
     #hidden_state = [0,0,0,0]

     # Sequence to process
     sequence = ["I", "love", "recurrent", "neural"]

     # Process sequence step by step
     for word in sequence:
         y, hidden_state = rnn(word, hidden_state)

     # Final output after processing the entire sequence
     output = y

     print("RNN output: ", output)
     # RNN output: networks
```

Key ideas:

- Process the sequence **step by step**.
- Update the **hidden state** at each step.
- Reuse the **same RNN model** on each word.
- The **final output** depends on the entire sequence.

This is how RNNs process sequences one element at a time.

Recurrent Neural Networks (RNNs)



Update the **hidden state** at each time step:

$$\mathbf{h}_t = \tanh(\mathbf{W}_{\mathbf{hh}}^\top \mathbf{h}_{t-1} + \mathbf{W}_{\mathbf{xh}}^\top \mathbf{x}_t)$$

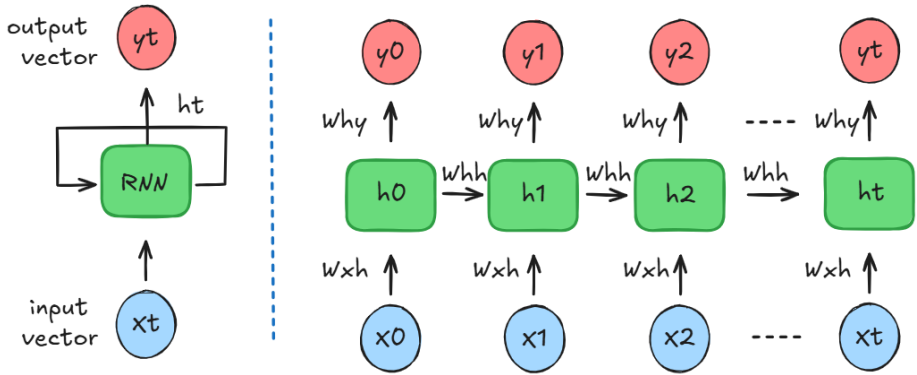
The output is computed from the hidden state:

$$\hat{\mathbf{y}}_t = \mathbf{W}_{\mathbf{hy}}^\top \mathbf{h}_t$$

- $\mathbf{x}_t$: input at time t
- $\mathbf{h}_t$: hidden state at time t
- $\mathbf{W_{xh}}, \mathbf{W_{hh}}, \mathbf{W_{hy}}$: weight matrices (learned during training)
- tanh: non-linear activation function

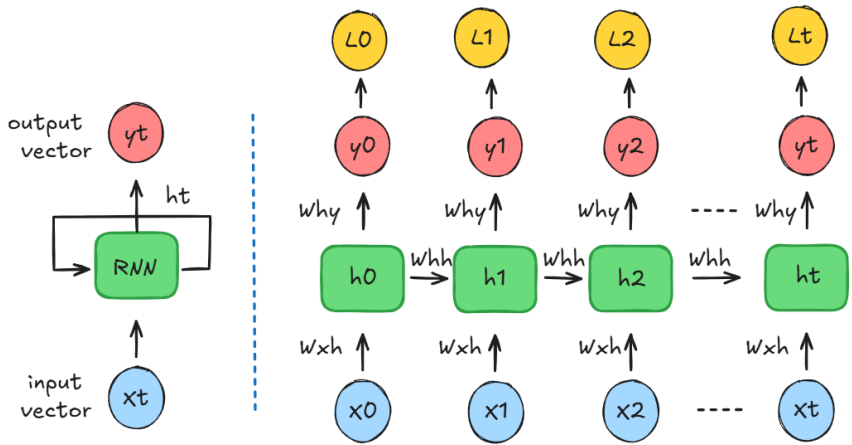
Unfolding RNNs

RNNs reuse the same weights $\mathbf{W_{xh}}, \mathbf{W_{hh}}, \mathbf{W_{hy}}$ at each step



Unfolding RNNs with Loss

We calculate $\mathcal{L}(t)$ at each time step, then perform the average $\frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(t_i)$



RNNs with TensorFlow

3 Recurrent Neural Networks

Recurrent Neural Network with Tensor Flow

```python
import tensorflow as tf

# Define a custom RNN cell class
class RNNCell(tf.keras.layers.Layer):
    def __init__(self, rnn_units, input_dim, output_dim):
        # Initialize the base Layer
        super(RNNCell, self).__init__()

        # Define weight matrices:
        # - W_xh: from input to hidden state
        # - W_hh: from previous hidden state to new hidden state
        # - W_hy: from hidden state to output
        self.W_xh = self.add_weight(shape=(rnn_units, input_dim))
        self.W_hh = self.add_weight(shape=(rnn_units, rnn_units))
        self.W_hy = self.add_weight(shape=(output_dim, rnn_units))

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

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

        # Compute the output from the current hidden state
        output = self.W_hy @ self.h

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

What the code shows:

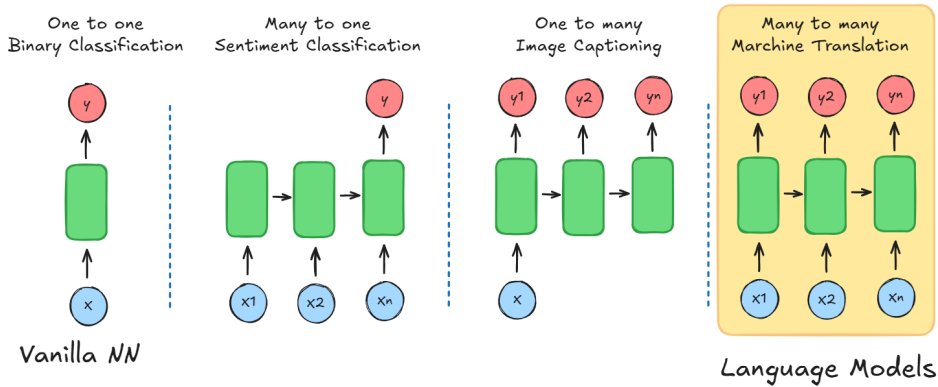
- **Weight matrices:** W_{xh}, W_{hh}, W_{hy}.
- **Hidden state update:** $\tanh(W_{hh}h_{t-1} + W_{xh}x_t)$.
- **Output computation:** $y_t = W_{hy}h_t$.
- **State reuse:** the same state flows through the sequence.

This mirrors the mathematical formulation we introduced earlier.

From Vanilla Networks to Language Models

3 Recurrent Neural Networks

Language Models: Predict or generate **many-to-many** sequences



Introduction to Artificial Neural Networks

4 Language Models

- Context
- Neural Networks (core concepts)
- Recurrent Neural Networks
- Language Models

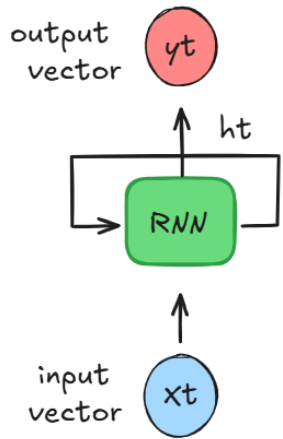
Language models: problem statement

What challenges does a **language model** need to address?

To model text as a sequence, we need to:

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

Recurrent Neural Networks meet these criteria, but are they the best choice?



Section 4.1

Language models: next token prediction

Can You Guess the Next Word?

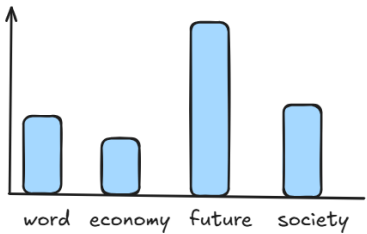
4 Language Models

Language Models learn to predict the next word based on the words they have already seen (*context*). Let's consider the following example:

"Artificial Intelligence is shaping the _____."

Possible predictions:

- world
- economy
- future
- society

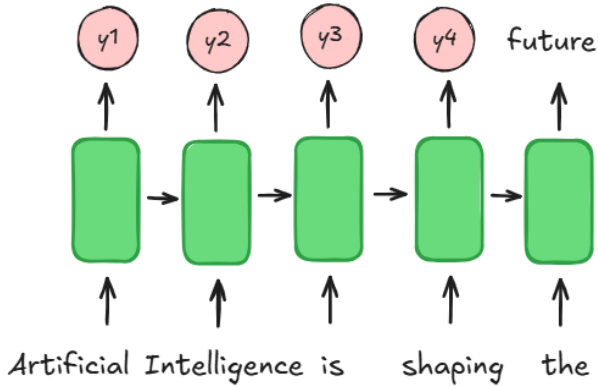


Language Models rank these outputs and select the most likely one.

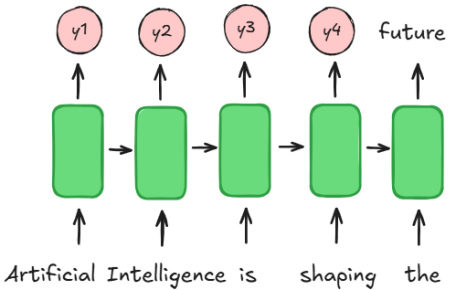
Can You Guess the Next Word?

4 Language Models

We could implement a RNN to achieve this task



Why This Naive Approach Fails?



Looks good, but...

- Feeding raw text to a Neural Network **does not work**.
- **Neural Networks process numbers, not words.**
- We need to **convert words into tokens and vectors**.

*This is the role of **Tokenization** and **Embeddings** in Language Models.*

Section 4.2

Tokenization and Word Embeddings

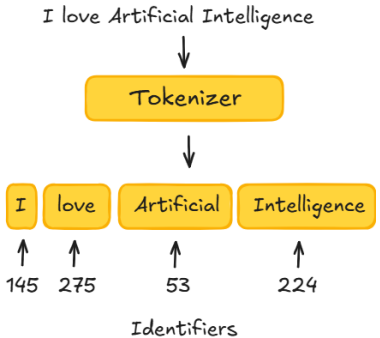
Tokenization: Breaking Text into Pieces

What is Tokenization?

Splits text into smaller units called **tokens**.

- **Words** (e.g., `"Artificial"`)
- **Subwords** (e.g., `"Intelli"`, `"gence"`)
- **Characters** (e.g., `"A"`, `"r"`, ...)

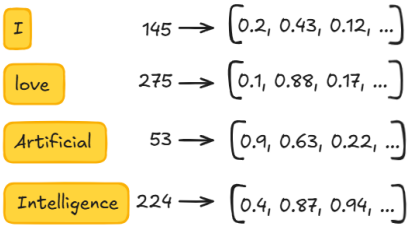
The tokenizer builds a **vocabulary of size** N, assigning a unique integer to each token.



Embeddings: From Tokens to Vectors

What are Embeddings?

- Neural Networks work with **vectors of numbers**, not identifiers.
- An **embedding layer** converts each token identifier into a **dense vector** of fixed size.
- These vectors capture **semantic relationships** between tokens.



Let's delve into embedding spaces...

Embeddings: Capturing Meaning in Vectors

- **Word Embeddings** map words to **dense vectors** in a continuous space.
- The **distance** between vectors reflects **semantic similarity** between the corresponding words.
- These vectors capture **relationships** between different words.
- Traditional word embeddings provide **a single vector representation** for each word, regardless of context.
- Directions in this space encode **semantic dimensions**, such as: gender, plurality and verb tense.

Embeddings allow neural networks to generalize and understand relationships between words.

Visualizing Word Embeddings

Words in a Vector Space

The simplest way to grasp the intuition behind word embeddings is to visualize how words are placed in a **vector space**.

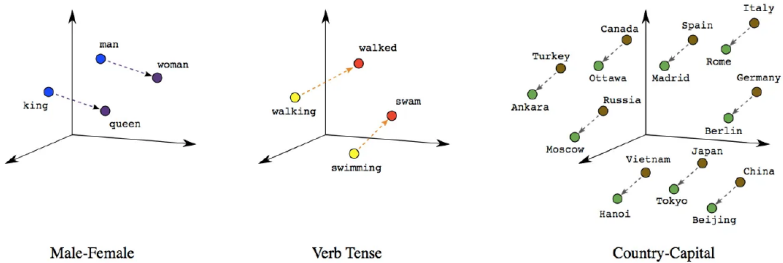


Figure: Example of word vectors (image from Google for Developers)

Vector Similarity: the Dot Product

How do we measure similarity between vectors in a space?
We can use the **dot product**.

$$\mathbf{a} \cdot \mathbf{b} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \cdot \begin{bmatrix} b_1 & b_1 & b_1 \end{bmatrix} = a_1 b_1 + a_2 b_1 + a_3 b_1$$

The higher (in magnitude) the dot product, the higher the alignment (positive or negative); the lower the dot product, the lower the alignment.

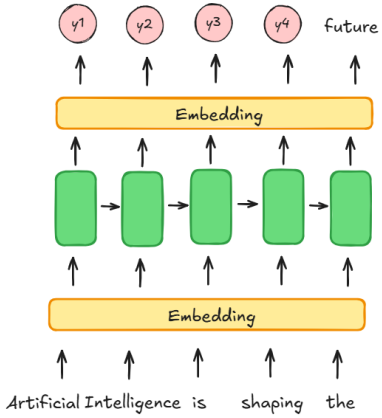
This allows the machine to estimate **similarity between words and concepts**.

Can You Guess the Next Word?

We add an Embeddings layer:

- Input words are mapped to **dense numerical vectors**.
- The Neural Network processes these **vector representations**.
- The output vectors are then **decoded back into words**.

It sounds promising, but...



Section 4.3

Limitations of RNN

Struggling with Long-Term Dependencies

Theory:, RNNs can remember distant information.

Practice: the longer the distance, the more they forget.

Example: Same output word, increasing distance of the key clue

- *The dog* _____ → **barked**
- *The dog that saw the cat* _____ → **barked**
- *The dog, noticed the cat walking on the window, therefore it* _____ → **barked**

In longer sequences, RNNs tend to lose the subject and make incorrect predictions.

Struggling with word order

Another challenge is capturing word order correctly.

Word Order Matters

Subtle changes in the order of words affect meaning. RNNs process words sequentially, but may not preserve these nuances.

$$\textit{"John loves Mary"} \quad \neq \quad \textit{"Mary loves John"}$$

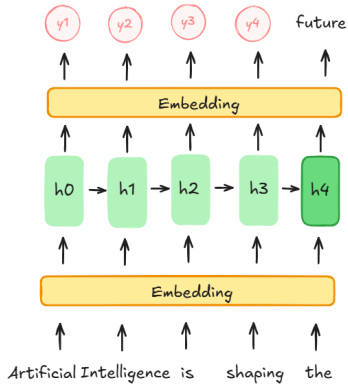
Word meaning depends on position — and RNNs may miss that.

Limitations of Recurrent Models

Even though RNNs can handle sequences, they face several **critical limitations**:

- **Encoding bottleneck:** a fixed-length vector that encapsulates the meaning of the sentence.
- **No parallelization:** sequences are processed step-by-step.
- **Limited memory:** struggles with long-range dependencies.

To overcome these issues, we need a more structured approach…



Section 4.4

Encoder - Decoder Architecture

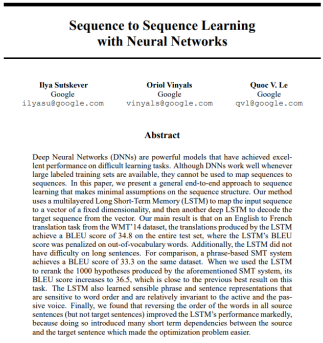
A Turning Point: Sequences to Sequences

4 Language Models

In 2014, **Sutskever, Vinyals, and Le** introduced a major breakthrough: the first successful **sequence-to-sequence (seq2seq)** model based on deep RNNs.

- Introduced the **encoder–decoder architecture**.
- Trained end-to-end to *generate output sequences*, such as translations.
- Key idea: *compress the entire input into a fixed-size context vector.*

The model learns to produce sequences — not just classify them…



Sequence to Sequence Learning with Neural Networks

Ilya Sutskever Google ilyasu@google.com

Oriol Vinyals Google vinyals@google.com

Quoc V. Le Google qvl@google.com

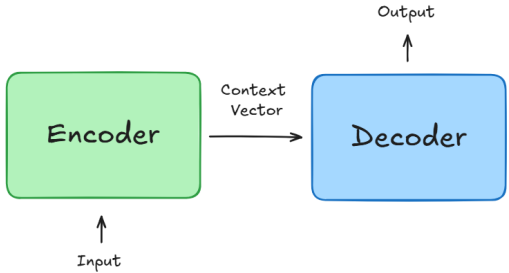
Abstract

Deep Neural Networks (DNNs) are powerful models that have achieved excellent performance on difficult learning tasks. Although DNNs work well whenever large labeled training sets are available, they cannot be used to map sequences to sequences. In this paper, we present a general end-to-end approach to sequence learning that makes minimal assumptions on the sequence structure. Our method uses a multilayered Long Short-Term Memory (LSTM) to map the input sequence to a vector of a fixed dimensionality, and then another deep LSTM to decode the target sequence from the vector. Our main result is that on an English to French translation task from the WMT'14 dataset, the translations produced by the LSTM achieve a BLEU score of 34.8 on the entire test set, where the LSTM's BLEU score was penalized on out-of-vocabulary words. Additionally, the LSTM did not have difficulty on long sentences. For comparison, a phrase-based SMT system achieves a BLEU score of 33.3 on the same dataset. When we used the LSTM to rerank the 1000 hypotheses produced by the aforementioned SMT system, its BLEU score increases to 36.5, which is close to the previous best result on this task. The LSTM also learned sensible phrase and sentence representations that are sensitive to word order and are relatively invariant to the active and the passive voice. Finally, we found that reversing the order of the words in all source sentences (but not target sentences) improved the LSTM's performance markedly, because doing so introduced many short term dependencies between the source and the target sentence which made the optimization problem easier.

The Encoder–Decoder Architecture

The **encoder–decoder architecture** is the foundation of sequence-to-sequence modeling: it maps an input sequence to a fixed-size context vector, then generates an output sequence.

- The **encoder** processes the input and encodes it into a single vector.
- The **decoder** unfolds this vector into the target sequence.



The Encoder–Decoder Architecture

Let's take a closer look at the three key elements of the encoder–decoder structure:

1. Encoder

The encoder processes each token in the input sequence step by step. Its goal is to compress the entire sequence into a single vector — the *context vector*.

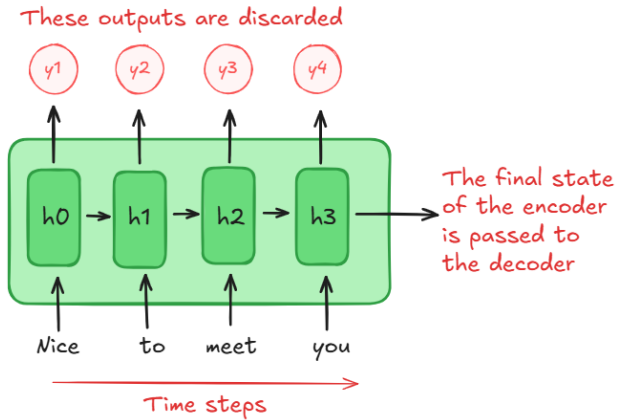
2. Context Vector

This fixed-length vector is expected to encapsulate the *full meaning* of the input sequence. It acts as a bridge, summarizing the input and guiding the decoder's predictions.

3. Decoder

The decoder receives the context vector and generates the output sequence *token by token*. At each step, it uses the context and previously generated tokens to predict the next one.

The Encoder Block

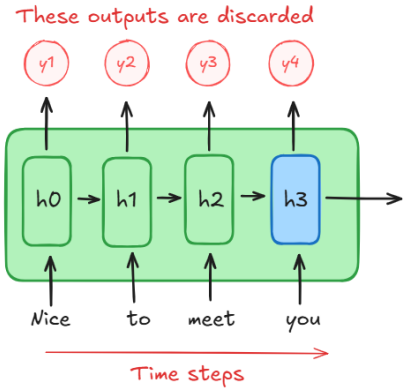


The Encoder Block

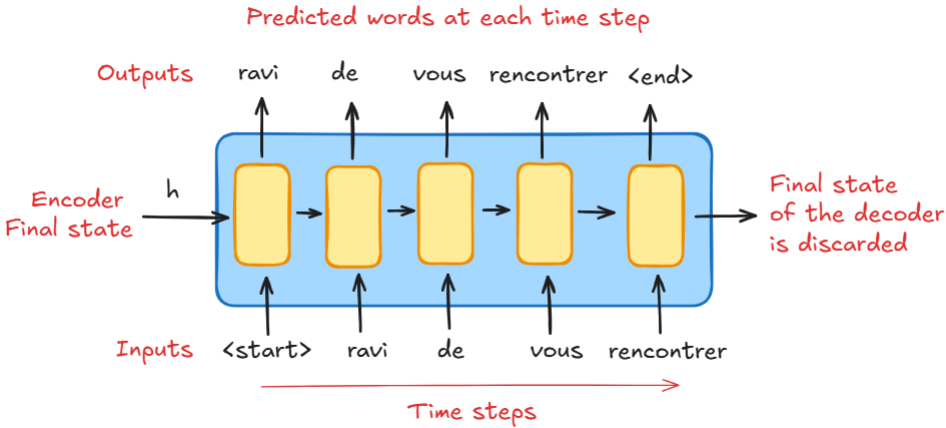
What does the encoder do? The encoder processes the input sequence token by token. Each word is transformed into a hidden state h_i.

We do not use all hidden states in the decoding phase:

- Only the **final hidden state** is passed forward.
- This final state becomes the **context vector** — a summary of the entire input.
- The intermediate outputs are **discarded**.



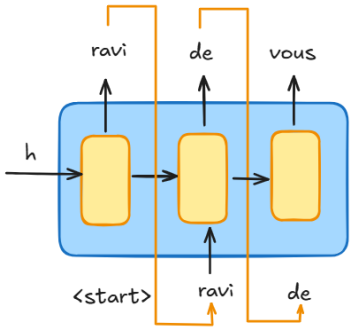
The Decoder Block



The Decoder Block

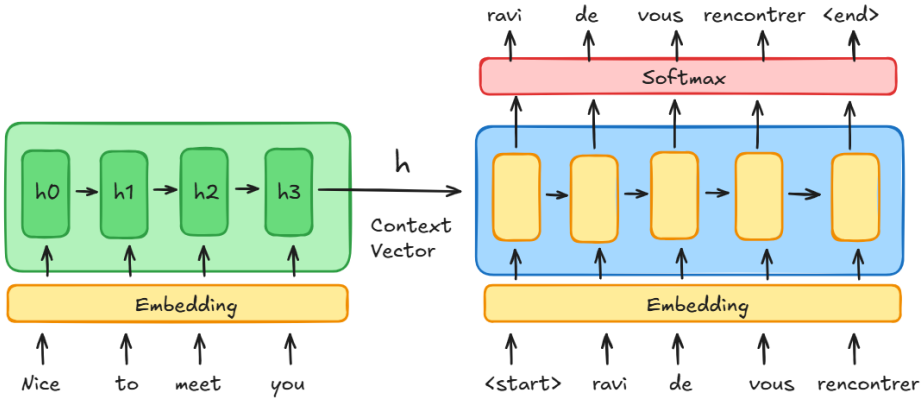
What does the decoder do? The decoder receives the **context vector** (i.e., the final state of the encoder) and generates the target sentence one token at a time.

- The decoder begins with a special `<start>` token.
- At each step, it predicts the next word in the output sequence.
- The prediction from the previous time step is used as input for the next.
- The process continues until the model predicts a special `<end>` token.



Encoder - Decoder in Action

Putting all the pieces together, we arrive at this end-to-end model.



Limitations of the Encoder-Decoder Model

The encoder-decoder model works well for short sentences but faces significant challenges as input length grows.

- **Information bottleneck:** All input information must be compressed into a single fixed-size vector, regardless of sentence length or complexity.
- **Loss of detail:** Important context from earlier parts of the input may be forgotten - especially in long sequences.
- **No dynamic access:** The decoder has no direct access to the individual encoder states; it must rely only on the final vector.

To overcome these issues, we need a way for the decoder to "look back" at the input — this is where attention comes in.

Section 4.5

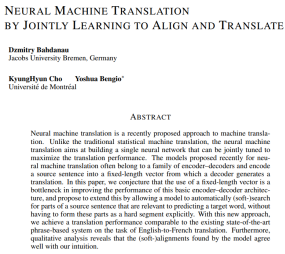
The Attention Mechanism

Learning to Align and Translate

4 Language Models

In 2015, to overcome the fixed-size vector bottleneck, *Bahdanau, Cho, and Bengio* proposed a new mechanism: **soft search**.

- Instead of compressing all information into a single vector, the decoder can now **access all encoder hidden states**.
- At each decoding step, the model learns to **attend** to different parts of the input — based on relevance.



NEURAL MACHINE TRANSLATION
BY JOINTLY LEARNING TO ALIGN AND TRANSLATE

Dzmitry Bahdanau
Jacobs University Bremen, Germany

KyunghHyun Cho **Yoshua Bengio***
Université de Montréal

ABSTRACT

Neural machine translation is a recently proposed approach to machine translation. Unlike the traditional statistical machine translation, the neural machine translation aims at building a single neural network that can be jointly tuned to maximize the translation performance. The models proposed recently for neural machine translation often belong to a family of encoder–decoders and encode a source sentence into a fixed-length vector from which a decoder generates a translation. In this paper, we conjecture that the use of a fixed-length vector is a bottleneck in improving the performance of this basic encoder–decoder architecture, and propose to extend this by allowing a model to automatically (soft-)search for parts of a source sentence that are relevant to predicting a target word, without having to form these parts as a hard segment explicitly. With this new approach, we achieve a translation performance comparable to the existing state-of-the-art phrase-based system on the task of English-to-French translation. Furthermore, qualitative analysis reveals that the (soft-)alignments found by the model agree well with our intuition.

This mechanism — where the decoder attends to encoder outputs — is called ***cross-attention****. To understand it fully, we first introduce a simpler concept:* ***self-attention****.*

Self-Attention Intuition

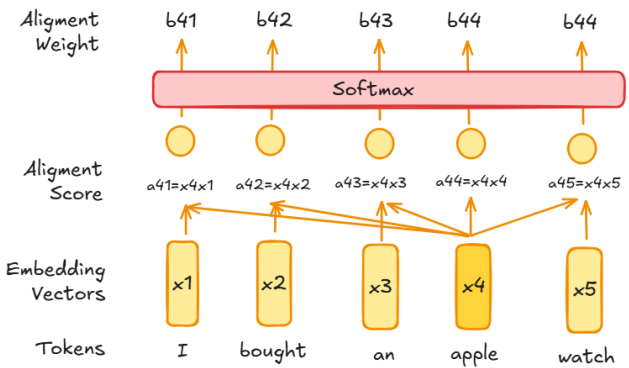
To understand self-attention, imagine updating the representation of each word based on a **weighted average of all the other words** in the sentence — including itself.

- Each token has a vector representation (embedding).
- For each token, we compute a score (*alignment score*), measuring the alignment to the other tokens.
- These weights are used to build a new, context-enriched vector.
- The result: every word can ”see” the rest of the sentence and adapt its meaning.

This allows the model to capture dependencies — regardless of position or distance.

Self-Attention in Action

Here's how we compute the updated vector for the word **apple** using a weighted sum of all embedded tokens:



Self-Attention in Action

Here's how we compute the updated vector for the word **apple** using a weighted sum of all embedded tokens:

$$\mathbf{x}'_4 = \alpha'_{4,1}\mathbf{x}_1 + \alpha'_{4,2}\mathbf{x}_2 + \alpha'_{4,3}\mathbf{x}_3 + \alpha'_{4,4}\mathbf{x}_4 + \alpha'_{4,5}\mathbf{x}_5$$

- Each alignment weight $\alpha'_{4,j}$ tells us how much attention the word **apple** pays to the j-th token.
- The updated vector $\mathbf{x}'_4$ is a mixture of all the input vectors $\mathbf{x}_j$, scaled by these weights.

How Self-Attention Works with Q, K, V

In modern language models, self-attention is computed using three learned matrices: **Query (Q)**, **Key (K)**, and **Value (V)**.

Each input token vector $\mathbf{x}_i$ is transformed into:

- **query** vector $\mathbf{q}_i$ (what I'm looking for),
- **key** vector $\mathbf{k}_j$ (what I contain),
- **value** vector $\mathbf{v}_j$ (what I offer).

The attention score between token i and j is computed as:

$$\text{score}_{i,j} = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}$$

Each token gathers contextual information from the whole sentence.

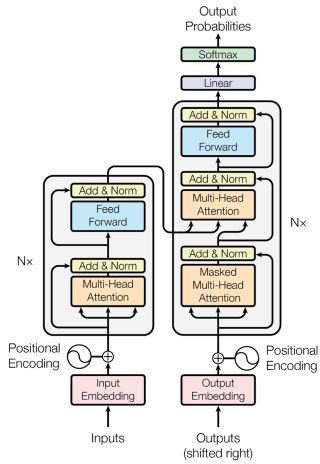
Attention Is All You Need

4 Language Models

In 2017, **Vaswani et al.**. introduced a model that relies entirely on **self-attention**.

- Introduced the **Transformer** architecture.
- Replaces RNNs with **multi-head self-attention**.
- Enables **parallelization** and faster training.

This model changed the field forever.



A Journey Through Language Models...

4 Language Models

Over the last few hours, we've explored the foundations of how machines learn to understand and generate language.

- We started with **perceptrons**, the simplest form of artificial neurons.
- You learned how **neural networks** stack and transform information into powerful representations.
- You discovered how **recurrent networks** process sequences.
- You followed the evolution toward **language models**, **encoder–decoder architectures**, and finally...
- You reached the **Transformer** — the architecture behind today's AI.

The adventure has just begun!