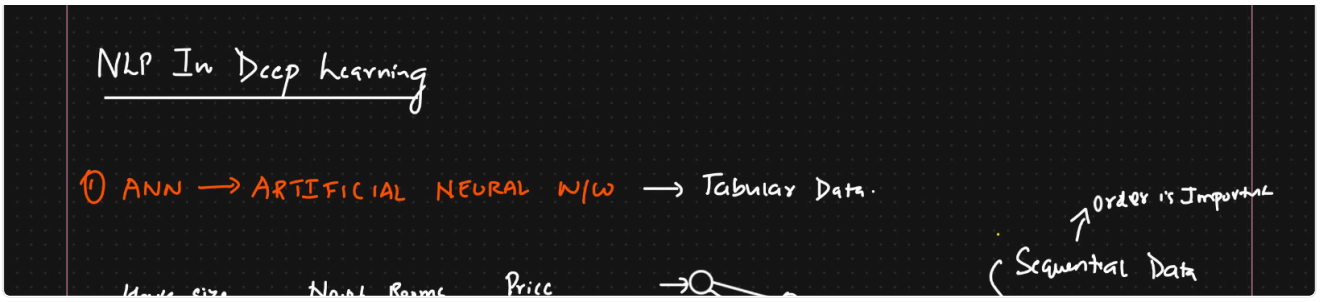
02.Deep Learning for NLP

Tags: #permanent
Description:
Theme:Mestrado ITA 🇧🇷, CM 219 - Processamento de Linguagem Natural (NLP), Machine Learning for NLP

ID: 20250418151344



Brief Description:

This documentation discusses **Artificial Neural Networks (ANNs)** and **deep learning architectures** specifically applied to tasks involving **text data**. It focuses on how to convert raw text into numerical vectors that can be processed by machine learning and deep learning models.

> **Text data → Vectors → Numerical representations**

The following techniques are commonly used for this transformation:

1. One-Hot Encoding
2. Bag of Words (BoW)
3. Term Frequency–Inverse Document Frequency (TF-IDF)
4. Word2Vec and Average Word2Vec

These methods have demonstrated strong performance in classical NLP tasks such as **sentiment analysis**, **text classification**, and **document clustering**. However, to go one step further into more advanced tasks, it's necessary to explore **deep learning-based Natural Language Processing (NLP)** approaches.

Artificial Neural Network (ANN)

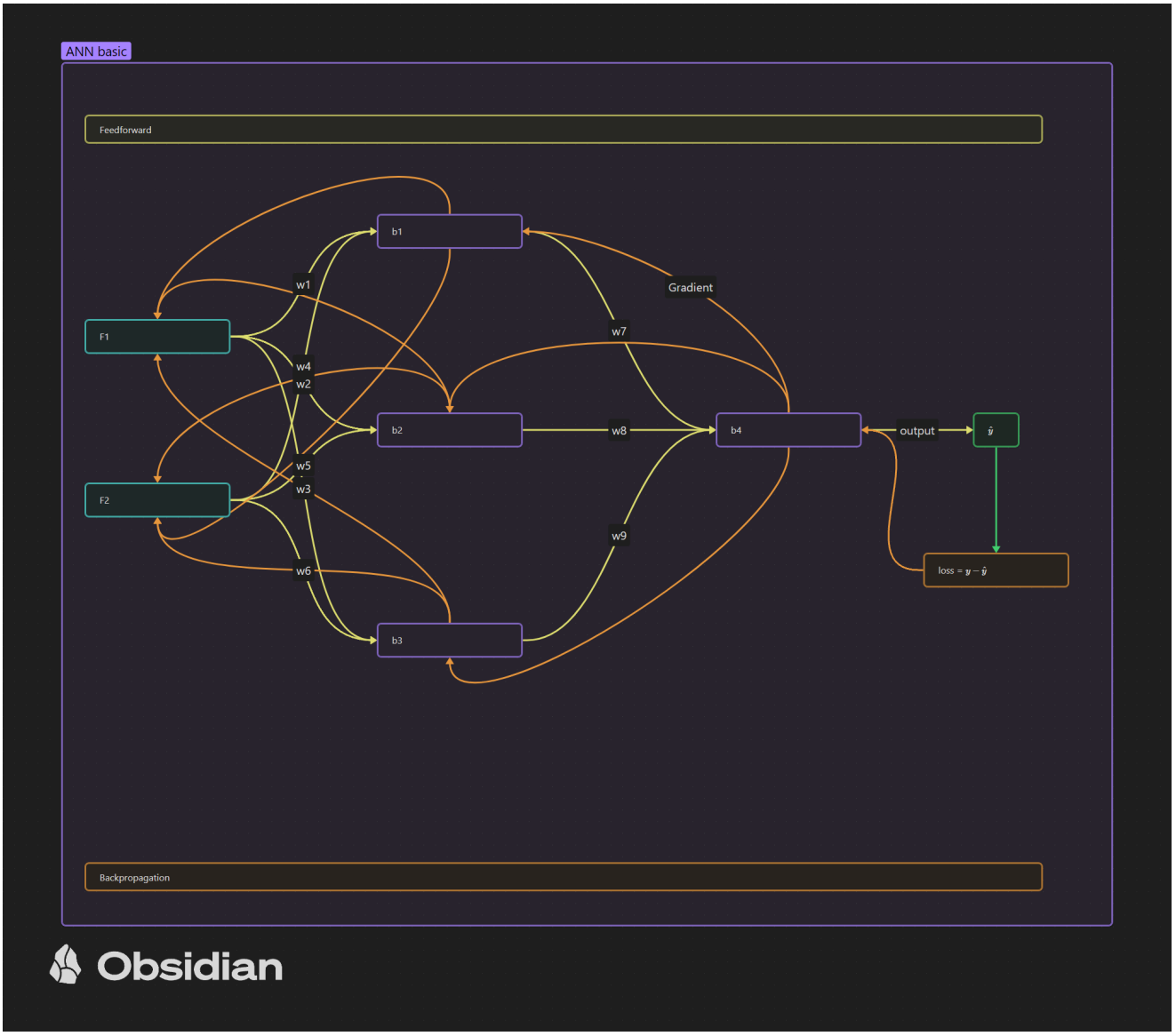


Figure 01: ANN architecture

Considering a tabular data as input, an **Artificial Neural Network (ANN)** is capable of performing tasks such as **classification** or **regression**, depending on the nature of the output which sequence of data doesn't matter.

Convolution Neural Networks

Convolutional Neural Network (CNN) architectures are typically used for data with spatial structure, such as **images**, **video frames**, and **medical imaging**. CNNs have shown exceptional performance in a wide range of computer vision tasks, including:

1. **Image Classification** – Predicting the class or category of an entire image.
2. **Object Detection** – Identifying and locating objects within an image.

Recurent Neural Network and others like

1. RNN
2. LSTM RNN
3. GRU RNN

4. Encoder Decoder
5. Attention is all you need

Recurent Neural Network and others like are specially used with **sequential data.**

kind of application:

1. text generation
2. chatbot conversation
3. meaning changed
4. language translation
5. Auto suggestion
6. sales data (sequential data beacause is in datetime) to sales forcasting

Example

Dataset - Sentiment Analysis

Text	**O/P**	
the food is good	1	
the food is bad	0	
the food is not good	0	

unic words:
6
vocabulary:
food
good
bad
not

Steps

1. text preprocessing
text to vector
BOW
TF-IDF
Word2Vec

BOW

	food	good	bad	not
S1	1	1	0	0
S2	1	0	1	0
S3	1	0	1	1

For this specific task - sentimental analysis - , **Bag of Words (BoW)** is not the most suitable method, as it does not preserve **word order** or **sequence information**, resulting in a loss of sentence meaning. This limitation is also present, to varying degrees, in techniques such as **TF-IDF** and **Word2Vec**, which offer better semantic sensitivity but still lack an understanding of the full syntactic structure.

The main reason why **standard ANN architectures** are not well-suited for text-based tasks is that they process each word **independently** during both the feedforward and backpropagation stages. This means they fail to capture the **sequence information** and, consequently, the **semantic meaning** of entire sentences. As a result, the model is unable to understand context, word order, or dependencies between words — all of which are essential in natural language processing tasks.

Based on this limitation, researchers began exploring neural network-based solutions capable of handling **sequential data**. As outlined in this overview paper, this line of research led to the development of **Recurrent Neural Network (RNN)** architectures, which are specifically designed to model temporal dependencies and preserve word order — making them well-suited for natural language processing tasks.

Recurrent Neural Network (RNN)

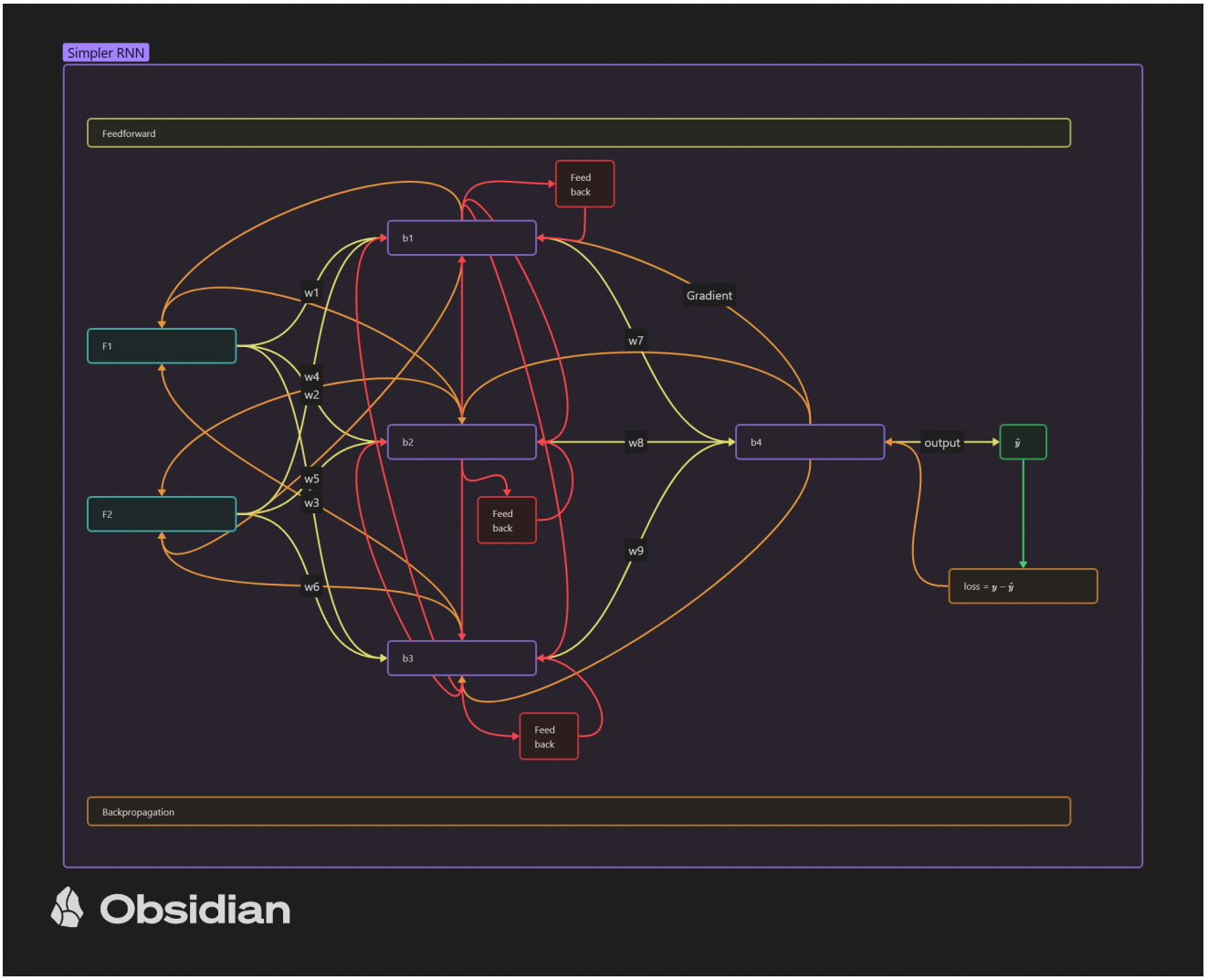


Figure 02: Simpler RNN

Unlike traditional ANNs, **Recurrent Neural Networks (RNNs)** incorporate a **feedback mechanism** that allows neurons in the hidden layer to share information across time steps. In other words, at each time step, the input is processed by the hidden layer, and the resulting state is **passed forward** and **used as context** for processing the next input.

This architecture enables the network to retain information from previous time steps, making it well-suited for handling **sequential data**. As a result, the loss of meaning that typically occurs in models that treat words independently (such as standard ANNs) can be mitigated, since RNNs allow hidden neurons to share temporal context with one another.

The ability of RNNs to process sequences has led to their widespread adoption in tasks such as language modeling (Mikolov et al., 2010), speech recognition (Graves et al., 2013), and machine translation (Sutskever et al., 2014). However, despite their conceptual elegance, early RNN implementations encountered significant training difficulties, most notably the **vanishing and exploding gradient problems** (Bengio, Simard, & Frasconi, 1994), which hindered their capacity to learn long-term dependencies.

To better understand the underlying architecture of a Recurrent Neural Network (RNN), let us consider a simplified example. Suppose we have a small corpus composed of three distinct text statements. After preprocessing steps such as tokenization and stop word

removal, we are left with a vocabulary of five unique words: `the`, `food`, `good`, `bad`, and `not`.

Text	Label (y)
the food is good	1
the food is bad	0
the food is not good	0

Despite the presence of six words in total, removing duplicates and stop words yields a vocabulary of size 5. Using vectorization methods such as **Bag of Words (BoW)**, **TF-IDF**, or even **Word2Vec** with static embeddings, we can transform these sentences into fixed-size vectors. For the sake of this example, we apply BoW to produce a document-term matrix with shape (3×5)(3 \times 5)(3×5), where each row corresponds to one sentence and each column to one term in the vocabulary.

This input matrix is passed to the RNN's input layer. Let's assume an RNN architecture with a **single hidden layer composed of 3 neurons** and a **single neuron in the output layer**. The forward propagation in this setup involves several weight matrices and bias terms:

1. **Input to Hidden Layer**:
   - The input matrix has shape (3×5)(3 \times 5)(3×5), and it is multiplied by a weight matrix of shape (5×3)(5 \times 3)(5×3), totaling **15 parameters**.
   - This multiplication produces an intermediate representation of shape (3×3)(3 \times 3)(3×3).
2. **Hidden Layer Computation**:
   - Each hidden neuron receives input from all 3 dimensions of the intermediate output, requiring a **3×33 \times 33×3** recurrent weight matrix (**9 weights**) and **3 bias terms**—one per neuron.
3. **Hidden to Output Layer**:
   - The hidden layer output (3×3)(3 \times 3)(3×3) is then multiplied by a weight vector of shape (3×1)(3 \times 1)(3×1), adding **3 additional weights**.
   - Finally, a single bias is applied at the output layer.

Summarizing, the total number of trainable parameters in this RNN architecture amounts to:

S1	S2	S3		Hidden layer		output layer	total
1,0,0,0,1	0,0,0,0,1	0,0,1,0,1		b1			
1,0,0,0,0	1,0,1,0,0	1,0,0,0,0	$(3X5).(5X3)$	b2	$(3X3).(3X1)$	b4	
0,1,0,0,0	0,1,0,0,1	1,1,0,0,1		b3			
0,1,1,0,0	0,1,0,0,0	0,1,0,0,1					

S1	S2	S3		Hidden layer		output layer	total
0,1,0,0,1	0,1,1,0,1	0,1,1,0,0					
			15 weights	9 weights and 3 bias		3 weights and 1 bias	31 param

Component	Parameters
Input-Hidden	$5 \times 3 = 15 \text{ weights}$
Recurrent Weights	$3 \times 3 = 9 \text{ weights}$
Hidden Biases	3 bias terms
Hidden-Output	$3 \times 1 = 3 \text{ weights}$
Output Bias	1 bias term
Total	**31 parameters**

This example illustrates the layered and recursive nature of RNNs, in which input sequences are processed in chunks and passed forward, with hidden states capturing temporal dependencies. It is worth noting that in practical applications, RNNs typically use **embedding layers** rather than BoW to preserve semantic relationships, and more complex architectures like **LSTM** or **GRU** are often employed to handle long-term dependencies.

Feedback notation and Feedforward Computation

The input sequence be:

$$X = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T), \quad \mathbf{x}_t \in \mathbb{R}^n \tag{1}$$

where T is the length of the sequence, and each x_T is an input vector (e.g., a word vector or a row from a document-term matrix).

At each time step, the **hidden state** $\mathbf{h}_t \in \mathbb{R}^m$ is computed recursively:

$$\mathbf{h}_t = \phi\left(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h\right) \tag{2}$$

Where:

- $\mathbf{W}_{xh} \in \mathbb{R}^{m \times n}$: input-to-hidden weights
- $\mathbf{W}_{hh} \in \mathbb{R}^{m \times m}$: recurrent weights (feedback loop)
- $\mathbf{b}_h \in \mathbb{R}^m$: bias vector
- ϕ: activation function (e.g., `tanh` or `ReLU`)

This **feedback loop**, represented by $\mathbf{W}_{hh}\mathbf{h}_{t-1}$, is the core of what distinguishes RNNs from feedforward models: it enables **temporal memory** and **state transitions** over time.

The output at each time step is computed as:

$$\mathbf{y}_t = \phi\left(\mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y\right) \tag{3}$$

Where:

- $\mathbf{W}_{hy} \in \mathbb{R}^{k \times m}$: hidden-to-output weights
- $\mathbf{b}_y \in \mathbb{R}^k$: output bias
- ϕ: activation function (e.g., `tanh` or `ReLU`)

There are typically two scenarios depending on the task:

- **Many-to-One**: Use only the last hidden state for prediction (e.g., sentiment classification):
- **Many-to-Many**: Compute an output at every time step (e.g., sequence labeling or translation).

The term $\mathbf{W}_{hh}\mathbf{h}_{t-1}$ introduces a recursive dependency that enables the network to **encode contextual information** from the past. This mechanism is essential for modeling data where the meaning or output depends on prior inputs—such as language, audio, or time series.

This architecture was first formalized in early works by **Rumelhart, Hinton, and Williams (1986)**, and later developed into modern RNN formulations by **Elman (1990)**, who emphasized the idea of “context units” that carry forward state information:

> *Elman, J. L. (1990). Finding structure in time. Cognitive Science, 14(2), 179–211.*

However, despite the simplicity of the model, training deep or long RNNs is notoriously difficult due to issues like **vanishing/exploding gradients**, which were analytically described by **Bengio, Simard, and Frasconi (1994)**:

> *Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. IEEE Transactions on Neural Networks, 5(2), 157–166.*

Suppose:

- Input dimension: $n = 5$
- Hidden layer size: $m = 3$
- Output size: $k = 1$

Compute the parameter sizes:

Component	Shape	Count
Input-Hidden Weights	$\mathbf{W}_{xh} \in \mathbb{R}^{3\times5}$	15

Component	Shape	Count
Recurrent Weights	$\mathbf{W}_{hh} \in \mathbb{R}^{3\times3}$	9
Hidden Bias	$\mathbf{b}_h \in \mathbb{R}^3$	3
Hidden-Output Weights	$\mathbf{W}_{hy} \in \mathbb{R}^{1\times3}$	3
Output Bias	$\mathbf{b}_y \in \mathbb{R}^1$	1
Total		**31**

Backpropagation throught time(BPTT) & Truncated BPTT

After the input $\mathbf{x}_t$ flows through one or more hidden layers, and the final hidden state is computed (let's call it $\mathbf{h}_t$, or specifically $\mathbf{h}_t^{(L)}$ for the **last hidden layer**), the network produces a prediction $\hat{y}$ at time step t as:

$$\hat{y}_t = \phi\left(\mathbf{h}_t^{(L)} \cdot \mathbf{W}_o + \mathbf{b}_o\right) \tag{4}$$

Where:

- $\hat{y}_t$ is the output (a scalar or vector depending on the task)
- $\mathbf{h}_t^{(L)} \in \mathbb{R}^{1\times m}$: the final hidden state
- $\mathbf{W}_o \in \mathbb{R}^{m\times k}$: weight matrix from hidden to output layer
- $\mathbf{b}_o \in \mathbb{R}^{1\times k}$: output bias
- ϕ is the **activation function**, e.g.,:
  - `softmax` for multi-class classification
  - `sigmoid` for binary classification
  - `identity` (no activation) for regression

Once the forward pass has produced the predicted output $\hat{y}$, the next step is to **evaluate the model's performance** by computing the **loss**, i.e., the discrepancy between the prediction $\hat{y}$ and the actual target y. This is done using a **predefined loss function**, which is chosen based on the nature of the task:.

Cross-Entropy Loss:

$$\mathcal{L} = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)] \tag{5}$$

where :

- (y_i): labeled values
- $(\hat{y}_i)$: predictions

This loss value $\mathcal{L}$ is used as the **objective function** to be minimized during training. In the next stage, **backpropagation through time (BPTT)** is applied to compute gradients of this loss with respect to the model parameters, and then update those parameters using **gradient descent** or one of its variants (e.g., Adam).

Since the objective is to atualize the weight matrices W_{xh}, W_{hh}, and W_{ho}, for this task is computed the partial derivatives in order to reach the minimum scalar which made the gradient become zero it means that the gradient vector is in some extreme minimum value of the loss surface.

Below are the derivations for $\mathbf{W}_{ho}, \mathbf{W}_{hh}, \text{ and } \mathbf{W}_{xh}$, as shown in Equations (6) through (12).

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{ho}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \frac{\partial \phi_o}{\partial \mathbf{W}_{ho}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \mathbf{h}_t \tag{6}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \frac{\partial \phi_o}{\partial \mathbf{h}_t} \cdot \frac{\partial \mathbf{h}_t}{\partial \phi_h} \cdot \frac{\partial \phi_h}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \mathbf{W}_{ho} \cdot \frac{\partial \mathbf{h}_t}{\partial \phi_h} \cdot \frac{\partial \phi_h}{\partial \mathbf{W}_{hh}} \tag{7}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{xh}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \frac{\partial \phi_o}{\partial \mathbf{h}_t} \cdot \frac{\partial \mathbf{h}_t}{\partial \phi_h} \cdot \frac{\partial \phi_h}{\partial \mathbf{W}_{xh}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \mathbf{W}_{ho} \cdot \frac{\partial \mathbf{h}_t}{\partial \phi_h} \cdot \frac{\partial \phi_h}{\partial \mathbf{W}_{xh}} \tag{8}$$

Since each hidden state $\mathbf{h}_t$ depends recursively on previous time steps, we unfold the gradient using the chain rule:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \mathbf{W}_{ho} \sum_{k=1}^{t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \cdot \frac{\partial \mathbf{h}_k}{\partial \mathbf{W}_{hh}} \tag{9}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{xh}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \mathbf{W}_{ho} \sum_{k=1}^{t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \cdot \frac{\partial \mathbf{h}_k}{\partial \mathbf{W}_{xh}} \tag{10}$$

Under a simplified approximation (assuming linear recurrence), the unfolded expressions can be written as:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \mathbf{W}_{ho} \sum_{k=1}^{t} (\mathbf{W}_{hh}^{\top})^{t-k} \cdot \mathbf{h}_k \tag{11}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{xh}} = \sum_{t=1}^{T} \frac{\partial \ell_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \mathbf{W}_{ho} \sum_{k=1}^{t} (\mathbf{W}_{hh}^{\top})^{t-k} \cdot \mathbf{x}_k \tag{12}$$

After computing the gradients (i.e., partial derivatives of the loss function $\mathcal{L}$ with respect to the weight matrices), the next step is to **update each weight matrix** in order to minimize the loss.

This is done using the **gradient descent update rule**, which adjusts the weights in the direction opposite to the gradient:

$$W_{\text{new}} = W_{\text{old}} - \eta \frac{\partial \mathcal{L}}{\partial W_{\text{old}}} \tag{13}$$

Where η is the learning rate, small scalar which controls the step by gradient descendent

This rule is applied **iteratively** over many training epochs, allowing the model to converge toward a minimum of the loss function.

Problems with RNN

Although Recurrent Neural Networks (RNNs) are powerful models for sequential data, they suffer from **several limitations** that hinder their performance on tasks involving long-range context.

Vanishing Gradient Problem

During training, RNNs use **Backpropagation Through Time (BPTT)** to adjust weights. However, when sequences are long, the gradients used to update weights can **shrink exponentially** as they are propagated backward through time.

As a result:

- Earlier layers (closer to the start of the sequence) receive **very small gradients**,
- Which means they **learn very slowly or not at all**,
- Causing the model to **"forget" earlier inputs** when making predictions.

This is known as the **vanishing gradient problem**, formally discussed by Bengio et al. (1994).

Text	Label (y)
the food is good	1
the food is bad	0
the food is not good	0

In these short sequences, RNNs can easily learn the relationships between words and the output. The dependency is local and the network does not have to remember words that are far apart.

Let’s consider a common text generation task:

```
Input: I like to play ___ .
```

The blanked word heavily depends on the previous phrase. This is manageable with a short sequence. But let’s increase the length:

```
Input: My name is Vinicius and I like sports, like football, tennis, and I
also like to make ____
```

In this case, predicting the correct word (e.g., *videos*, *music*, *projects*) requires remembering context from much earlier in the sentence. **As the sentence grows**, the RNN struggles to retain the relevant information — this is known as the problem of **long-term dependencies**.

RNNs theoretically can model dependencies across time steps. However, in practice, they **fail to capture long-range patterns** due to the combination of:

- Vanishing gradients,
- Limited memory (since hidden states are overwritten at every time step)

RNNs do **not scale well** with sequence length because their memory **decays** over time and this limitation motivated the development of more advanced architectures like LSTM and GRU, which introduce mechanisms to retain information over longer time range.

End-to-End Project (ANN) - Churn Modelling

ANN - Churn Classifier project

This project addresses a **binary classification problem** using a Kaggle dataset containing details about a bank's customers. The **target variable** is binary and indicates whether a customer has **churned** (i.e., left the bank) or continues to remain an active customer.

Feature	Description
`RowNumber`	Index of the row in the dataset (ranging from 1 to 10,000)
`CustomerId`	Unique identifier for each bank customer
`Surname`	Customer's last name
`CreditScore`	Customer's credit score
`Geography`	Country of residence
`Gender`	Gender of the customer (Male/Female)
`Age`	Age of the customer
`Tenure`	Number of years the customer has been with the bank
`Balance`	Current account balance of the customer
`NumOfProducts`	Number of banking products the customer is using
`Exited`	Target variable: 1 = customer churned; 0 = customer remained with the bank

Exploratory Data Analysis

This dataset is relatively simple to work with and is intended primarily for **educational purposes**. The first step in the analysis involved reading the file and inspecting the initial characteristics of the data.
Subsequently, the first three features — `RowNumber`, `CustomerId`, and `Surname` — were dropped, as they do not contribute meaningful information to the classification task. These columns function primarily as **identifiers or indices**, and therefore offer no predictive value for the model.

Two categorical features in the dataset — `Gender` and `Geography` — required encoding before being used as input for the model.

The first feature, `Gender`, was encoded using Scikit-learn's `LabelEncoder` method, which assigns an integer value to each unique class starting from zero. As a result, the original values `"Male"` and `"Female"` were transformed into `1` and `0`, respectively.

The second categorical feature, `Geography`, contains three unique classes. In this case, using `LabelEncoder` would introduce a **false ordinal relationship**, since the third class would be assigned a value of `2`, implying magnitude where none exists. Therefore, **One-Hot Encoding** was used as a more appropriate method. This approach creates **three new binary columns**, one for each unique class in `Geography`, with values of `0` or `1` indicating the presence of each category.

Saving and Loading Models with `pickle`

In Python, the `pickle` module is used for **serialization** and **deserialization** of Python objects. Serialization refers to the process of converting a Python object (such as a trained machine learning model) into a byte stream, which can be saved to a file or transferred over a network. Deserialization is the reverse process — converting the byte stream back into the original Python object.

Why use `pickle` in Machine Learning?

In machine learning workflows, it is common to serialize trained models so that they can be:

- **Saved** and reused without retraining,
- **Shared** across systems or environments,
- **Integrated** into production applications for inference.

Using `pickle` allows you to store your trained model in a `.pkl` file and load it later exactly as it was.

Build a ANN Model

A simple **sequential neural network** was implemented with **two hidden layers**, each using the **ReLU activation function**. The **output layer** consists of a **single neuron** with a **sigmoid activation function**, suitable for binary classification tasks.

The **sigmoid function** is a nonlinear activation function with an “S”-shaped curve that produces output values in the range $[0, 1]$:

$$f(z) = \frac{1}{1 + e^{-z}} \tag{14}$$

- **Properties**:
  - Smoothly maps input values into the $[0, 1]$ interval.
- **Disadvantages**:
  - **Vanishing gradient**: For extreme values of z , the gradient approaches zero, which can significantly slow down learning in deep networks.

ReLU returns z directly for positive values and 0 for negative values:

$$f(z) = \max(0, z) \tag{15}$$

- **Properties**:
  - Introduces non-linearity and is efficient for deep networks.
  - Easily differentiable (except at the point $z = 0$).
- **Disadvantages**:
  - **Dead Units**: If many neurons output $z < 0$, they may stop learning.

Layer (type)	Output Shape	Param
dense (Dense)	(None, 64)	83
dense_1 (Dense)	(None, 32)	2,08
dense_2 (Dense)	(None, 1)	3

Model params

Was chosen the Adaptative Moment Estimaton (Adam) as a optimazer, one of the most popular optimazers used in neural networks architectures. The method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients Kingma, D. P., & Ba, J. (2015). Adam: A Method for Stochastic Optimization

The **cost function** (or loss function) in a machine learning algorithm is a metric used to evaluate how well the model is performing.
It measures the difference between the predictions made by the model and the actual values from the training data.
The goal of model training is to **minimize this cost function** by adjusting the model’s parameters so that the predictions become as close as possible to the true values.

An optimization algorithm, such as **Gradient Descent**, uses the cost function to iteratively update the model's parameters, aiming to reduce the loss and consequently improve the model's prediction accuracy.

For this output was chose the binary cross-entropy.

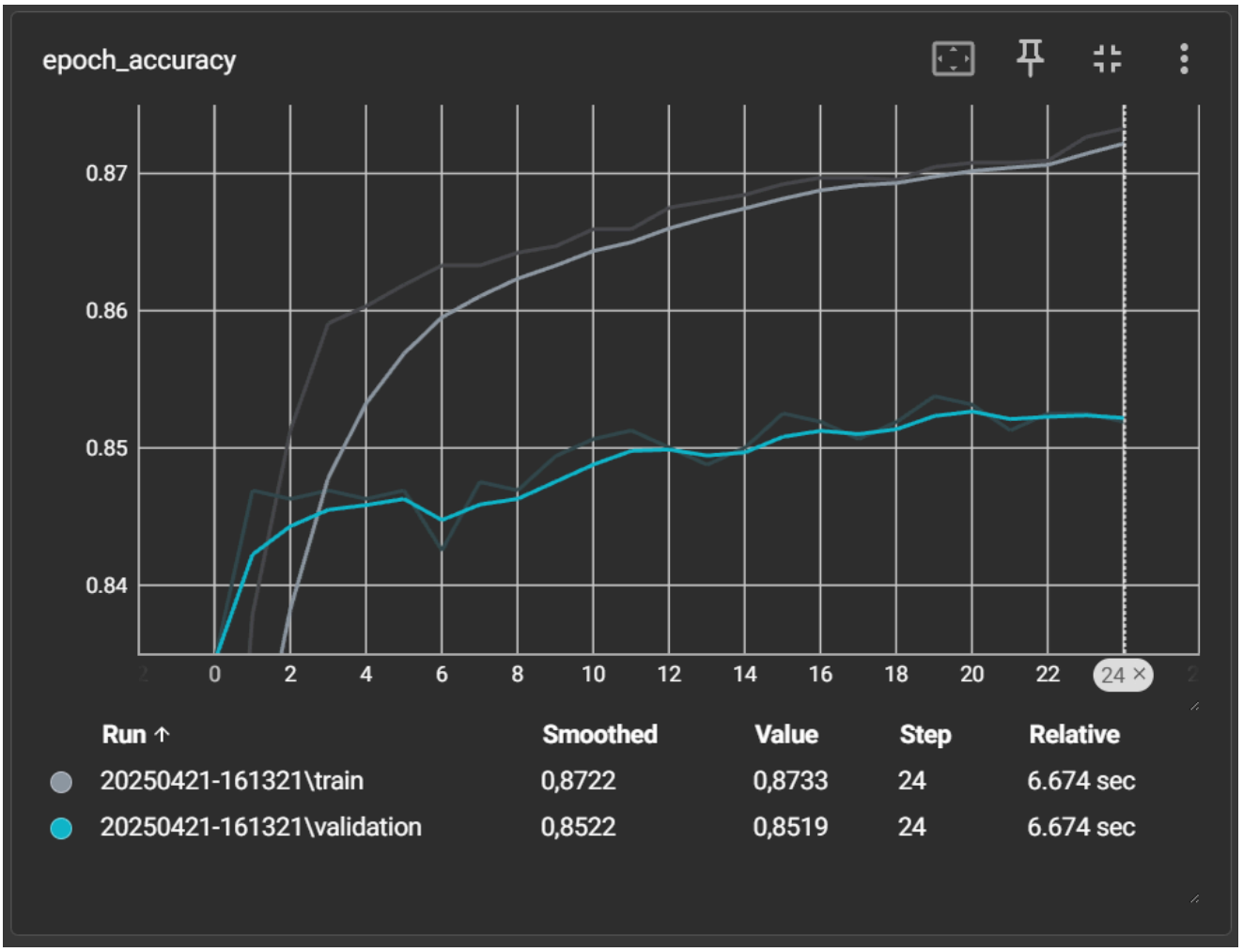


Figure 03: Accuracy per epoch

The model was initially configured to train for **100 epochs**. However, the **Early Stopping** technique was applied to halt training when the **validation loss** stopped decreasing for **10 consecutive epochs**. This method also ensured that the best model weights — those corresponding to the lowest validation loss — were retained.
As a result, training concluded after only **25 epochs**, achieving approximately **85% accuracy** on the validation set.

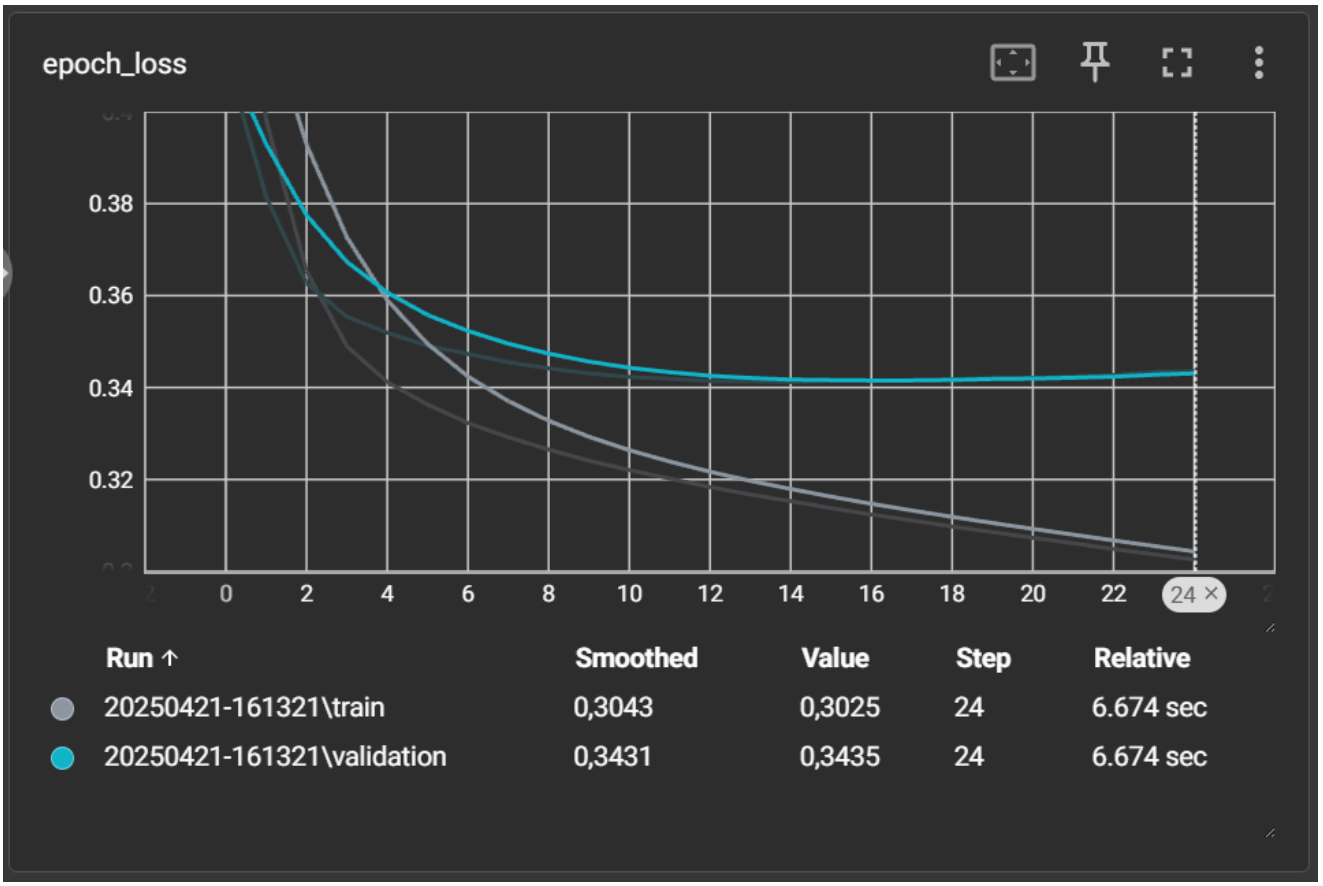


Figure 04: Loss per eooch

Determining the Optimal Number of Hidden Layers and Neurons in an Artificial Neural Network (ANN)

Choosing the optimal architecture for an Artificial Neural Network can be challenging and often requires empirical experimentation. However, several guidelines and strategies can help in making more informed decisions:

- **Start Simple:** Begin with a simple architecture (e.g., one or two hidden layers) and increase complexity only if necessary;
- **Grid Search/Random Search:** Use these techniques to explore various combinations of layers and neurons systematically or stochastically;
- **Cross-Validation:** Apply cross-validation to evaluate the generalization performance of different architectures;
- **Heuristics an Rules oh Thumb:** Some heuristics and empirical rules can provides starting points, such as:

- The number of neurons in a hidden layer should typically lie between the size of the input layer and the output layer;
- It is common practice to start with one to two hidden layers, which are sufficient for most problems unless the task is highly complex.

Text To Vector Representation

After preprocessing, the cleaned text was converted into numerical representations using vectorization strategies. In natural language processing (NLP), **word embeddings** refer to the representation of words as real-valued vectors. These vectors aim to capture the semantic meaning of words, such that words that are close in the vector space are expected to be similar in meaning (Mikolov et al., 2013a).

Word2Vec is a widely used word embedding technique introduced in 2013. It employs a shallow neural network to learn word associations from a large corpus. Once trained, the model can identify synonyms or suggest relevant words to complete a sentence (Mikolov et al., 2013b). As the name suggests, Word2Vec maps each distinct word to a unique vector of continuous values that captures its contextual meaning in the corpus.

In this work, a pre-trained embedding model developed by Google was used. This model was trained on a portion of the Google News dataset, comprising approximately 100 billion words. It provides 300-dimensional vectors for 3 million words and phrases, which were extracted using a data-driven approach described in *Distributed Representations of Words and Phrases and their Compositionality* (Mikolov et al., 2013b).

`Word2Vec` is a technique designed for natural language processing, originally introduced in 2013. `Word2Vec` is a neural network algorithm capable of associating words within a large corpus. Once the model is trained, it becomes capable of identifying synonyms and predicting the next word in an incomplete sentence. As the name suggests, `Word2Vec` represents each word through a numerical vector.

Given a `corpus`, the `unique words – vocabulary` are extracted and numerically mapped to specific indices. The number of indices corresponds to the dimensionality of the vector representing each word. As illustrated in the table 03

	boy	**girl**	**king**	**queen**	**apple**	**mango**
gender	-1	1	-0.92	0.93	0.1	0.23
royal	0.01	0.002	0.95	0.96	0.0002	0.0002
age	0.03	0.003	0.75	0.76	0.001	0.003
food	0.00001	0.00001	0.00001	0.00001	0.96	0.97
...	...	...	...	...	...	...

The indices represent the features, while the vectors correspond to the vertical columns assigned to each word.

In this way, it is possible—through mathematical operations between vectors—to derive new words, such as:

$$King - Boy + Queen = Girl \tag{16}$$

Cosine Similarity

This type of reasoning is made possible through **cosine similarity**, one of the most widely used metrics to measure the similarity between two vectors. Cosine similarity evaluates the cosine of the angle between two vectors in a multi-dimensional space. The closer the cosine value is to 1, the more similar the vectors are, which often reflects semantic similarity between the corresponding words (Mikolov et al., 2013; Turney & Pantel, 2010).

Vector similarity refers to the degree of closeness or relatedness between two vectors in a multi-dimensional space. In the context of word embeddings, it quantifies how semantically related two words are, based on the distance or angle between their respective vectors (Jurafsky & Martin, 2021).

Figure 05: Vector Similarity

$$Similarity = cos(\theta) = \frac{A \cdot B}{\parallel A \parallel \parallel B \parallel} \tag{17}$$

Where:

$$\vec{A} \cdot \vec{B} \text{ It is the dot product between the vectors.} \tag{18}$$

$$\parallel A \parallel \parallel B \parallel \text{ They are the norms (or magnitudes) of the vectors.} \tag{19}$$

For example, assuming the angle between two vectors is 45°, the cosine similarity is:

$$\cos(45^\circ) = 0.7071 \tag{20}$$

Thus, the cosine distance would be:

$$1 - \cos(45^\circ) = 1 - 0.7071 = 0.291 \tag{21}$$

A cosine distance close to 0 indicates that the vectors are pointing in nearly the same direction, suggesting a high degree of semantic similarity — often interpreted as the words being synonyms or used in similar contexts.

Advantages of Word2Vec

1. **Dense Matrix**
   - Solves the overfitting problem caused by sparse matrices.
2. **Captures Semantic and Contextual Information**
   - Enables cosine similarity-based comparisons.
3. **Vocabulary Size**
   - The embedding matrix has a fixed dimensionality — as seen in the Google Word2Vec model.
4. **Out of Vocabulary (OOV)**
   - Due to its ability to capture semantics and context, unlabeled or unseen words can be approximated using the context window of known words.

Two of the most widely adopted architectures for training the Word2Vec model are **Continuous Bag of Words (CBOW)** and **Skip-Gram** (Mikolov et al., 2013).

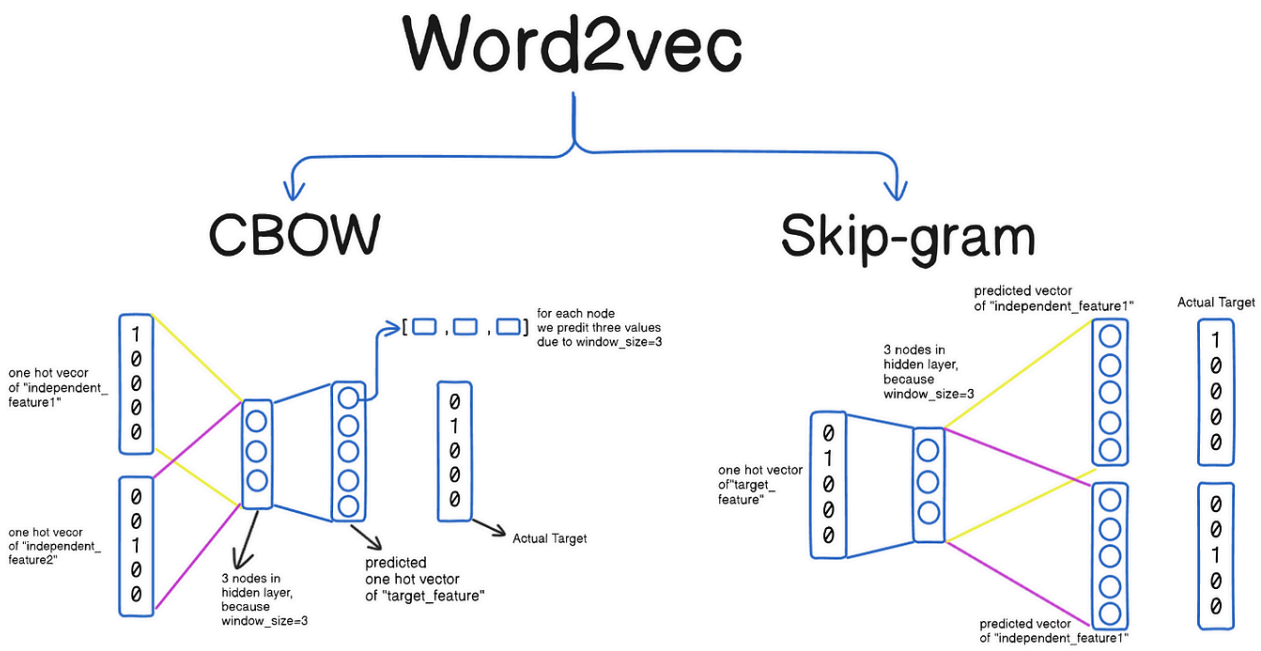


Figure 06: Word2Vec

CBOW – Continuous Bag of Words

The CBOW model predicts a target word based on its surrounding context. The input is formed by a set of context words within a fixed-size window, and the goal is to predict the word located at the center of this window.

The corpus is segmented using a sliding window mechanism (`window size`), which defines the number of words on either side of the target. Each word is initially transformed into a

one-hot encoded vector — a sparse representation where only one position corresponding to the word's index in the vocabulary is set to 1, and all others are 0.

These one-hot vectors serve as inputs to a fully connected neural network. The hidden layer has a dimensionality equal to the predefined embedding size and is responsible for converting the sparse one-hot inputs into dense, real-valued vector representations.

The output layer produces a vector with dimensionality equal to the vocabulary size. A `softmax` activation function is applied to compute the probability distribution over all possible words, estimating which word is most likely to be the target (central) word.

The model is trained using a **feedforward** pass followed by weight updates computed through **backpropagation**, minimizing a loss function (typically cross-entropy).

CBOW is known for its efficiency in handling large corpora and tends to perform better when the training data is abundant and the vocabulary is large (Mikolov et al., 2013).

$$\text{iNeuron company is related to data sceicne} \tag{22}$$

$$\text{window size} = 5 \tag{23}$$

$$\text{center and output: is, input : iNeuron company related to} \tag{24}$$

$$\text{center and output: related, input: company is to data} \tag{25}$$

$$\text{center and output: to, input: is related data sceience} \tag{26}$$

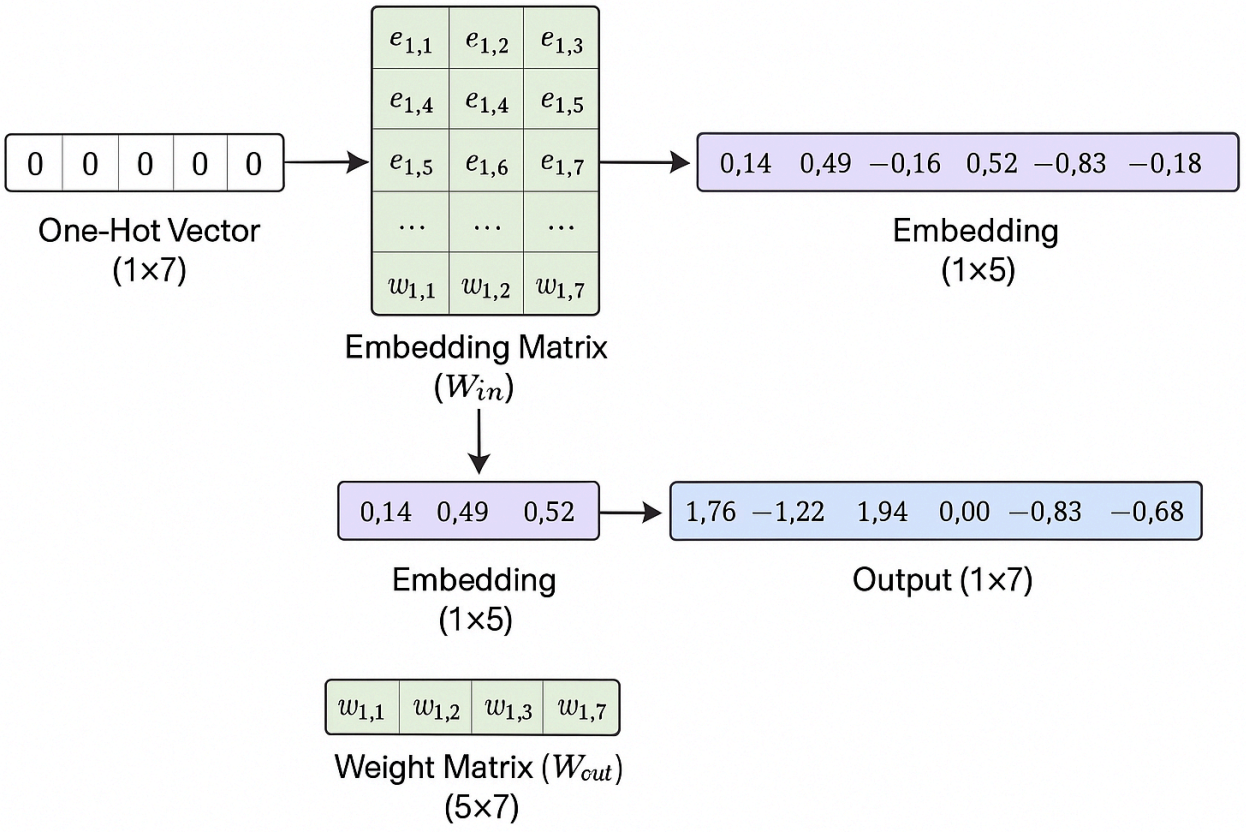


Figure 07: CBOW ANN description

Step	Shape	Description
One-hot (input)	(1 x 7)	Selects the word from the vocabulary
`W_in` (embedding)	(7 x 5)	Each row = vector representation of a word
Embedding result	(1 x 5)	Vector of the selected word
`W_out` (output weights)	(5 x 7)	Each column = score for each word in the vocabulary
Final output	(1 x 7)	Logits (unnormalized scores) for all vocabulary words

Table 03: Matrix Size Steps

Skip-Gram

The **Skip-Gram** model is conceptually the inverse of the **CBOW** architecture. While CBOW uses context words as input to predict the central word, Skip-Gram takes a single **central word** as input and attempts to predict the surrounding **context words** (Mikolov et al., 2013).

Model	Input	Output
CBOW	Context words	Central word
Skip-Gram	Central word	Context words

table 04: CBOW and Skip-Gram differences

Given a central word, the model learns to maximize the probability of correctly predicting words within a defined context window (e.g., two words before and after the target word). This approach is particularly effective for capturing **semantic relationships** in sparse datasets or when dealing with rare words, as it provides multiple training samples per input word (Rong, 2014).

The input word is one-hot encoded and passed through an embedding matrix (`W_in`) to produce a dense vector representation. This vector is then multiplied by a second weight matrix (`W_out`) to generate multiple outputs — one for each expected context word. Each output is compared against the actual context using a `softmax` function and a loss function such as cross-entropy, and the model is trained using **backpropagation**.

Skip-Gram is especially useful when the goal is to learn high-quality representations for infrequent words, due to its ability to learn from a single input and multiple outputs across a wide range of contexts (Goldberg & Levy, 2014).

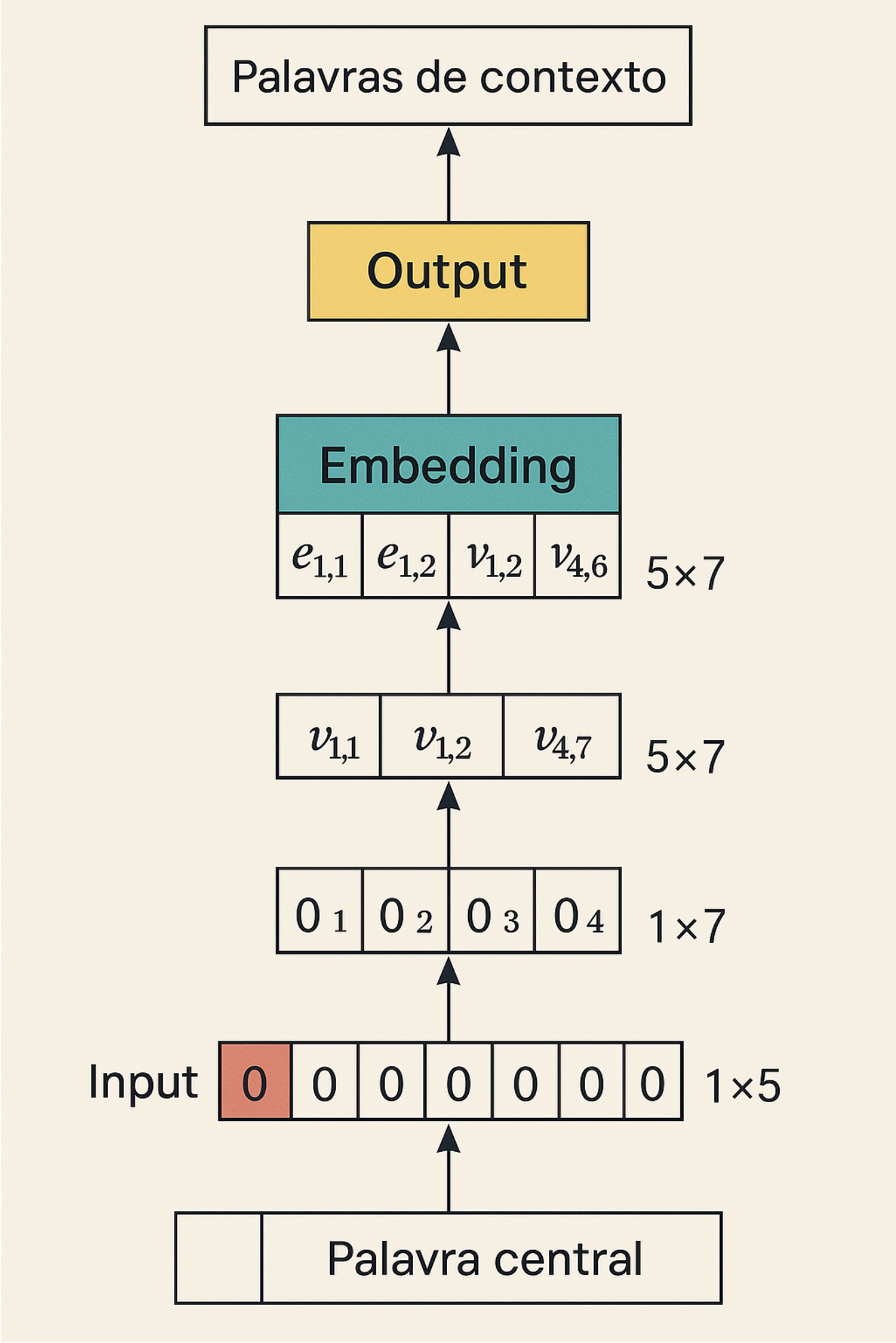


Figure 08: Skip-Gram Steps

Step	Shape	Description
One-hot (input)	(1 x 7)	Central word encoded as a one-hot vector
`W_in` (embedding)	(7 x 5)	Each row represents the dense (embedding) vector of a word
Embedding result	(1 x 5)	Embedding vector of the central word
`W_out` (output weights)	(5 x 7)	Each column represents a score for each vocabulary word
Final output	(1 x 7)	Logits (unnormalized scores) to predict each vocabulary word (one at a time)

table 05: Skip-Gram matrix steps

CBOW vs Skip-Gram

- **CBOW** is more efficient for smaller corpora, as it averages the context — reducing noise and improving generalization.
- **Skip-Gram** performs better on large corpora and is particularly effective at learning representations for rare words, since it generates multiple output samples per central word.

1. **Increase training data**:
   - The larger the dataset, the higher the model's potential accuracy.
2. **Increase** `window size`:
   - Expanding the `window size` increases the amount of contextual information and, consequently, can lead to higher-dimensional **embedding vectors**.

End-to-End RNN Problem - Imdb Dataset

This project is based on Imdb movie Reviews Dataset structed by text reviews as a main feature and a positive ou negative target output provide a set of 25,000 highly polar movie reviews for training, and 25,000 for testing. There is additional unlabeled data for use as well. Raw text and already processed bag of words formats are provided. For that task will be builded a simple RNN architecture to predict the classes, traning and deploy the entire project.

$$\text{Training data shape: } (25000,), \text{ training labels shape: } (25000,)$$

$$\text{Testing data shape: } (25000,), \text{ testing labels shape: } (25000,)$$

Word Embeddings

Before we begin, let's begin with a **simple and practical example** of how to create word embeddings using the **Sequential API** in the **Keras** library. Consider the following list of

sample sentences:

$$\text{sent} = \begin{bmatrix} \text{'the glass of milk'}, \\ \text{'the glass of juice'}, \\ \text{'the cup of tea'}, \\ \text{'I am a good boy'}, \\ \text{'I am a good developer'}, \\ \text{'understand the meaning of words'}, \\ \text{'my name is rondon'}, \\ \text{'my life is very good'}, \\ \text{'my life is bad'} \\ \end{bmatrix} \tag{27}$$

These sentences will be used to demonstrate the preprocessing steps, tokenization, and how to pass the resulting sequences into an **Embedding layer** within a neural network.

Unlike traditional one-hot encoding, which transforms each word into a high-dimensional sparse vector with a vocabulary-sized dimension (e.g., 10,000) and only a single `1` among zeros, the `tensorflow.keras.preprocessing.text.one_hot()` method provides a more **compact and efficient representation**.

Instead of generating a full binary vector, this method returns the **index position** where the `1` would have been, effectively mapping each word to an **integer value** using a hashing function. This approach significantly reduces memory usage and avoids the inefficiency of managing large sparse matrices during preprocessing.

After applying this technique, the same list of sentences is transformed into the following **numerical representation**:

$$\begin{bmatrix} [914, 7786, 7674, 3473], \\ [914, 7786, 7674, 5858], \\ [914, 5780, 7674, 7146], \\ [4639, 42, 6084, 8211, 3672], \\ [4639, 42, 6084, 8211, 7236], \\ [5761, 914, 3651, 7674, 2477], \\ [3402, 1857, 4223, 2162], \\ [3402, 6721, 4223, 3133, 8211], \\ [3402, 6721, 4223, 399]] \end{bmatrix} \tag{28}$$

One of the main limitations of this approach, however, is that the resulting sequences have **variable lengths**, depending on the number of words in each sentence. Neural networks require inputs of **uniform length**, so this variability must be addressed. Without resolving it—typically through padding or truncation—it is not possible to train any standard neural network architecture.

To address this issue, the `pad_sequences()` method from the **Keras preprocessing module** was used to **standardize the length of all input sentences**. This function applies **pre-padding**, which adds zeros at the beginning of each sequence based on a predefined

`maxlen` parameter. This ensures that all sequences have the same length, allowing them to be **properly processed by neural network models**.

The following matrix illustrates the padded version of the previously encoded sentences:

$$\begin{gathered}
[0,0,0,0,914,7786,7674,5858], \\
[0,0,0,0,914,5780,7674,7146], \\
[0,0,0,4639,42,6084,8211,3672], \\
[0,0,0,4639,42,6084,8211,7236], \\
[0,0,0,5761,914,3651,7674,2477], \\
[0,0,0,0,3402,1857,4223,2162], \\
[0,0,0,3402,6721,4223,3133,8211], \\
[0,0,0,0,3402,6721,4223,399]
\end{gathered} \tag{29}$$

This transformation prepares the data for input into an **embedding layer**, ensuring that the model receives fixed-length input sequences.

When an `Embedding` **layer** is created in Keras using the syntax `Embedding(vocab_size, dim)`, it initializes a **weight matrix** of shape ($\text{vocab\_size} \times \text{dim}$. For instance, if `vocab_size = 10,000` and `dim = 10`, the layer creates a matrix of size $10,000 \times 10$, where each row corresponds to a vector representation of a word in the vocabulary. This matrix is initially filled with small random values, which are updated during the training process.

When a word index is passed into the Embedding layer (e.g., 7786 corresponding to the word “glass”), the layer performs a **lookup operation**, returning the row in the weight matrix that corresponds to that index. Conceptually, this is equivalent to a dictionary lookup, where the index acts as the key, and the resulting vector is the value. Mathematically, this operation can be interpreted as a **matrix multiplication** between a one-hot encoded vector (with a single `1` at the word’s index and `0`s elsewhere) and the weight matrix. The result is the same as selecting the corresponding row of the embedding matrix, but it is implemented more efficiently without explicitly constructing one-hot vectors.

During training, the weights in this embedding matrix are updated via **backpropagation**. The network learns to organize this vector space such that **words appearing in similar contexts** are mapped to vectors that are **close together**. This means the embeddings capture meaningful **semantic relationships** between words. For example, the vectors for "king" and "queen" may become similar, and the **difference** between them could represent a high-level concept such as **gender**.

This embedding technique is **much more efficient** than traditional one-hot encoding. Instead of representing each word as a sparse vector of size `vocab_size` (e.g., 10,000 dimensions, mostly filled with zeros), embeddings allow each word to be represented as a **dense vector** of fixed size `dim` (e.g., 10 dimensions), containing values that the model learns to optimize. This not only improves computational efficiency but also allows the model to learn and generalize more effectively from the data.

$$
\begin{bmatrix}
[-0.03325985, -0.03622857, -0.02012959, 0.03779368, 0.00585018, 0.01601349, 0.02976579, -0 \\
[-0.03325985, -0.03622857, -0.02012959, 0.03779368, 0.00585018, 0.01601349, 0.02976579, -0 \\
[-0.03325985, -0.03622857, -0.02012959, 0.03779368, 0.00585018, 0.01601349, 0.02976579, -0 \\
[0.04380025, 0.0300787, -0.03908987, -0.04970375, -0.02178363, -0.0267205, 0.03519152, 0. \\
[0.0450046, 0.01269367, 0.04119075, 0.01686937, 0.03329903, 0.02941674, -0.03357425, -0.0 \\
[0.02338025, 0.0171949, 0.00778427, -0.00195087, -0.01593274, 0.00360554, 0.04338589, -0.0 \\
[0.02657955, 0.03887156, 0.00596725, -0.02162695, -0.02136395, 0.03807999, -0.0375643, -
\end{bmatrix}
$$

The representation above corresponds to the **embedding output** of the first sentence, which now has a shape of $(1, 8, 10)$. This means the sentence consists of **8 words**, and each word is represented as a **10-dimensional vector**. The first dimension (1) refers to the **batch size**, while the second and third dimensions represent the **sequence length** and **embedding size**, respectively. These dimensions are determined by the `maxlen` parameter and the **embedding vector size** defined during model construction.

The model defined as:

```
model = Sequential()
model.add(Embedding(vocab_size, dim, input_length=sent_length))
model.compile('adam', 'mse')
```

Layer (type)	Output Shape	Param
embedding (Embedding)	?	0 (unbuilt

Embedding model

Embedding Layer

The embedding layer is parameterized by a matrix:

$$\mathbf{W} \in \mathbb{R}^{V \times D}$$

Where:

- V is the **vocabulary size**
- D is the **embedding dimension**

Given an input sequence $\mathbf{x} = [x_1, x_2, \ldots, x_T]$, where each $x_t \in \{1, \ldots, V\}$ is a word index and $T = \text{sent\_length}$, the embedding layer performs a lookup operation:

$$\mathbf{Z} = [\mathbf{W}_{x_1}, \mathbf{W}_{x_2}, \ldots, \mathbf{W}_{x_T}] \in \mathbb{R}^{T \times D}$$

So each input sequence becomes a matrix $\mathbf{Z}$ of shape $T \times D$, where each row is a word vector.

Model Output

Because no additional layers are added after the embedding, the **output of the model is simply the embedding matrix $\mathbf{Z}$** for each input sequence.

Compile Function

The model is compiled with:

```
model.compile(optimizer='adam', loss='mse')
```

- The **loss** is Mean Squared Error (MSE), commonly used for regression tasks.
- The **optimizer** is Adam, which performs gradient-based updates using first and second moment estimates of the gradient, as detailed by Kingma & Ba (2015).

Formally, the **MSE loss** between predicted output $\hat{\mathbf{y}}$ and true output $\mathbf{y}$ is:

$$\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

Numerical Representation Example

Given the sequence-to-index transformation:

$$\begin{bmatrix} 914, 7786, 7674, 3473 \\ 914, 7786, 7674, 5858 \\ 914, 5780, 7674, 7146 \\ 4639, 42, 6084, 8211, 3672 \\ \dots \end{bmatrix}$$

Each index x_t is mapped to its corresponding embedding vector $\mathbf{w}_{x_t} \in \mathbb{R}^D$, resulting in:

$$\mathbf{Z} \in \mathbb{R}^{B \times T \times D}$$

Where:

- B: number of sentences (batch size)
- T: sentence length
- D: embedding dimension

For instance, if $T = 8$ and $D = 10$, one example becomes:

$$\mathbf{Z} \in \mathbb{R}^{1 \times 8 \times 10}$$

This allows the model to represent sequences in a continuous vector space, as introduced by Mikolov et al. (2013).

The defined Keras model utilizes a single `Embedding` layer to transform sequences of token indices into continuous vector representations, formalized by a weight matrix $\mathbf{W} \in \mathbb{R}^{V \times D}$,

where V is the vocabulary size and D is the dimensionality of the embedding space. Given an input sentence $\mathbf{x} = [x_1, x_2, \ldots, x_T]$, the model performs a lookup operation such that each token x_t is replaced by its corresponding row $\mathbf{W}_{x_t}$ in the embedding matrix, producing an output tensor $\mathbf{Z} \in \mathbb{R}^{T \times D}$.

This transformation enables the encoding of discrete linguistic units into a dense, continuous space, facilitating the learning of semantic relationships between words, as originally proposed in Mikolov et al. (2013) through the Word2Vec architectureWord2Vec paper. The model is compiled using the Adam optimizer, an adaptive stochastic gradient descent algorithm that computes individual learning rates from estimates of first and second moments of the gradients, allowing efficient convergence in high-dimensional parameter spaces (Kingma & Ba, 2015)ADAM A METHOD FOR STOCH….

The chosen loss function, Mean Squared Error (MSE), evaluates the average of squared differences between the predicted and target vectors, making it suitable for regression-style learning tasks where the goal is to approximate continuous vector outputs. Overall, the model mathematically captures the transition from symbolic input to vectorial encoding while optimizing embedding representations through backpropagation using Adam, rooted in the theoretical foundations of modern representation learning.

Inspect Imdb keras datasets

The IMDB dataset provided by Keras contains movie reviews that have already been **preprocessed into Bag-of-Words format**, where each word is represented by an **integer index** based on a predefined **vocabulary size of 10,000**. Along with the dataset, a dictionary mapping of words to their corresponding indices is provided, which makes it possible to **decode the reviews** back into their textual form.

Before the data can be used as input for a neural network, the **sequence lengths must be standardized**. This is accomplished using the `pad_sequences` method, where the **maximum sequence length** is set to **500**. As a result, all reviews are either truncated or padded to ensure uniform shape, making them suitable for processing in the model pipeline.

```
model= Sequential()

model.add(Embedding(max_features, 128, input_length=max_length))

model.add(SimpleRNN(128, return_sequences=True,activation="relu",
kernel_initializer="he_normal"))

model.add(SimpleRNN(128, return_sequences=True,activation="relu",
kernel_initializer="he_normal"))

model.add(SimpleRNN(128, return_sequences=False,activation="relu",
```

```
kernel_initializer="he_normal"))

model.add(Dense(1, activation='sigmoid'))
```

This architecture begins with an **Embedding layer**, which converts input word indices into dense vectors of dimension **128**. The input length is set to `max_length`, ensuring compatibility with the padded sequences.

The model then stacks **three** `SimpleRNN` **layers**, each with **128 units**, matching the embedding dimension. The parameter `return_sequences=True` is used in the first two RNN layers to ensure that each layer outputs the **full sequence of hidden states** — one for each time step — so that subsequent RNN layers can process the **entire temporal sequence**.

> If `return_sequences` is set to `False` in an intermediate RNN layer, the next RNN would receive only a single vector (the final hidden state), thereby **losing access to time-series structure** in the data.

The **last RNN layer** uses `return_sequences=False`, meaning it outputs only the final hidden state. This is a common approach when the model performs **sequence classification**, as it allows the classification decision to be based on the **entire processed input** through the final state.

The final `Dense` layer consists of a **single neuron with a sigmoid activation function**, producing a probability value for binary classification.

Setting	Output Shape	Purpose
`return_sequences=False` (default)	`(batch_size, hidden_units)`	Only the final output (last time step)
`return_sequences=True`	`(batch_size, time_steps, hidden_units)`	Full sequence of outputs, one for each time step

Initializer weights

Another extremely important hiperparameter is the initializer weights, the bottom line of the model training process especially in deep neural networks architectures where will be ajust the wights interactively in order to minimalize the error.

when has a inadequate inicializing the algorithm could have a very slow training time or even never converge into a local minimum, may cause **vanishing or exploding gradients** during backpropagation, which hinders the learning process. Initializing all weights with the same value can deal with **symmetry**, where neurons learn the same features, limiting the network's capacity.

Improper initialization can lead to issues like vanishing or exploding gradients, which hinder the learning process. Two seminal works that address this challenge are by LeCun et al. (1998) and Glorot & Bengio (2010), introducing the LeCun and Xavier (also known as Glorot) initialization methods, respectively.

LeCun initialization is designed to maintain the variance of activations across layers, particularly effective when using activation functions like the hyperbolic tangent (tanh). In this approach, weights are initialized with a variance of $\frac{1}{n_{\text{in}}}$, where n_{in} is the number of input connections to a neuron. This helps in preserving the scale of the input signal throughout the network, facilitating efficient learning.

Glorot and Bengio introduced the Xavier initialization to address the challenges of training deep feedforward neural networks. This method aims to keep the variance of activations and gradients consistent across layers. Weights are initialized with a variance of $\frac{2}{n_{\text{in}}+n_{\text{out}}}$, balancing the number of input and output connections. This strategy is particularly beneficial when using activation functions like tanh or sigmoid.

Initialization Method	Recommended Activation Function(s)	Variance Formula (Normal Distribution)
LeCun	SELU	$\frac{1}{n_{\text{in}}}$
Xavier (Glorot)	Tanh, Sigmoid, Softmax	$\frac{2}{n_{\text{in}}+n_{\text{out}}}$
He (Kaiming)	ReLU, Leaky ReLU, ELU	$\frac{2}{n_{\text{in}}}$

Table 07: Kernel initialization

Model Evaluate

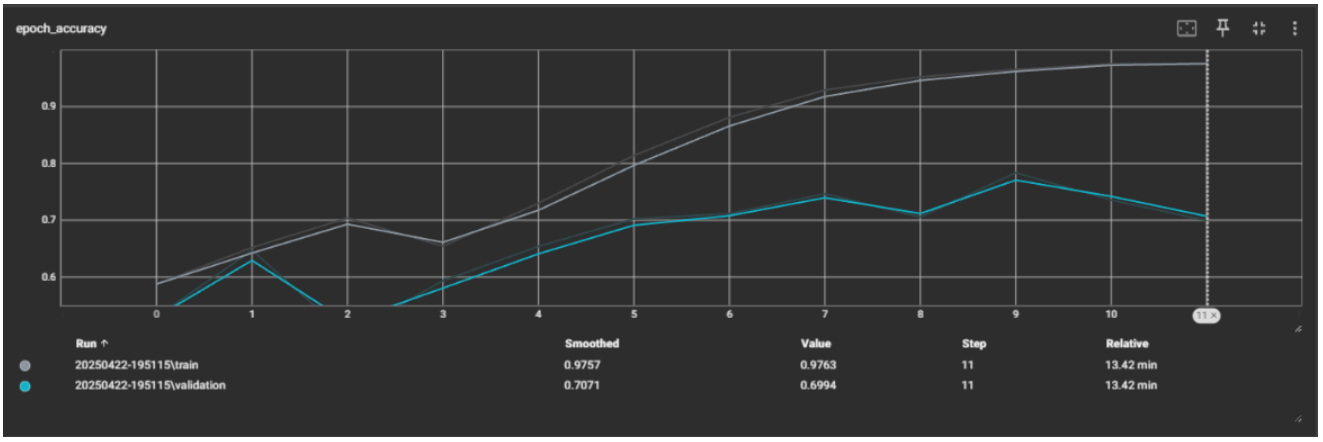


Figure 09: Accuracy per epoch

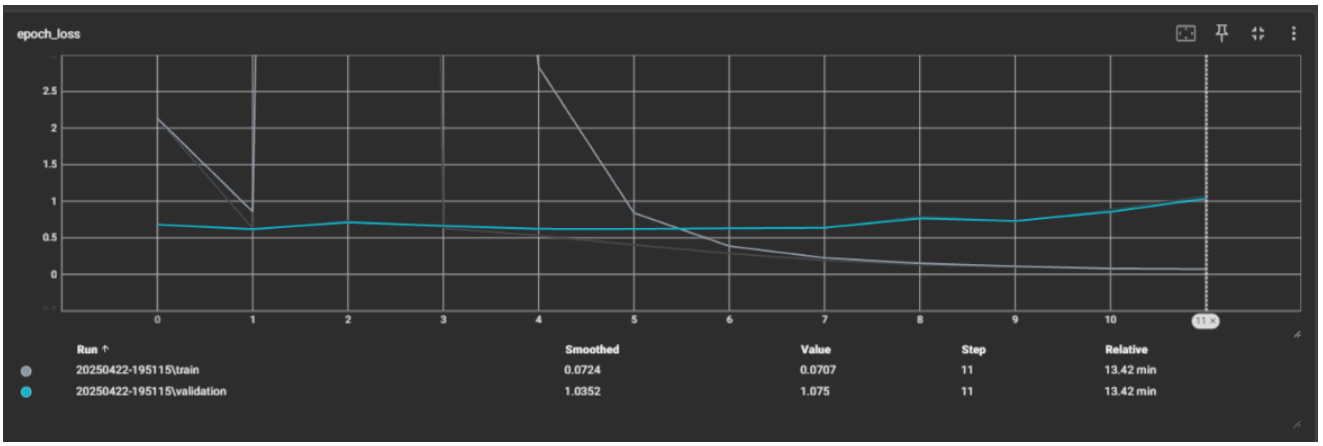


Figure 10: Loss per epoch

The model achieved high performance on the training data, with an **accuracy of 0.9734** and a **loss of 0.0783**. However, it did not generalize as well to the validation data, where performance dropped to a **validation accuracy of 0.7836** and a **validation loss of 0.7229**. The model's performance declined even further on the test dataset, achieving an **accuracy of 0.6453** and a **loss of 0.6122**. These results indicate that the model is likely **overfitting** the training data and failing to generalize effectively to unseen examples.

An important observation lies in the **validation loss**, which is nearly **ten times greater** than the training loss. This significant gap suggests that the **gradients may have begun to explode**, leading to unstable learning during validation. One possible cause of this behavior is the use of the **ReLU activation function**, which, while effective and commonly used, can lead to issues such as **exploding gradients** or **dead neurons**, particularly in deep or recurrent architectures without proper regularization or gradient clipping.

```
model= Sequential()

model.add(Embedding(max_features, 128, input_length=max_length))

model.add(SimpleRNN(128, return_sequences=True,activation="tanh",
kernel_initializer="glorot_uniform"))

model.add(SimpleRNN(128, return_sequences=False,activation="tanh",
kernel_initializer="glorot_uniform"))

model.add(Dense(1, activation='sigmoid'))
```

In a second attempt, a modified model was implemented with **one fewer hidden layer** and the **activation function changed from ReLU to** `tanh`, using the `glorot_uniform` initializer for improved stability. This architecture achieved **slightly better results**, with a **training accuracy of 0.9318** and a **training loss of 0.1877**. On the validation set, the model reached a **validation accuracy of 0.8388** and a **validation loss of 0.5420** at its best epoch.

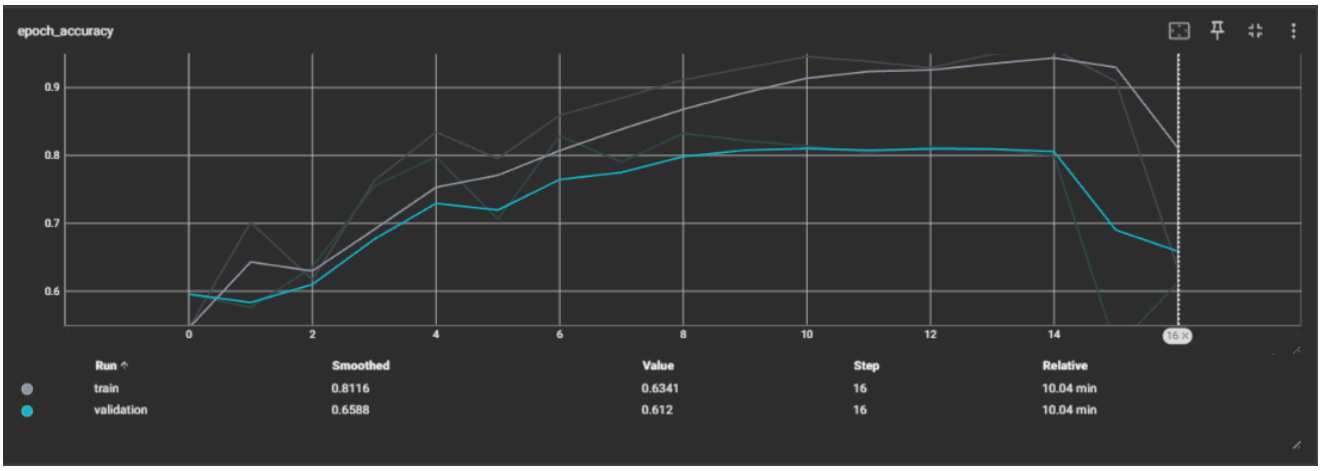


Figure 11: Accuracy per epoch

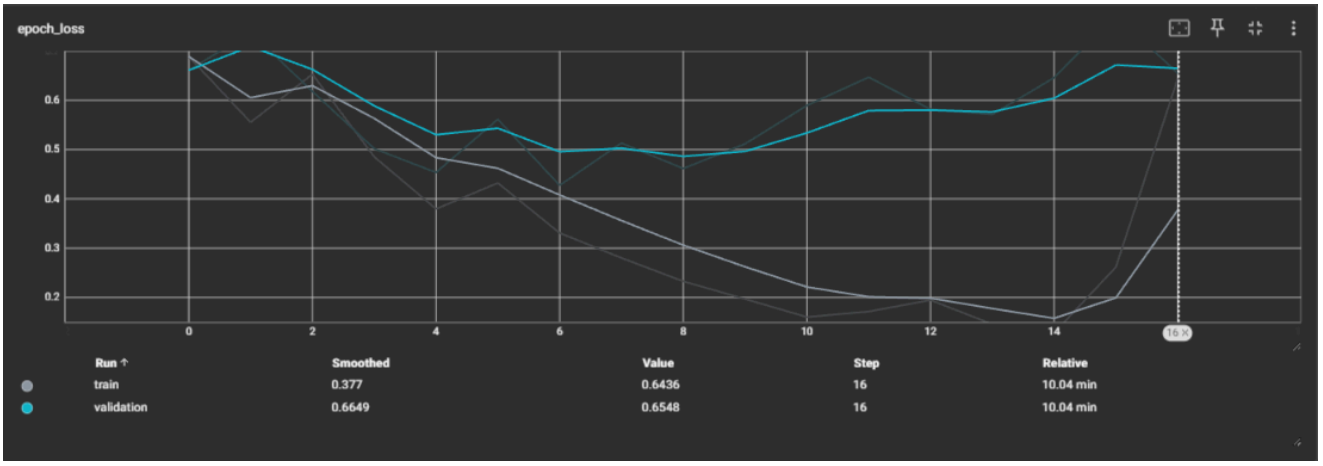


Figure 12: Loss per epoch

Despite these improvements, signs of **gradient instability** persisted, and **early stopping** was triggered after only **12 epochs**. Interestingly, the model generalized even better to unseen data, achieving a **test accuracy of 0.8185** and a **test loss of 0.4046**. These results suggest that reducing model depth and replacing ReLU with `tanh` helped alleviate overfitting and improved overall performance on real-world data.

Review 01:

> *"Good show. THE LAST OF US was indeed a great series. After hearing many raves and good things from people that I know, I finally decided to give it a chance. Like the game, it is definitely worth your time. Well made, compelling and great characters with real chemistry made this series a fun watch, a slower build to some pretty intense scenes. While it did have periods of time that it moved more slowly, it certainly never felt boring. Pedro Pascal as Joe Miller and Bella Ramsey as Ellie Williams made this a great show, and the chemistry they develop feels natural and heartwarming."*

- **Actual IMDB Rating:** 9.0
- **Model Prediction:** `Positive`
- **Confidence Score:** `0.98`

Review 02:

> *"There was so much potential here. Although zombie movies and TV shows have been overplayed in the past 20–30 years, this show has some legs under it. It got me hooked in the first episode and I was really excited for the rest. Unfortunately, even though there were some saving grace episodes or moments here and there, I ended season one feeling let down. Pedro Pascal does a phenomenal job and is the one reason I kept watching. His character slowly develops over time and you really care about him. I know a lot of people loved her, but it was hard for me to watch Bella Ramsey. She acted a couple of parts very well (especially episode 8), but other than when she is yelling and screaming and supposed to be annoying, the times when she was supposed to be likable fell flat and I had a hard time getting through those episodes."*

- **Actual IMDB Rating:** 4.0
- **Model Prediction:** `Negative`
- **Confidence Score:** `0.04`

To evaluate the model's performance on **real-world, unseen data**, two public reviews about *The Last of Us* series (Season 2) were selected from the IMDB website. The first reviewer rated the series **9.0**, while the second gave it a **4.0**, providing a useful contrast for sentiment analysis.

The model successfully identified the sentiment in both cases. The first review was correctly classified as **positive**, with a **prediction score of 0.98**, while the second review — more critical in tone — was classified as **negative**, with a **prediction score of 0.04**. These results suggest that the model is able to generalize well to public, informal reviews and correctly interpret the **emotional tone** conveyed through natural language.

References

- ANN - Churn Classifier project
- **1. Schmidt, R. M. (2019).** *Recurrent Neural Networks (RNNs): A gentle introduction and overview*. arXiv preprint arXiv:1912.05911. Available at: https://arxiv.org/abs/1912.05911
- **2. Sutskever, I., Vinyals, O., & Le, Q. V. (2014).** *Sequence to sequence learning with neural networks*. arXiv preprint arXiv:1409.3215. Available at: https://arxiv.org/abs/1409.3215
- **3. Zen, H., & Sak, H. (2015).** *Unidirectional long short-term memory recurrent neural network with recurrent output layer for low-latency speech synthesis*. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 4470–4474). Available at: https://research.google.com/pubs/archive/43266.pdf
- Bengio, Y., Simard, P., & Frasconi, P. (1994). *Learning long-term dependencies with gradient descent is difficult*. IEEE Transactions on Neural Networks, 5(2), 157–166. https://doi.org/10.1109/72.279181

- Graves, A., Mohamed, A. R., & Hinton, G. (2013). *Speech recognition with deep recurrent neural networks*. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 6645–6649. https://doi.org/10.1109/ICASSP.2013.6638947
- Hochreiter, S., & Schmidhuber, J. (1997). *Long short-term memory*. Neural Computation, 9(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735
- Mikolov, T., Karafiát, M., Burget, L., Cernocký, J., & Khudanpur, S. (2010). *Recurrent neural network based language model*. In *Interspeech*, 1045–1048.
- Sutskever, I., Vinyals, O., & Le, Q. V. (2014). *Sequence to sequence learning with neural networks*. In *Advances in Neural Information Processing Systems (NeurIPS)*, 3104–3112. https://papers.nips.cc/paper_files/paper/2014/file/a14ac55a4f27472c5d894ec1c3c743d2-Paper.pdf
- Elman, J. L. (1990). *Finding structure in time*. Cognitive Science, 14(2), 179–211. https://doi.org/10.1207/s15516709cog1402_1
- Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). *Learning internal representations by error propagation*. In *Parallel distributed processing: Explorations in the microstructure of cognition* (Vol. 1, pp. 318–362).
- Werbos, P. J. (1990). *Backpropagation through time: what it does and how to do it*. Proceedings of the IEEE, 78(10), 1550–1560. https://doi.org/10.1109/5.58337
-
- Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. (2011). Learning Word Vectors for Sentiment Analysis. *The 49th Annual Meeting of the Association for Computational Linguistics (ACL 2011).*