Recurrent Neural Networks and Transformer-encoder

Juliette Mingot

January 2026

Contents

1 Neural Networks and Learning Principles

Global notation. Scalars are denoted by lowercase letters (e.g. x), vectors by bold lowercase letters (e.g. $\mathbf{x}$), and matrices by bold uppercase letters (e.g. $\mathbf{W}$). All vectors are column vectors by default.

Unless stated otherwise:

- $\mathbf{x}_t \in \mathbb{R}^d$ denotes an input feature dimension d vector at time t,
- $\mathbf{h}_t \in \mathbb{R}^h$ denotes a hidden state,
- $\mathbf{W} \in \mathbb{R}^{m\times n}$ denotes a linear mapping from $\mathbb{R}^n$ to $\mathbb{R}^m$,
- $\odot$ denotes the element-wise (Hadamard) product.

Batch dimensions are omitted in theoretical derivations for clarity.

1.1 Mathematical definition of an Artificial Neural Network

Lets defined our artifial neuron as $x \in \mathbb{R}^n$ be an input vector. Given $\phi : \mathbb{R} \to \mathbb{R}$ a nonlinear activation function, an artifical neuron is a function $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}$ defined by $f(\mathbf{x}) = \phi(\mathbf{w}^\top \mathbf{x} + b)$, where $\mathbf{x} \in \mathbb{R}^d$ is the input vector, $\mathbf{w} \in \mathbb{R}^d$ is the weight vector, and $b \in \mathbb{R}$ is a bias term.

A layer (hidden layer) is a vector-valued function defined as $h : \mathbb{R}^{n_{\ell-1}} \to \mathbb{R}^{n_l}$ defined by $\mathbf{h} = \phi(\mathbf{W}\mathbf{x} + \mathbf{b})$,where $\mathbf{W} \in \mathbb{R}^{n_l \times n_{\ell-1}}$,$\mathbf{b} \in \mathbb{R}^{n_l}$ and ϕ act componentwise.

A feed-forward neural network, also known as a multilayer perceptron (Rumelhart et al. (1986)) when multiple hidden layers are used, means that the information flows in a single direction (input-to-output). Hence, A feed-forward neural network with $\boldsymbol{L}$ layers is a composition of $\boldsymbol{\mathcal{F}}_\theta = \boldsymbol{f}_{\boldsymbol{L}} \circ \boldsymbol{f}_{\boldsymbol{L}-1} \circ \cdots \circ \boldsymbol{f}_1$, where each $\boldsymbol{f}_\ell(\mathbf{x}) = \phi_\ell(\mathbf{W}_\ell(\mathbf{x}) + \mathbf{b}_\ell)$, and the parameter set is $\boldsymbol{\theta} = \{\mathbf{W}_\ell, \mathbf{b}_\ell\}_{\ell=1}^{\boldsymbol{L}}$

Our target variable is the gold log-return stock market. Let the prediction target be $y(t) \in \mathbb{R}$ and the network induces is $\hat{y}(t) = \boldsymbol{\mathcal{F}}_\theta(\mathbf{x}(t))$.

Our loss function is defined as folow :

$$\boldsymbol{\mathcal{L}}(\theta) = \sum_{k=1}^{K} \ell(y(t_k), \hat{y}(t_k))$$

We distinguish between continuous calendar time $t \in \mathbb{R}^+$ and discrete observation indices $k \in \{1, \ldots, K\}$. The loss is evaluated only at observed times t_k.

The training correspond to :

$$\theta^\star = \arg\min_\theta \boldsymbol{\mathcal{L}}(\theta)$$

which is the gradient-based optimization

1.2 Backpropagation and Gradient-Based Optimization

A neural network learn their weight and biases using the gradient descent algorithm throught the loss function. The backpropagation methode allow neural network to train network (find the optimal weight and bias). The idea of back propagation tels us that from the partial derivative expression $\partial\boldsymbol{\mathcal{L}}/\partial\mathbf{w}$ of the loss function with respect or anny weight (or bias with $\partial\boldsymbol{\mathcal{L}}/\partial\mathbf{b}$) in the network. The expression tells us how substantially the cost changes when we change the weights and biases, how a specific bias or weight changes the overall behavior of the network.

In order to understand the idea deeply, we will based our explanation from Nielsen (2015) and use some new notation to the existing one this notation will not be used in the next part. Let us use $\mathbf{w}^\ell_{jk}$, the weight for the connection from the k^{th} neuron in the $(\ell-1)^{th}$ layer to the j^{th} neuron in the ℓ^{th} layer. Hence, we can rewrite $\mathbf{x}^\ell_j = \phi\left(\sum_k \mathbf{w}^\ell_{jk} \cdot \mathbf{x}^{\ell-1}_j + \mathbf{b}^\ell_j\right)$, this expression tels us that the activation of the j^{th} neuron in the ℓ^{th} layer is related to the activation in the $(\ell-1)^{th}$ layer, where the sum is over all neurons k in the $(\ell-1)^{th}$ layer.

Backpropagation is based upon four fundamental equations. In order to do that, we introduce δ^ℓ_j which is the error in the corresponding neuron and layer. In order to simplify the expression we will define $\boldsymbol{z}^\ell_j = \sum_k \mathbf{w}^\ell_{jk} \cdot \mathbf{x}^{\ell-1}_j + \mathbf{b}^\ell_j$.

The four fundamental equations of backpropagation are thus:

$$\boldsymbol{\delta}^\ell = \nabla_{\mathbf{x}^\ell}\boldsymbol{\mathcal{L}} \odot \phi'(z^\ell), \tag{1}$$

Equation 1 $\boldsymbol{\delta}^\ell$, measures how sensitive the loss is to changes in the weight input z^ℓ of the final layer. This is the starting point of the backpropagation, it defines the "error signal" for the final layer. $\nabla_{\mathbf{x}^\ell}\boldsymbol{\mathcal{L}}$ measure the loss change with respect to the output activation and $\phi'(z^\ell)$ The derivative of the activation function, evaluated at the weighted input at the last layer ℓ.

$$\boldsymbol{\delta}^\ell = (\mathbf{W}^{\ell+1})^T\delta^{\ell+1} \odot \phi'(z^\ell), \tag{2}$$

Equation 2 $\boldsymbol{\delta}^\ell$, is the heart of backward pass. it recursively compute the error signal $\boldsymbol{\delta}^\ell$ for any hidden layer ℓ, given error signal $\boldsymbol{\delta}^{\ell+1}$ from the layer ahead of its signal. The expression $(\mathbf{W}^{\ell+1})^T\delta^{\ell+1}$, project the error backward through the weigh. Transposing the matrix $\mathbf{W}^{\ell+1}$ performs this backward projection. $\phi'(z^\ell)$, convert error relative to the

output x^ℓ into an error relative to the weighted input. Hence, $\boldsymbol{\delta}^\ell$ ask the question: "**how the loss is to changes in the weighted input $\boldsymbol{z}^\ell$ of the hidden layer**.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^\ell} = \boldsymbol{\delta}^\ell, \tag{3}$$

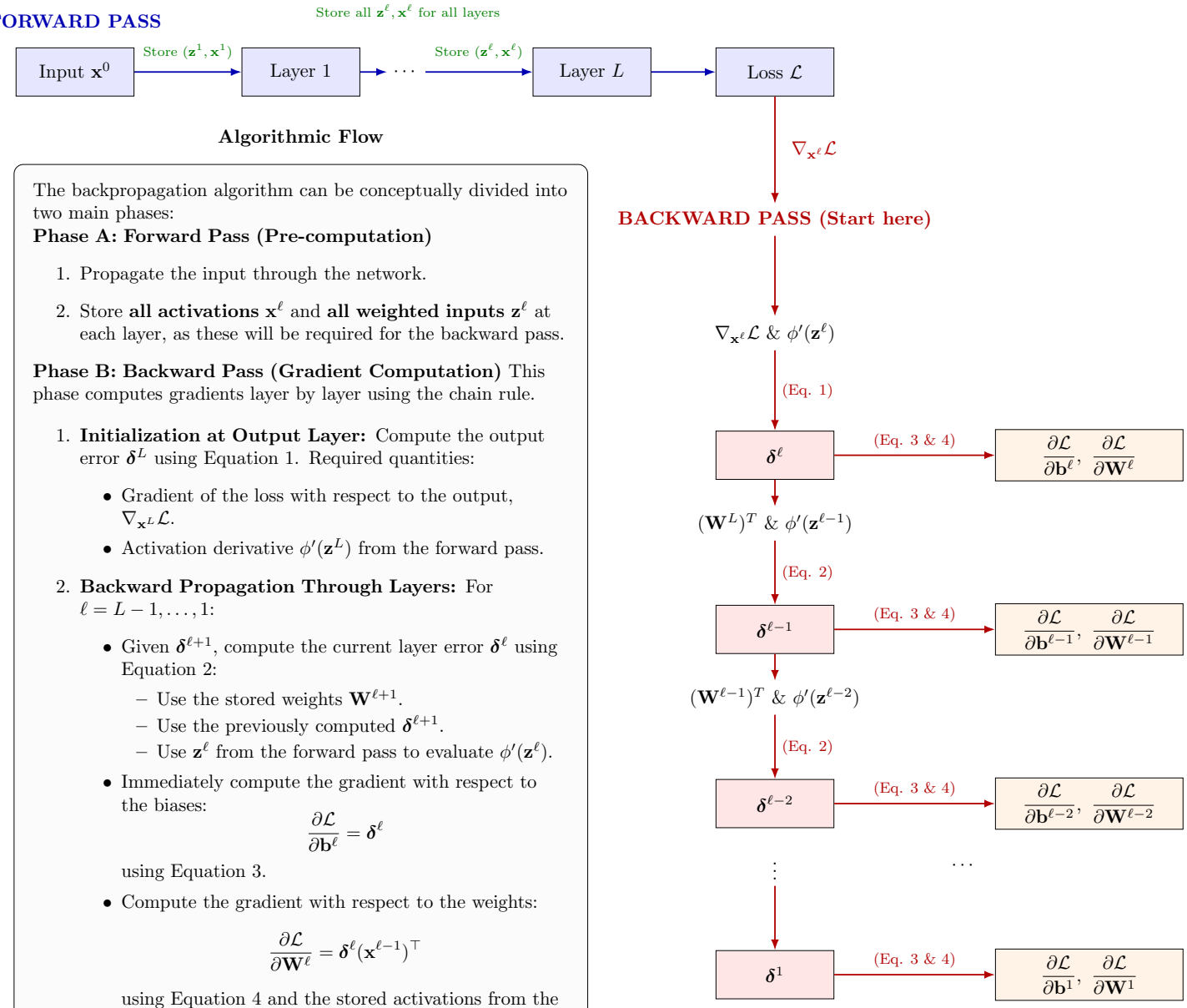
The thirds equation tells us tha the sensitivity of the loss of bias is exactly the loss to the weighted input.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^\ell} = \boldsymbol{\delta}^\ell (\mathbf{x}^{\ell-1})^T. \tag{4}$$

The final goal of backpropagation is to identify the sensitivity of the weight relative to the loss. $\mathbf{x}^{\ell-1})^\top$ is the transpose input that was fed into the layer ℓ during forward-pass. The gradient for a weight is proportional to the error signal of the neuron it connect to $\boldsymbol{\delta}^\ell$, multiplied by the activation of the neuron it connect from $\boldsymbol{x}^\ell$.

Derivations of the four Backpropagation equation are collected in Annex (see 7.1)

1.2.1 The Algorithmic Flow



Algorithmic Flow

The backpropagation algorithm can be conceptually divided into two main phases:
Phase A: Forward Pass (Pre-computation)

1. Propagate the input through the network.
2. Store **all activations** $\mathbf{x}^\ell$ and **all weighted inputs** $\mathbf{z}^\ell$ at each layer, as these will be required for the backward pass.

Phase B: Backward Pass (Gradient Computation) This phase computes gradients layer by layer using the chain rule.

1. **Initialization at Output Layer:** Compute the output error $\boldsymbol{\delta}^L$ using Equation 1. Required quantities:
   - Gradient of the loss with respect to the output, $\nabla_{\mathbf{x}^L}\mathcal{L}$.
   - Activation derivative $\phi'(\mathbf{z}^L)$ from the forward pass.
2. **Backward Propagation Through Layers:** For $\ell = L-1, \ldots, 1$:
   - Given $\boldsymbol{\delta}^{\ell+1}$, compute the current layer error $\boldsymbol{\delta}^\ell$ using Equation 2:
     - Use the stored weights $\mathbf{W}^{\ell+1}$.
     - Use the previously computed $\boldsymbol{\delta}^{\ell+1}$.
     - Use $\mathbf{z}^\ell$ from the forward pass to evaluate $\phi'(\mathbf{z}^\ell)$.
   - Immediately compute the gradient with respect to the biases:
     $$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^\ell} = \boldsymbol{\delta}^\ell$$
     using Equation 3.
   - Compute the gradient with respect to the weights:
     $$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^\ell} = \boldsymbol{\delta}^\ell (\mathbf{x}^{\ell-1})^\top$$
     using Equation 4 and the stored activations from the forward pass.
3. **Iterate:** The computed $\boldsymbol{\delta}^\ell$ becomes $\boldsymbol{\delta}^{\ell+1}$ for the next iteration, moving backward through the network.

2 Sequential Modeling with Recurrent Neural Networks

Discrete-time assumption. In this section, we consider sequences sampled at regular discrete time steps indexed by $t = 1, \ldots, T$. Each observation is represented by an input vector $\mathbf{x}_t \in \mathbb{R}^d$. Irregularly sampled time series are addressed explicitly in Section 4.

2.1 Recurrent Neural Networks

Unlike feedforward neural networks, which assume independent inputs. Recurrent Neural Networks (RNNs) where firstly introduced in 1990 by Werbos (1990), there are designed to model sequential data by introducing a latent state that evolves over time. At each time step, the network updates an internal hidden state that summarizes past information and influences future predictions.

2.1.1 Hidden States and Temporal Dependency

An RNN maintains a hidden state $\mathbf{h}_t \in \mathbb{R}^h$, which encodes information from the current input $\mathbf{x}_t$ as well as from all previous inputs through recursion. According to Elman (1990), the hidden state update is defined as

$$\mathbf{h}_t = \phi_h(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h)\,, \tag{5}$$

where

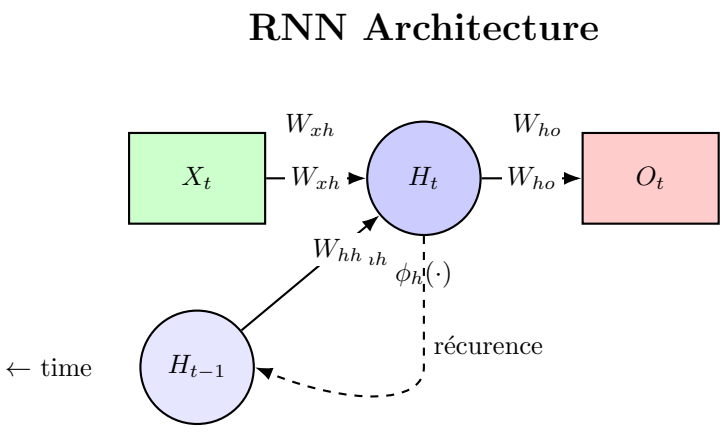
$$\mathbf{W}_{xh} \in \mathbb{R}^{h\times d}, \quad \mathbf{W}_{hh} \in \mathbb{R}^{h\times h}, \quad \mathbf{b}_h \in \mathbb{R}^h,$$

and ϕ_h is a nonlinear activation function applied componentwise (e.g. tanh).

Because $\mathbf{h}_{t-1}$ already contains information from earlier time steps, the hidden state $\mathbf{h}_t$ implicitly depends on the entire input history:

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{x}_{t-1}, \ldots, \mathbf{x}_1).$$

This recursive structure enables RNNs to capture temporal dependencies and contextual effects in sequential data.



2.1.2 Backpropagation Through Time

Training an RNN requires computing gradients of a loss function with respect to parameters that are reused at every time step.

Unlike feedforward networks, where each parameter appears once in the computational graph, RNN parameters influence the loss through multiple temporal paths. To compute gradients correctly, the network is unrolled across time, and the chain rule is applied along both depth and temporal dimensions. This procedure is known as *Backpropagation Through Time* (Werbos (1990)).

For example, the gradient of the loss with respect to the recurrent weight matrix $\mathbf{W}_{hh}$ takes the form

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} \sum_{k=1}^{t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial \mathbf{W}_{hh}}. \tag{6}$$

This expression highlights that a single parameter update must account for how perturbations at early time steps propagate forward and affect future predictions. Schmidt (1912) give a well detailed explanation about this algorithme.

2.1.3 Vanishing and Exploding Gradients

During BPTT, gradients are propagated backward through repeated applications of the hidden-state transition Jacobian. The sensitivity of the hidden state at time t with respect to an earlier hidden state at time $k < t$ is given by

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} = \prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}. \tag{7}$$

From the update equation (5), the Jacobian at time j can be written as

$$\frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}} = \mathbf{D}_j \mathbf{W}_{hh}, \qquad \mathbf{D}_j = \operatorname{diag}(\phi_h'(\cdot)). \tag{8}$$

Let $\|\cdot\|$ denote a submultiplicative matrix norm. Assuming $\|\mathbf{D}_j\| \leq \gamma$ for all j, where $\gamma > 0$ depends on the activation function, we obtain the bound

$$\left\| \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \right\| \leq (\gamma \|\mathbf{W}_{hh}\|)^{t-k}. \tag{9}$$

If $\gamma \rho(\mathbf{W}_{hh}) < 1$, where $\rho(\cdot)$ denotes the spectral radius, the gradient norm decays exponentially as $t - k$ increases, leading to the *vanishing gradient problem*. Conversely, if $\gamma \rho(\mathbf{W}_{hh}) > 1$, the gradient grows exponentially, resulting in *exploding gradients*. These phenomena severely limit the ability of standard RNNs to learn long-range temporal dependencies.

This analysis motivates the introduction of gated recurrent architectures, which explicitly regulate information and gradient flow through time.

2.2 Long Short-Term Memory Networks

Standard RNNs suffer from vanishing and exploding gradients due to repeated multiplicative interactions in the hidden-state dynamics. Developed by Hochreiter and Schmidhuber (1997), the Long Short-Term Memory (LSTM) networks address this limitation by introducing an explicit memory variable whose evolution allows for more stable gradient flow over long time horizons.

2.2.1 Memory Cells and Gating Mechanisms

An LSTM (also well detailed by Schmidt (1912)), maintains two state variables at each time step: a hidden state $\mathbf{h}_t \in \mathbb{R}^h$ and a memory cell $\mathbf{c}_t \in \mathbb{R}^h$. The memory cell is designed to store information additively over time, while gates control how information is written, erased, and read.

Given an input $\mathbf{x}_t \in \mathbb{R}^d$, the LSTM update equations are defined as

$$\mathbf{i}_t = \sigma(\mathbf{W}_{xi}\mathbf{x}_t + \mathbf{W}_{hi}\mathbf{h}_{t-1} + \mathbf{b}_i) \quad \text{(input gate)}, \tag{10}$$

$$\mathbf{f}_t = \sigma(\mathbf{W}_{xf}\mathbf{x}_t + \mathbf{W}_{hf}\mathbf{h}_{t-1} + \mathbf{b}_f) \quad \text{(forget gate)}, \tag{11}$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_{xo}\mathbf{x}_t + \mathbf{W}_{ho}\mathbf{h}_{t-1} + \mathbf{b}_o) \quad \text{(output gate)}, \tag{12}$$

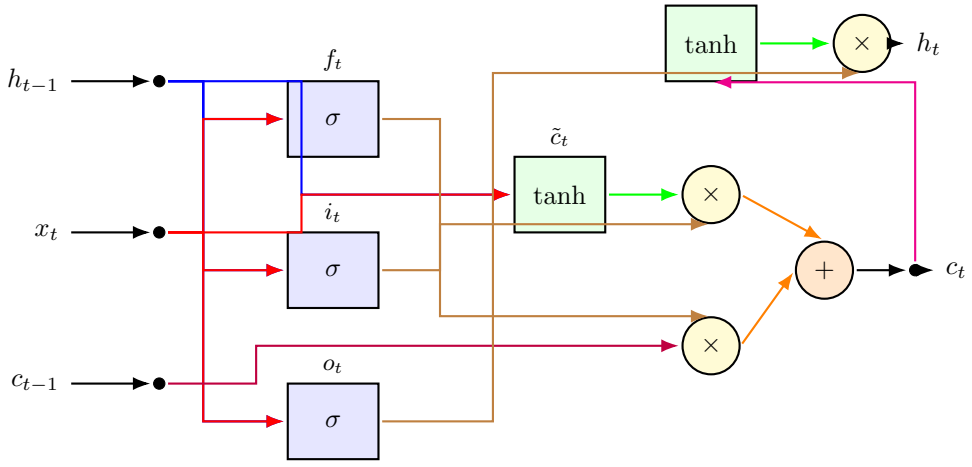
$$\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_{xc}\mathbf{x}_t + \mathbf{W}_{hc}\mathbf{h}_{t-1} + \mathbf{b}_c) \quad \text{(candidate cell state)}, \tag{13}$$

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t \quad \text{(cell update)}, \tag{14}$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \quad \text{(hidden state)}. \tag{15}$$

All weight matrices $\mathbf{W}_{x*} \in \mathbb{R}^{h\times d}$, $\mathbf{W}_{h*} \in \mathbb{R}^{h\times h}$, and biases $\mathbf{b}_* \in \mathbb{R}^h$ are learnable parameters. The sigmoid function $\sigma(\cdot)$ ensures that gate activations lie in $(0, 1)$.

Cellule Long Short-Term Memory (LSTM)



2.2.2 Stability of Gradient Propagation

The key mechanism that stabilizes gradient propagation in LSTMs is the additive update of the memory cell. The gradient of the loss $\mathcal{L}$ with respect to the cell state satisfies the

recursive relation

$$\frac{\partial \mathcal{L}}{\partial \mathbf{c}_t} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}\frac{\partial \mathbf{h}_t}{\partial \mathbf{c}_t} + \frac{\partial \mathcal{L}}{\partial \mathbf{c}_{t+1}}\frac{\partial \mathbf{c}_{t+1}}{\partial \mathbf{c}_t} \tag{16}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{c}_t} = \underbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} \odot \mathbf{o}_t \odot \left(1 - \tanh^2(\mathbf{c}_t)\right)}_{\text{contribution from the hidden state}} + \underbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{c}_{t+1}} \odot \mathbf{f}_{t+1}}_{\text{contribution from the next cell state}} . \tag{17}$$

The forget gate $\mathbf{f}_{t+1}$ directly modulates the contribution of future gradients to the current time step. When $\mathbf{f}_{t+1} \approx \mathbf{1}$, gradients propagate nearly unchanged across time, allowing long-term dependencies to be learned. This controlled gradient flow explains why LSTMs are significantly more stable than standard RNNs when trained with Back-propagation Through Time.

Gates as Valves in Gradient Flow

- **Forget gate $\mathbf{f}_t$.** Controls how much of the gradient from $\mathbf{c}_{t+1}$ flows back to $\mathbf{c}_t$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{c}_t} \supset \frac{\partial \mathcal{L}}{\partial \mathbf{c}_{t+1}} \odot \mathbf{f}_{t+1}. \tag{18}$$

  When $\mathbf{f}_{t+1} \approx \mathbf{1}$, gradients are preserved over long time spans; when $\mathbf{f}_{t+1} \approx \mathbf{0}$, gradient flow is effectively blocked.

- **Input gate $\mathbf{i}_t$.** Controls how strongly the candidate state influences the loss:

$$\frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{c}}_t} = \frac{\partial \mathcal{L}}{\partial \mathbf{c}_t} \odot \mathbf{i}_t. \tag{19}$$

- **Output gate $\mathbf{o}_t$.** Determines how much of the memory cell affects the hidden-state gradient:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{o}_t} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} \odot \tanh(\mathbf{c}_t). \tag{20}$$

2.2.3 Limitations of LSTM

Despite their effectiveness, LSTMs have several practical limitations. The presence of multiple gates increases the computational cost and parameter count relative to simpler recurrent architectures. Moreover, while LSTMs mitigate gradient instability, they do not eliminate it entirely, and learning extremely long-range dependencies may still require careful initialization and regularization.

2.3 Gated Recurrent Units

Gated Recurrent Units (GRUs) was invented by Cho et al. (2014). It provide a simplified alternative to LSTMs that retains the core idea of gated information flow while reducing architectural complexity. Unlike LSTMs, GRUs do not maintain a separate memory cell; instead, memory is integrated directly into the hidden state.

2.3.1 Update and Reset Gates

A GRU maintains a single hidden state $\mathbf{h}_t \in \mathbb{R}^h$. Given input $\mathbf{x}_t \in \mathbb{R}^d$, the GRU update equations are

$$\mathbf{z}_t = \sigma(\mathbf{W}_{xz}\mathbf{x}_t + \mathbf{W}_{hz}\mathbf{h}_{t-1} + \mathbf{b}_z) \quad \text{(update gate)}, \tag{21}$$

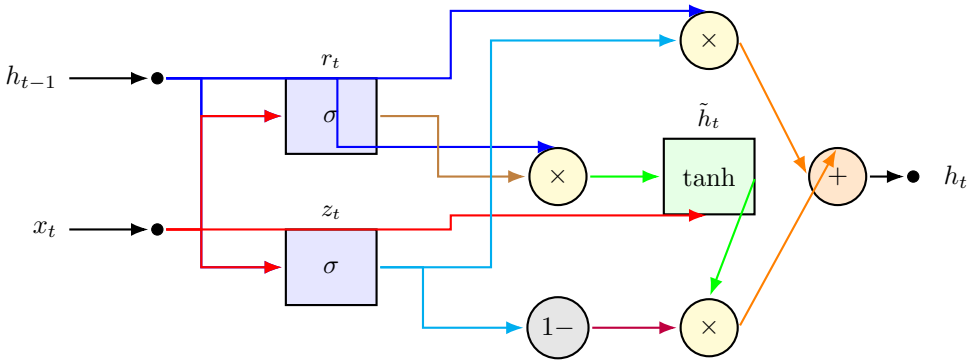
$$\mathbf{r}_t = \sigma(\mathbf{W}_{xr}\mathbf{x}_t + \mathbf{W}_{hr}\mathbf{h}_{t-1} + \mathbf{b}_r) \quad \text{(reset gate)}, \tag{22}$$

$$\tilde{\mathbf{h}}_t = \tanh(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}(\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{b}_h) \quad \text{(candidate state)}, \tag{23}$$

$$\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t \quad \text{(state update)}. \tag{24}$$

As in the LSTM case, all parameters are learnable, with weight matrices of dimensions $\mathbf{W}_{x*} \in \mathbb{R}^{h\times d}$ and $\mathbf{W}_{h*} \in \mathbb{R}^{h\times h}$.

Cellule GRU (Gated Recurrent Unit)



2.3.2 Relation to Gradient Stability

The update gate $\mathbf{z}_t$ in a GRU plays a role analogous to the forget gate in an LSTM. When $\mathbf{z}_t$ is small, the hidden state is largely preserved, allowing gradients to flow across time steps with limited attenuation. Conversely, when $\mathbf{z}_t$ is large, the hidden state is rapidly updated to incorporate new information.

This adaptive interpolation between memory retention and state renewal enables GRUs to mitigate vanishing gradients while maintaining a simpler structure than LSTMs.

3 Continuous-Time Perspectives on Neural Networks

Depth-time distinction. In this chapter, the variable t denotes *network depth* rather than calendar time. This distinction is made explicit when transitioning to continuous-depth models.

3.1 Residual Networks

As neural networks grow deeper, training becomes increasingly difficult due to the degradation of gradient signals across layers. Even when vanishing gradients are mitigated,

deeper architectures may exhibit higher training error than their shallower counterparts, indicating an optimization difficulty rather than a representational limitation.

3.1.1 Skip Connections and Residual Learning

Introduced by He et al. (2016), Residual Learning addresses this issue by introducing *skip connections* that allow information to bypass one or more nonlinear transformations. Instead of learning a direct mapping from inputs to outputs, residual networks reformulate the learning problem as the estimation of a residual function.

Let $\mathbf{h}_\ell \in \mathbb{R}^h$ denote the hidden representation at layer (or depth) ℓ. A standard feedforward layer computes

$$\mathbf{h}_\ell = \mathcal{F}_\ell(\mathbf{h}_{\ell-1}), \tag{25}$$

where $\mathcal{F}_\ell : \mathbb{R}^h \to \mathbb{R}^h$ is a nonlinear transformation composed of linear mappings and activation functions.

In a residual network, the layer output is defined instead as

$$\mathbf{h}_\ell = \mathbf{h}_{\ell-1} + \mathcal{F}_\ell(\mathbf{h}_{\ell-1}), \tag{26}$$

where the identity mapping $\mathbf{h}_{\ell-1} \mapsto \mathbf{h}_{\ell-1}$ is implemented via a skip connection.

This formulation allows the network to learn the residual function

$$\mathcal{R}_\ell(\mathbf{h}_{\ell-1}) = \mathbf{h}_\ell - \mathbf{h}_{\ell-1},$$

which empirically has been shown to be easier to optimize than the original mapping.

Interpretation. Skip connections can be interpreted as enabling iterative refinement of representations across depth. If the residual function $\mathcal{F}_\ell$ is small, the network effectively preserves the input representation, allowing deeper models to behave similarly to shallower ones when needed. This property makes residual architectures robust to increasing depth and forms the conceptual foundation for continuous-depth models such as Neural Ordinary Differential Equations.

3.1.2 Effect on Gradient Propagation.

During backpropagation, the gradient of the loss $\mathcal{L}$ with respect to the representation at layer $\ell - 1$ is given by

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_{\ell-1}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_\ell} \left(\mathbf{I} + \frac{\partial \mathcal{F}_\ell(\mathbf{h}_{\ell-1})}{\partial \mathbf{h}_{\ell-1}} \right), \tag{27}$$

where $\mathbf{I}$ denotes the identity matrix.

The presence of the identity term ensures that, even if the Jacobian of $\mathcal{F}_\ell$ has small singular values, a direct gradient path remains available. As a result, gradients can propagate across many layers without being excessively attenuated, greatly improving the trainability of deep networks. *Derivation of the gradient flow are collected in Annex (see 7.2)*

3.2 Neural Ordinary Differential Equations

Residual networks can be interpreted as discrete-time dynamical systems evolving over network depth. Neural Ordinary Differential Equations (Chen et al. (2018)) extend ResNet's viewpoint by modeling depth as a continuous variable and defining the evolution of the hidden state through an ordinary differential equation.

3.2.1 Continuous-Depth Limit of Residual Networks

Let $t \in [0, T]$ denote a continuous depth variable. We interpret the discrete residual update (26) as a finite-difference approximation of a continuous-time dynamical system. Specifically, writing

$$\frac{\mathbf{h}_\ell - \mathbf{h}_{\ell-1}}{\Delta t} = \mathcal{R}_\ell(\mathbf{h}_{\ell-1}), \qquad \Delta t > 0, \tag{28}$$

and letting $\Delta t \to 0$, we obtain the continuous-depth limit

$$\frac{d\mathbf{h}(t)}{dt} = f_\theta(\mathbf{h}(t), t), \tag{29}$$

where $\mathbf{h}(t) \in \mathbb{R}^h$ denotes the hidden state at depth t, $f_\theta : \mathbb{R}^h \times \mathbb{R} \to \mathbb{R}^h$ is a neural network parameterized by θ, and the parameters are shared across all depths.

Given an initial condition $\mathbf{h}(0)$, the output of the model is defined as the solution of the initial value problem (29) evaluated at depth T:

$$\mathbf{h}(T) = \mathbf{h}(0) + \int_0^T f_\theta(\mathbf{h}(t), t)\, dt = \mathrm{ODESolve}(f_\theta, \mathbf{h}(0), 0, T)\,. \tag{30}$$

3.2.2 ODE-Recurrent Neural Networks

While Neural ODEs model the hidden state as a fully continuous-time trajectory, ODE-Recurrent Neural Networks (ODE-RNNs) developed by Rubanova et al. (2019) are specifically designed for irregularly-sampled time series, where observations may arrive at uneven intervals. The key idea is to combine continuous-time latent evolution with discrete updates driven by observed data.

Continuous-Time Hidden State Evolution

Let $\{t_i\}_{i=0}^N$ denote the observation times and $x_i \in \mathbb{R}^d$ the corresponding inputs. Between two consecutive observation times t_{i-1} and t_i, the hidden state $h(t) \in \mathbb{R}^h$ evolves according to a parameterized ordinary differential equation

$$\frac{dh(t)}{dt} = f_\theta(h(t)), \quad t \in (t_{i-1}, t_i), \tag{31}$$

where f_θ is typically a neural network that defines the latent dynamics.

The hidden state immediately before the next observation is obtained by solving this ODE:

$$h_i' = \mathrm{ODESolve}\big(f_\theta, h_{i-1}, t_{i-1}, t_i\big). \tag{32}$$

Discrete Observation Updates

At the observation time t_i, the hidden state is updated using a standard recurrent cell (e.g., GRU or LSTM) to incorporate the new input x_i:

$$h_i = \text{RNNCell}(h'_i, x_i). \tag{33}$$

This discrete update allows the model to integrate the observed information while maintaining the continuous-time latent trajectory between observations. The combination of continuous evolution and discrete updates enables ODE-RNNs to naturally handle irregularly-sampled data.

4 Transformers and Bidirectional Transformer Encoder

This section introduces the Transformer architecture from first principles. No prior knowledge of attention mechanisms is assumed. The objective is to provide the conceptual foundations required to understand BERT and its role in the modeling framework used in this work.

4.1 Transformer architecture and self-attention mechanism

Classical sequence models such as Recurrent Neural Networks (RNNs) and Long Short-Term Memory networks (LSTMs) process sequences sequentially, updating a hidden state one time step after another. While effective, this sequential nature limits parallelization and makes it difficult to capture long-range dependencies in long sequences.

The Transformer architecture was introduced by Vaswani et al. (2017) to address these limitations. Instead of processing tokens sequentially, a Transformer processes the entire sequence simultaneously by allowing each element of the sequence to directly interact with all others. This interaction is achieved through a mechanism called *attention*.

A Transformer is composed of two main blocks:

- an **encoder**, which maps an input sequence to a sequence of contextual representations,
- a **decoder**, which generates an output sequence in an autoregressive manner.

Different models correspond to different uses of these blocks. BERT relies exclusively on the encoder, while models such as GPT rely exclusively on the decoder. In this work, the focus is placed on encoder-based architectures, as BERT is used for textual representation.

4.1.1 The self-attention mechanism of the encoder

Intuition The key idea behind self-attention is that the representation of a word should depend on other words in the same sentence. Rather than assigning fixed importance based on position, self-attention allows the model to learn which words are relevant to each other based on their content.

For each token in the sequence, the model computes a weighted combination of all other tokens. Tokens that are more relevant receive higher weights, while less relevant tokens contribute less to the final representation.

Token representation The Transformer encoder operates on data represented as a sequence of tokens, where each token is associated with a fixed-dimensional embedding. Let

$$\mathbf{x}_j^{(0)} \in \mathbb{R}^D, \quad j = 1, \ldots, N,$$

denote the embedding of the j-th token. We collect all token embeddings into the matrix

$$\mathbf{X}^{(0)} = \begin{pmatrix} \mathbf{x}_1^{(0)} & \mathbf{x}_2^{(0)} & \cdots & \mathbf{x}_N^{(0)} \end{pmatrix} \in \mathbb{R}^{D \times N}.$$

Each column of $\mathbf{X}^{(0)}$ represents a token embedding that initially contains only local information. The purpose of the Transformer encoder is to transform these representations into context-aware embeddings by allowing interactions between all tokens.

After passing through L encoder layers, the output is

$$\mathbf{X}^{(L)} \in \mathbb{R}^{D \times N},$$

where $\mathbf{x}_j^{(L)} = \mathbf{X}_{:,j}^{(L)}$ is the final representation of token j.

Encoder block overview Each encoder layer is composed of two main components:

1. A multi-head self-attention mechanism that aggregates information across tokens.
2. A position-wise feed-forward neural network that processes features independently for each token.

Notation Let:

- N be the number of tokens,
- D the embedding (model) dimension,
- H the number of attention heads,
- $d_k = D/H$ the dimensionality of each attention head.

4.1.2 Stage 1: Self-attention (token interaction)

Given the input $\mathbf{X}^{(l-1)} \in \mathbb{R}^{D \times N}$ to encoder layer ℓ, the self-attention mechanism produces an output $\mathbf{Y}^{(l)} \in \mathbb{R}^{D \times N}$.

For a token at position j, the output vector is computed as a weighted average of all input token representations:

$$\mathbf{y}_j^{(l)} = \sum_{i=1}^{N} \mathbf{x}_i^{(l-1)} A_{i,j}^{(l)},$$

where $A_{i,j}^{(l)}$ measures the relevance of token i to token j.

In matrix form, this operation is written as

$$\mathbf{Y}^{(l)} = \mathbf{X}^{(l-1)} \mathbf{A}^{(l)},$$

where $\mathbf{A}^{(l)} \in \mathbb{R}^{N \times N}$ is the attention matrix satisfying (*column-wise normalization*)

$$\sum_{i=1}^{N} A_{i,j}^{(l)} = 1 \quad \forall j.$$

Queries, keys, and values The attention matrix is computed dynamically from the input representations using learned linear projections. For each attention head $h = 1, \ldots, H$, we define parameter matrices

$$\mathbf{W}_{Q,h}^{(l)}, \mathbf{W}_{K,h}^{(l)}, \mathbf{W}_{V,h}^{(l)} \in \mathbb{R}^{d_k \times D}.$$

The corresponding queries, keys, and values are given by

$$\mathbf{Q}_h^{(l)} = \mathbf{W}_{Q,h}^{(l)} \mathbf{X}^{(l-1)}, \quad \mathbf{K}_h^{(l)} = \mathbf{W}_{K,h}^{(l)} \mathbf{X}^{(l-1)}, \quad \mathbf{V}_h^{(l)} = \mathbf{W}_{V,h}^{(l)} \mathbf{X}^{(l-1)},$$

with

$$\mathbf{Q}_h^{(l)}, \mathbf{K}_h^{(l)}, \mathbf{V}_h^{(l)} \in \mathbb{R}^{d_k \times N}.$$

Intuitively, queries encode what information a token seeks, while keys encode what information a token provides.

Attention weights The attention scores for head h are computed as

$$\mathbf{S}_h^{(l)} = \frac{(\mathbf{Q}_h^{(l)})^\top \mathbf{K}_h^{(l)}}{\sqrt{d_k}} \in \mathbb{R}^{N \times N}.$$

Applying a column-wise softmax yields the attention matrix

$$\mathbf{A}_h^{(l)} = \mathrm{softmax}(\mathbf{S}_h^{(l)}).$$

Multi-head attention Each head produces an output

$$\mathbf{Z}_h^{(l)} = \mathbf{V}_h^{(l)} \mathbf{A}_h^{(l)} \in \mathbb{R}^{d_k \times N}.$$

Multiple attention heads are computed in parallel, allowing the model to capture different types of relationships simultaneously in distinct subspaces.

The outputs of all heads are concatenated:

$$\mathbf{Z}^{(l)} = \mathrm{Concat}\left(\mathbf{Z}_1^{(l)}, \ldots, \mathbf{Z}_H^{(l)}\right) \in \mathbb{R}^{D \times N}.$$

A final linear projection is applied using

$$\mathbf{W}_O^{(l)} \in \mathbb{R}^{D \times D},$$

yielding

$$\mathrm{MHSA}(\mathbf{X}^{(l-1)}) = \mathbf{W}_O^{(l)} \mathbf{Z}^{(l)}.$$

Residual connection and layer normalization The attention output is combined with the input via a residual connection and normalized:

$$\tilde{\mathbf{X}}^{(l)} = \mathrm{LayerNorm}\left(\underbrace{\mathbf{X}^{(l-1)} + \mathrm{MHSA}(\mathbf{X}^{(l-1)})}_{\text{residual}}\right) \in \mathbb{R}^{D \times N}.$$

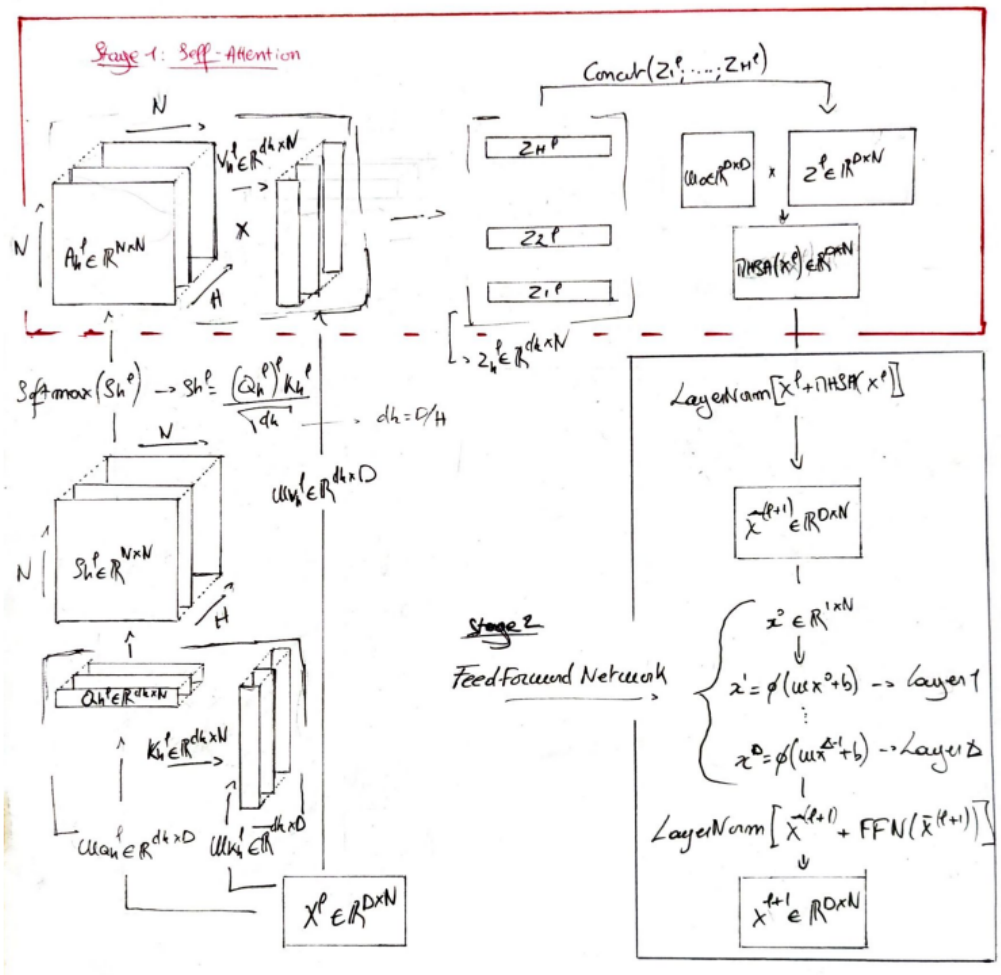
4.1.3 Stage 2: Position-wise feed-forward network

The normalized representation $\tilde{\mathbf{X}}^{(l)}$ is then passed through a position-wise feed-forward neural network, as defined in the previous section. The same network is applied independently to each token.

Residual connection and normalization (post-FFN) The output of the encoder layer is finally obtained as

$$\mathbf{X}^{(l)} = \text{LayerNorm}\left(\underbrace{\tilde{\mathbf{X}}^{(l)} + \text{FFN}(\tilde{\mathbf{X}}^{(l)})}_{\text{residual}}\right) \in \mathbb{R}^{D \times N}.$$

Encoder mechanism



For a more detailed explanation about the encoder and self-attention : Turner (2023)

4.2 Bidirectionality in BERT

Unlike traditional language models that process text from left to right, BERT (Devlin et al. (2019)) was designed to use both left and right context simultaneously. This means that the representation of a token depends on the entire sentence, not only on preceding tokens (unlike original transformer-encoder that we saw previously).

This bidirectional behavior is made possible by self-attention, which allows each token to attend to all other tokens in the sequence. As a result, BERT produces deeply contextualized representations that are particularly well suited for language understanding tasks.

4.2.1 Text Embedding technique with BERT

Given a tokenized input sequence of length N, BERT constructs the input to the Transformer encoder by combining three embedding components: token, segment, and position embeddings. The resulting embedding matrix is

$$\mathbf{E}^{(0)} \in \mathbb{R}^{D\times N},$$

with $D = 768$ for BERT-base.

Token embeddings Let $t_j \in \{1, \ldots, V\}$ denote the WordPiece vocabulary index of token j, where $V = 30{,}522$. Token embeddings are obtained from the learned matrix

$$\mathbf{W}_{\text{tok}} \in \mathbb{R}^{D\times V},$$

such that

$$\mathbf{e}_j^{\text{tok}} = \mathbf{W}_{\text{tok}}\mathbf{e}_{t_j}, \qquad \mathbf{E}^{\text{tok}} \in \mathbb{R}^{D\times N}.$$

These embeddings encode lexical and semantic information but are independent of token order and sentence structure.

Segment embeddings To distinguish between tokens belonging to different input segments (sentence A or B), each token is assigned a segment index $s_j \in \{0, 1\}$. Segment embeddings are drawn from

$$\mathbf{W}_{\text{seg}} \in \mathbb{R}^{D\times 2}, \qquad \mathbf{E}^{\text{seg}} \in \mathbb{R}^{D\times N},$$

with

$$\mathbf{e}_j^{\text{seg}} = \mathbf{W}_{\text{seg}}\mathbf{e}_{s_j}.$$

Position embeddings Since self-attention is permutation-invariant, positional information is injected via learned absolute position embeddings. Let $p_j = j - 1$ denote the position of token j, with a maximum sequence length $N_{\max} = 512$. Position embeddings are obtained from

$$\mathbf{W}_{\text{pos}} \in \mathbb{R}^{D\times N_{\max}}, \qquad \mathbf{E}^{\text{pos}} \in \mathbb{R}^{D\times N},$$

where

$$\mathbf{e}_j^{\text{pos}} = \mathbf{W}_{\text{pos}}\mathbf{e}_{p_j}.$$

Embedding composition The final embedding for token j is the sum of the three components:

$$\mathbf{e}_j^{(0)} = \mathbf{e}_j^{\text{tok}} + \mathbf{e}_j^{\text{seg}} + \mathbf{e}_j^{\text{pos}}.$$

In matrix form:

$$\mathbf{E}^{(0)} = \mathbf{E}^{\text{tok}} + \mathbf{E}^{\text{seg}} + \mathbf{E}^{\text{pos}} \in \mathbb{R}^{D \times N}.$$

This combined representation is normalized before entering the encoder:

$$\mathbf{X}^{(0)} = \text{LayerNorm}(\mathbf{E}^{(0)}).$$

Remarks All embedding matrices $\mathbf{W}_{\text{tok}}$, $\mathbf{W}_{\text{seg}}$, and $\mathbf{W}_{\text{pos}}$ are learnable parameters trained jointly with the model. Compared to standard word embeddings, BERT's embedding scheme explicitly encodes sentence membership and token position, producing input representations that are well-suited for deep bidirectional contextualization via self-attention.

Contextual token representations After passing through multiple Transformer encoder layers, the model outputs a matrix:

$$\mathbf{H} \in \mathbb{R}^{n \times 768},$$

where each row corresponds to a context-dependent representation of a token. These representations incorporate information from the entire sequence and reflect a specific semantics (*for our project it will be finance semantics*) learned during pre-training.

4.2.2 Pre-training and fine-tuning

The training of BERT is divided into two distinct phases.

Pre-training During pre-training, BERT is trained on large unlabeled text corpora using self-supervised objectives. The primary objective is *Masked Language Modeling*, where a subset of tokens is masked and the model is trained to predict them using the surrounding context. This phase allows the model to learn general linguistic and semantic structures.

Fine-tuning In the fine-tuning phase, the pre-trained model is adapted to a specific downstream task. This is achieved by adding a small task-specific layer on top of BERT and updating the parameters using labeled data.

4.2.3 Financial domain adaptation: FinBERT

FinBERT [Yang et al. (2020)] follows the same architecture as BERT but is further pre-trained on financial text. This additional pre-training allows the model to better capture the semantics of financial language, such as monetary policy statements, inflation dynamics, and market-related terminology.

In this work, FinBERT is used as a text encoder to extract financial-context-aware representations from policy documents.

5 Annex

5.1 Backpropagation Equations

5.1.1 Error in the output layer

Define the error at the output layer $\ell = L$ as

$$\delta^\ell = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^\ell}, \tag{34}$$

where $\mathbf{z}^\ell$ denotes the pre-activation vector and $\mathbf{x}^\ell = \phi(\mathbf{z}^\ell)$ the post-activation output of layer ℓ.

By the chain rule, component-wise:

$$\delta_j^\ell = \frac{\partial \mathcal{L}}{\partial x_j^\ell} \frac{\partial x_j^\ell}{\partial z_j^\ell} = \frac{\partial \mathcal{L}}{\partial x_j^\ell} \, \phi'(z_j^\ell), \tag{35}$$

and in vector form:

$$\delta^\ell = \nabla_{\mathbf{x}^\ell} \mathcal{L} \; \odot \; \phi'(\mathbf{z}^\ell). \tag{36}$$

For a quadratic loss $\mathcal{L} = \frac{1}{2}\|\mathbf{x}^\ell - \mathbf{y}\|^2$:

$$\delta^\ell = (\mathbf{x}^\ell - \mathbf{y}) \; \odot \; \phi'(\mathbf{z}^\ell). \tag{37}$$

5.1.2 Error in hidden layers

For a hidden layer $\ell < L$:

$$\delta^\ell = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^\ell}. \tag{38}$$

Using the chain rule over the next layer $\ell + 1$:

$$\delta^\ell = (\mathbf{W}^{\ell+1})^\top \delta^{\ell+1} \; \odot \; \phi'(\mathbf{z}^\ell), \tag{39}$$

since

$$\mathbf{z}^{\ell+1} = \mathbf{W}^{\ell+1}\mathbf{x}^\ell + \mathbf{b}^{\ell+1}, \quad \mathbf{x}^\ell = \phi(\mathbf{z}^\ell). \tag{40}$$

5.1.3 Gradient with respect to the bias

By definition,

$$z_j^\ell = \sum_k w_{jk}^\ell x_k^{\ell-1} + b_j^\ell. \tag{41}$$

Then

$$\frac{\partial \mathcal{L}}{\partial b_j^\ell} = \frac{\partial \mathcal{L}}{\partial z_j^\ell} \frac{\partial z_j^\ell}{\partial b_j^\ell} = \delta_j^\ell. \tag{42}$$

5.1.4 Gradient with respect to the weights

Similarly, for a weight w_{jk}^ℓ:

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^\ell} = \frac{\partial \mathcal{L}}{\partial z_j^\ell} \frac{\partial z_j^\ell}{\partial w_{jk}^\ell} = \delta_j^\ell \, x_k^{\ell-1}. \tag{43}$$

Batch dimensions are omitted in theoretical derivations for clarity.

5.2 Gradient Flow in Residual Blocks

Consider a residual block defined as

$$\mathbf{h}_\ell = \mathbf{h}_{\ell-1} + \mathbf{F}_\ell(\mathbf{h}_{\ell-1}), \tag{44}$$

where $\mathbf{F}_\ell$ is the residual function of layer ℓ, and $\mathbf{h}_{\ell-1}$ is the input to the block. Let $\mathcal{L}$ denote the loss function.

To compute the gradient of the loss with respect to the input $\mathbf{h}_{\ell-1}$, we apply the chain rule:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_{\ell-1}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_\ell} \cdot \frac{\partial \mathbf{h}_\ell}{\partial \mathbf{h}_{\ell-1}}. \tag{45}$$

Next, differentiate $\mathbf{h}_\ell$ with respect to $\mathbf{h}_{\ell-1}$:

$$\frac{\partial \mathbf{h}_\ell}{\partial \mathbf{h}_{\ell-1}} = \frac{\partial}{\partial \mathbf{h}_{\ell-1}} \Big[\mathbf{h}_{\ell-1} + \mathbf{F}_\ell(\mathbf{h}_{\ell-1})\Big] \tag{46}$$

$$= \mathbf{I} + \frac{\partial \mathbf{F}_\ell}{\partial \mathbf{h}_{\ell-1}}, \tag{47}$$

where $\mathbf{I}$ is the identity matrix, since $\frac{\partial \mathbf{h}_{\ell-1}}{\partial \mathbf{h}_{\ell-1}} = \mathbf{I}$.

Substituting back into the chain rule, the gradient of the loss becomes:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_{\ell-1}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_\ell} \cdot \left(\mathbf{I} + \frac{\partial \mathbf{F}_\ell}{\partial \mathbf{h}_{\ell-1}}\right). \tag{48}$$

If we consider **element-wise scalar notation** or assume that the gradient propagates diagonally, this can be simplified as:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_{\ell-1}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_\ell} \left(1 + \frac{\partial \mathbf{F}_\ell}{\partial \mathbf{h}_{\ell-1}}\right). \tag{49}$$

This clearly shows how the **identity skip connection** allows gradients to flow directly through the network, mitigating the vanishing gradient problem.

References

Chen, R. T. Q., Rubanova, Y., Bettencourt, J., and Duvenaud, D. (2018). Neural ordinary differential equations. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 31.

Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pages 4171–4186.

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

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

Nielsen, M. A. (2015). *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA.

Rubanova, Y., Chen, R. T., and Duvenaud, D. K. (2019). Latent ordinary differential equations for irregularly-sampled time series. *Advances in neural information processing systems*, 32.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323:533–536.

Schmidt, R. M. (1912). Recurrent neural networks (rnns): A gentle introduction and overview (2019). *arXiv preprint arXiv:1912.05911*.

Turner, R. E. (2023). An introduction to transformers. *arXiv preprint arXiv:2304.10557*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.

Werbos, P. J. (1990). Backpropagation through time: What it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560.

Yang, Y., Uy, M. C. S., and Huang, A. (2020). Finbert: A pretrained language model for financial communications. *arXiv preprint arXiv:2006.08097*.