A Complete Guide to RNNs, from First Principles to Backpropagation Through Time

Har Ashish Arora Dhairya Kuchhal

June 20, 2025

Abstract

Recurrent Neural Networks (RNNs) are foundational models for learning from sequential data, yet are not often given a proper mathematical treatment along with a walk-through of implementation, and this may feel discouraging for beginners. This paper presents a comprehensive, from-scratch derivation and implementation of the simple Elman RNN, aimed at building both rigorous mathematical understanding and practical intuition. We begin with the forward propagation equations, motivate their design, and proceed to derive Back-propagation Through Time (BPTT) from first principles using matrix calculus, with special attention to shape consistency, the chain rule, and the interpretation of gradient flow.

We also address key numerical challenges such as vanishing and exploding gradients, demonstrate their effects on training, and implement gradient clipping as a strategy to overcome this problem. Additionally, we provide a probabilistic justification for the commonly used squared error loss function via maximum likelihood estimation, linking neural network training with foundational ideas in statistics and probability.

The paper concludes with experiments on toggle and sinusoidal sequences, showing how even basic RNNs like these can learn simple temporal patterns when carefully implemented and trained. This work serves as both a tutorial and a reference, bridging theoretical foundations and practical deep learning, and sets the stage for exploring more advanced architectures such as LSTMs and transformers in the future.

Contents

1 Introduction

Despite the widespread use of Recurrent Neural Networks (RNNs) in natural language processing, time-series forecasting, and other areas, we found a lack of literature that bridges rigorous mathematical understanding with clean, from-scratch code implementations. In particular, most educational material glosses over matrix calculus subtleties and implementational challenges.

Recurrent Neural Networks are a class of neural networks that contain feedback loops and thus, they contain the ability to be able to track changes through time. For example, an input at a certain time t can be made to loop back around to the hidden layers and can be used to also determine the output at time $t + 1$ - allowing these neural networks to learn time-dependent outputs and signals, useful in various real-life applications.

This short paper aims to implement the Elman RNN [1], a simple Recurrent Neural Network architecture, from scratch - walking through the forward and backward passes with clear mathematics, and a side-by-side NumPy-based implementation. We train the model on a basic toggle-based temporal signal (a minimal task with alternating outputs) to focus on the model mechanics without unnecessary data complexity.

Initially we explore the Elman RNN architecture, followed by a high-level review of back-propagation and gradient descent for training. We then develop the mathematical formulation of Back Propagation Through Time (BPTT), interleaving each derivation step with its corresponding code. We conclude with a discussion on limitations, potential extensions, and final observations.

Our goal is to create a document that is simultaneously mathematical, intuitive, and implementable — useful for beginners and those attempting to build a fundamental understanding of the inner workings of RNN.

2 Intuition

Before we dive into studying the architecture of RNNs, it's helpful to get an intuition of what purposes RNNs serve.

Recurrent neural networks intrinsically use a recurrence relation (a self-feedback loop) in order to process sequential data. The main purpose of an RNN is to model sequential data in order to predict what will happen next in the sequence or to sometimes understand the significance/nuance of the order.

Say we have some sequentially ordered data, starting from time $t = 1$ to some time $t = T$. To model a sequence in this neural network, we will train the model sequentially, feeding the input data points from the start, that is, the input data point at $t = 1$ (say $\mathbf{x}_1$) to the input data point at $t = T$ (say $\mathbf{x}_T$).

It might be a natural instinct to ask - "How can I achieve *anything* when I train my model sequentially?" As per the classic ANNs that one studies when they are starting to learn about Neural Networks, the gradient descent should remain the same. This is precisely where the 'recurrence' in Recurrent Neural Networks strikes! As we train our data sequentially, we also take input from the data that we have trained earlier in the sequence so that it can affect our outcome of what's next in the sequence. Then, using the loss from the predicted output we get and what the output should have been, we calculate how to tweak the parameters of the neural network itself, as well as the parameters which control the affect of "past" data points on the current data point.

3 Architecture

3.1 Basic Architecture

This figure below shows the basic architecture of Elman RNNs.

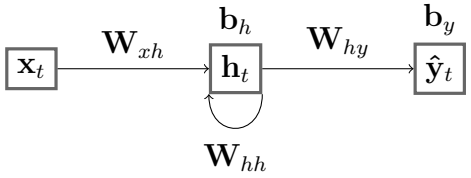

Figure 1: Basic architecture of Elman RNNs.

Not a pretty difficult model! As is visible, we have an input vector $\mathbf{x}_t$ at time t - this is where we feed the sequential input, going to the hidden layer $\mathbf{h}_t$ after being multiplied by a weight matrix $\mathbf{W}_{xh}$, a bias $\mathbf{b}_h$ being added, and a non-linear activation function such as $\sigma(x)$ (sigmoid) or tanh being applied, and the output layer being $\hat{\mathbf{y}}_t$, after the output at the hidden layer $\mathbf{h}_t$ gets multiplied by another weight matrix $\mathbf{W}_{hy}$ and a bias $\mathbf{b}_y$ is added to get the final output.

So far, this is no different from a standard feedforward neural network. The **twist** lies in the recurrence: the hidden state $\mathbf{h}_t$ is also influenced by the hidden state from the previous time step, $\mathbf{h}_{t-1}$, via the recurrent weight matrix $\mathbf{W}_{hh}$. This recurrent connection allows the model to retain information over time, eventually learning sequences of data and the ability to predict future outputs in the sequences.

Each of the components in the figure are vector-valued. Throughout this paper, we assume:

$$\mathbf{x}_t \in \mathbb{R}^n, \quad \mathbf{h}_t \in \mathbb{R}^m, \quad \hat{\mathbf{y}}_t \in \mathbb{R}^k$$

and hence,

$$\mathbf{W}_{xh} \in \mathbb{R}^{m \times n}, \quad \mathbf{W}_{hh} \in \mathbb{R}^{m \times m}, \quad \mathbf{W}_{hy} \in \mathbb{R}^{k \times m}$$

with bias terms:

$$\mathbf{b}_h \in \mathbb{R}^m, \quad \mathbf{b}_y \in \mathbb{R}^k.$$

Further, we also assume that $\hat{\mathbf{y}}_t$ is the value that the model predicts, and that $\mathbf{y}_t$ is the true value or the objective truth of the output. Using these two values, we can model the loss function, which we will see later, and accordingly tweak the parameters of our model.

What's going on here? We are trying to essentially predict the next value in a sequence. For example, if you had a series of sines as your $\mathbf{x}_t$ as $\sin(0.1), \sin(0.2), \ldots, \sin(T)$ as your input, you are trying to predict the next value in that sequence - for example, in the end, you would try to predict $\hat{\mathbf{y}}_t$ as $\sin(0.2), \sin(0.3), \ldots, \sin(T+1)$ and this $\sin(T+1)$ is essentially what we want to predict, and is what we are training the model to output. You will use the intermediate values to train your network and reduce a loss between the true values and the predicted values, so your model can accurately learn the time-varying signal, from the previous steps.

3.2 Unrolled Expanded Architecture

The above figure shows the expanded view of the same Elman Recurrent Neural Network. For clarity, we have shown the previous time step, from which the information is being

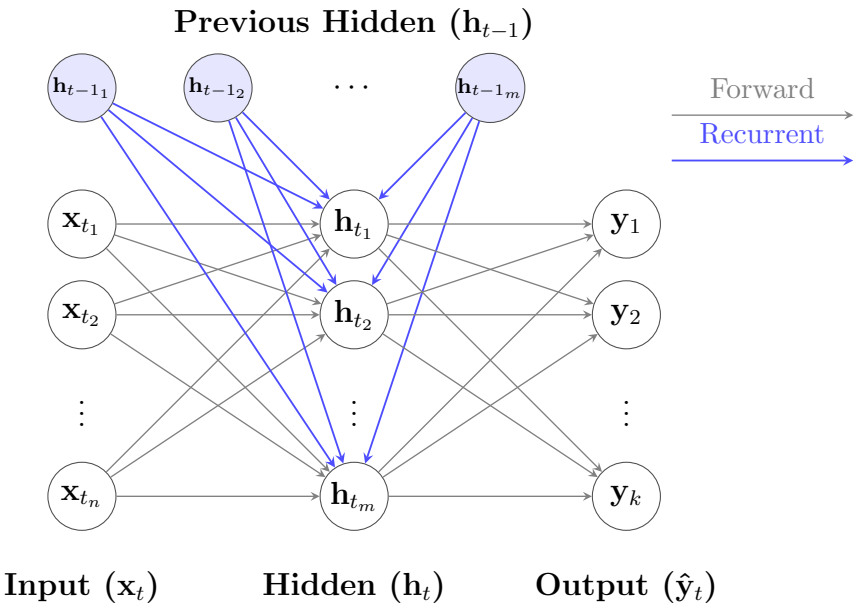

Figure 2: Neuron-level diagram of a single time step in an Elman RNN showing recurrent connections. Each hidden neuron receives inputs from all input neurons and all previous hidden state neurons. Each output neuron receives inputs from all current hidden neurons.

passed recursively to the current time step, that is, from $\mathbf{h}_{t-1}$ to $\mathbf{h}_t$, as a separate layer. This fact will be important for later, when we unroll the Neural Network over all time steps to pass information forward (feed-forward) and back-propagate the errors to tweak the values of our parameters. Going back all the way to when we compute $\mathbf{h}_1$ - you may wonder how we get $\mathbf{h}_0$! The answer is that this hidden state at time $t = 0$, denoted as $\mathbf{h}_0$, is typically initialised to the zero vector.

Note that all weights and biases are shared across time steps. This weight sharing is what allows the network to generalise across different temporal positions in the sequence, effectively treating each time step as statistically similar but contextually dependent.

3.3 Forward Pass Mathematics

We conclude this section with a simple understanding and statement of the mathematics and the notation we will use for the remaining portion of this paper, particularly that for the forward pass. This will cement our understanding of the forward pass, especially as it lays the foundation for the derivations in Backpropagation Through Time (BPTT).

At each time step t, the Elman RNN performs the following operations:

$$\begin{aligned}\mathbf{a}_t &= \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h \\ \mathbf{h}_t &= f(\mathbf{a}_t) \\ \hat{\mathbf{y}}_t &= \mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y\end{aligned}$$

Here:

- $\mathbf{x}_t \in \mathbb{R}^n$: input vector at time t
- $\mathbf{h}_{t-1} \in \mathbb{R}^m$: hidden state from the previous time step

- $\mathbf{a}_t \in \mathbb{R}^m$: pre-activation input to the hidden layer
- $\mathbf{h}_t \in \mathbb{R}^m$: current hidden state (post-activation)
- $\hat{\mathbf{y}}_t \in \mathbb{R}^k$: output at time t
- f: element-wise activation function (typically sigmoid or tanh)

This sequence of operations encodes how input and past information combine to produce the current hidden state and output. These equations form the computational graph over which gradients will be back-propagated in the training process.

4 Forward Propagation

We have already seen the equations which constitute the forward pass - let's now focus on their interpretation and implementation using NumPy.

4.1 Interpretation

Let's break down the equations intuitively:

- The current hidden state $\mathbf{h}_t$ is influenced by both the current input $\mathbf{x}_t$ and the previous hidden state $\mathbf{h}_{t-1}$ - this is what gives the RNN its sequential power and allows it to have a certain type of "memory" of previous states.
- The activation function (commonly sigmoid or tanh) introduces non-linearity, enabling the network to model complex relationships.
- Finally, the hidden state $\mathbf{h}_t$ is linearly transformed to produce the output $\hat{\mathbf{y}}_t$, which can then be compared to the true output.

This computation is repeated across the entire sequence, and thus the outputs $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_T$ can be interpreted as being generated step-by-step from the input sequence $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t$, while leveraging learned context from earlier steps.

If we stack all time steps together, forward propagation becomes a sequence of chained operations through time:

$$\begin{cases} \mathbf{a}_t = \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h \\ \mathbf{h}_t = \sigma(\mathbf{a}_t) \\ \mathbf{y}_t = \mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y \end{cases} \qquad \text{for } t = 1, 2, \ldots, T$$

Here, $\mathbf{a}_t$ is the pre-activation (or "logit") value at the hidden layer before applying the nonlinearity.

As a final sanity check, here are all the shapes of the matrices and vectors so far:

Variable	Shape
$\mathbf{x}_t$	$(n, 1)$
$\mathbf{h}_t$	$(m, 1)$
$\mathbf{y}_t$	$(k, 1)$
$\mathbf{W}_{xh}$	(m, n)
$\mathbf{W}_{hh}$	(m, m)
$\mathbf{W}_{hy}$	(k, m)
$\mathbf{b}_h$	$(m, 1)$
$\mathbf{b}_y$	$(k, 1)$

4.2 Implementation

Now that we've got the basic mathematics out of the way, let's begin implementing. In this section, we will initialise our variables, define the forward pass function, and also a non-linearity (the sigmoid activation function).

4.2.1 Weights and Biases Initialisation

We first begin by importing NumPy:

```
import numpy as np
```

We then initialise our weight matrices and bias vectors. We initialise the weight matrices to small random values through **Xavier (Glorot) Initialisation**.

Xavier initialisation essentially tells us to initialise our matrices with a mean 0 and variance $\frac{1}{n_{\text{in}}+n_{\text{out}}}$ where n_{in} is the number of neurons being fed into the layer and n_{out} is the number of output neurons. That is, for a layer with n_{in} inputs and n_{out} outputs:

$$W \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}} + n_{\text{out}}}\right)$$

You may wonder why this is necessary. Suppose, on the contrary, one initialised *all* the weights to the same value, say 0, without loss of generality. Then, two neurons in the same layer would produce identical outputs, and thus, in training, undergo identical updates because they receive identical gradients, and thus, as a result, they remain identical during training, and learn the same features. This is called the **Symmetry Problem**, and to overcome this problem, we initialise each weight in the network with a random value drawn independently, from a Gaussian Distribution, say. This ensures that training can become effective.

Another advantage of Xavier initialisation is that it controls the variance effectively so that the signal doesn't grow too much or shrink too much as it passes through each layer forwards and backwards, enabling faster converge and more stable training, reducing to a certain extent the problems of **Vanishing/Exploding Gradients**, which we will explore in detail later.

We can initialise the biases to the 0 vector since they do not participate in symmetry breaking, and thus this is a safe and conventional default.

```
def initialise(n, m, k):
    W_xh = np.random.randn(m, n) * np.sqrt(2.0 / (m + n))
    W_hh = np.random.randn(m, m) * np.sqrt(2.0 / (2 * m))
```

```
    W_hy = np.random.randn(k, m) * np.sqrt(2.0 / (k + m))
    b_h = np.zeros((m, 1))
    b_y = np.zeros((k, 1))
    return W_xh, W_hh, W_hy, b_h, b_y
```

Here n, m, k hold their usual meanings, that is, the dimensions of the input, hidden, and output layer respectively. We are explicitly casting the bias vectors as column vectors to avoid shape mismatch problems down the line (we spent about 3 hours debugging those out).

4.2.2 The Forward Pass

It is now fairly simple to define the element-wise sigmoid activation functions and the tanh activation functions:

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def tanh(x):
    return np.tanh(x)
```

A quick note on the sigmoid and tanh activation functions: the tanh non-linearity is zero-centred and the sigmoid function is not (centred at 0.5). In theory, both work, but many texts recommend the use of tanh. We will continue to work with sigmoid because of its simplicity in use here.

4.2.3 Unrolled RNN across time

It would be helpful to visualise the forward pass through time for a specific input $\mathbf{x}_t$ through a figure as shown below.

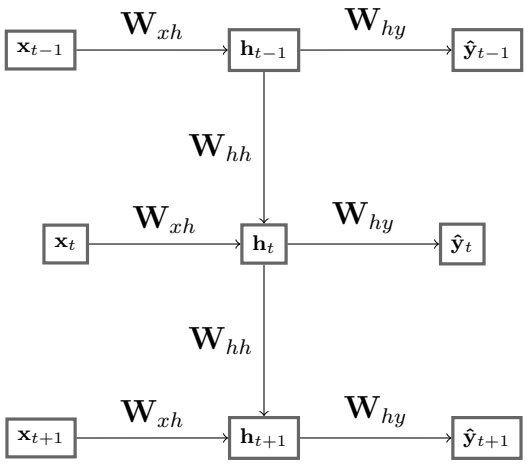

Figure 3: Unrolled Forward Pass through Time

This shows the inputs across three general time steps, $t = t, t = t - 1,$ and $t = t + 1$. Unrolling the RNN allows for a clearer visual picture of what's going on: each $\mathbf{h}_t$ is passed forward to calculate both its $\hat{\mathbf{y}}_t$ and further $\mathbf{h}_{t+1}$ which is what allows the network to "remember", in a way. It become easy to code each sequential pass now, as shown below.

We will now define the forward pass, the core part of our forward-propagation:

```
def forward_pass(W_xh, W_hh, W_hy, b_h, b_y, x_t, h_tminus1):
    x_t = x_t.reshape(-1, 1)
    a_t = W_xh @ x_t + W_hh @ h_tminus1 + b_h
    h_t = sigmoid(a_t)
    y_pred_t = W_hy @ h_t + b_y
    return a_t, h_t, y_pred_t
```

Here, we are explicitly reshaping the input vector at each time $\mathbf{x}_t$ to its corresponding column vector - that is, we are explicitly making it a 2^{nd} rank tensor instead of just a 1^{st} rank tensor, to avoid problems in matrix multiplication down the road. Also, `\mathbf{y}_pred_t` is used in the code to denote our prediction at time t, that is, $\hat{\mathbf{y}}_t$.

4.3 Forward Propagation Through Time

So far, we've only handled a single time step of the RNN — computing the hidden state $\mathbf{h}_t$ and output $\hat{\mathbf{y}}_t$ from input $\mathbf{x}_t$ and the previous hidden state $\mathbf{h}_{t-1}$. We've mentioned though that RNNs are meant to handle sequences!

Now, we extend the forward pass to handle an entire input sequence $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t\}$ and compute the corresponding outputs $\{\hat{\mathbf{y}}_1, \hat{\mathbf{y}}_2, \ldots, \hat{\mathbf{y}}_T\}$. At each step, the current hidden state $\mathbf{h}_t$ depends on both $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$, and the output $\hat{\mathbf{y}}_t$ depends on $\mathbf{h}_t$. We can chain this across time and create a whole loop and storing the intermediate values:

- Inputs: $\mathbf{x}_t$
- Pre-activation: $\mathbf{a}_t$
- Hidden state: $\mathbf{h}_t$
- Prediction: $\hat{\mathbf{y}}_t$

We store these intermediate values because they will be needed in the future, specifically, in back-propagation through time.

We start with an initial hidden state $\mathbf{h}_0$, which is typically initialised to a zero vector of shape $(m, 1)$ where m is the hidden size.

4.3.1 Implementation

The following code implements the full sequence-level forward pass. Note that here, X is the input sequence containing the whole sequence from time $t = 1$ to time $t = T$. Another subtlety to note is that computer systems are 0-indexed, whereas throughout the mathematics so far, we have been indexing our t from $t = 1$. It doesn't cause much issue, but is one thing to note.

```
def rnn_forward(X, h_0, W_xh, W_hh, W_hy, b_h, b_y): # X is input
    T = len(X) # Number of time steps

    x_list = []
    a_list = []
    h_list = [h_0] # Start with initial hidden state
    y_pred_list = []
```

```
8
9       for t in range(T):
10          x_t = X[t].reshape(-1, 1)
11          h_tminus1 = h_list[-1]
12          x_list.append(x_t)
13          a_t, h_t, y_pred_t = forward_pass(W_xh, W_hh, W_hy,
14                                            b_h, b_y, x_t, h_tminus1)
15          a_list.append(a_t)
16          h_list.append(h_t)
17          y_pred_list.append(y_pred_t)
18
19      return x_list, a_list, h_list, y_pred_list
```

A few important details:

- We are explicitly reshaping $\mathbf{x}_t$ to a column vector to ensure matrix multiplication compatibility and future problems down the line when we run our code.
- We collect each intermediate variable in a list to use later for backpropagation - this will become clear when touch on the mathematics and the matrix calculus for BPTT.
- The output list `y_pred_list` gives us all the predicted outputs $\hat{\mathbf{y}}_1, \hat{\mathbf{y}}_2, \ldots, \hat{\mathbf{y}}_T$ corresponding to the input sequence.
- The line `h_tminus1 = h_list[-1]` directly takes the last stored value of $\mathbf{h}_t$, that is, $\mathbf{h}_{t-1}$.

This forward pass enables the network to maintain a form of memory across time, as each $\mathbf{h}_t$ depends on its own past values.

A quick explanation as to what this code is doing: For each $\mathbf{x}_t$ that we input into the function, we call the `forward_pass()` function to return to us three specific values: $\mathbf{a}_t, \mathbf{h}_t,$ and, $\hat{\mathbf{y}}_t$. These values are then appended directly into their respective lists and stored in the memory for later use, and we are returning these lists, along with `x_list`. This loops over for all the sequential inputs, and through this, the model predicts the output which it thinks is the correct answer.

We are now ready to go over the necessary matrix calculus which will be required for the implementation of Back-Propagation Through Time, BPTT.

5 Matrix Calculus and Differentiation

The authors themselves encountered a lot of problems while going through the rules for matrix calculus. Here, assuming a basic knowledge of the rules of matrices and Linear Algebra, we continue and lay down the foundations of the rules for matrix calculus that we will need for Back Propagation Through Time (BPTT) and machine learning and deep learning in general.

5.1 From Scalar to Matrix Derivatives: Building Intuition

We already know how to find derivatives like $\frac{d}{dx}(x^2) = 2x$. In machine learning, we need to essentially extend this concept to work with vectors, matrices, and sometimes higher-order tensors. The noteworthy thing here is that **we're always computing how a scalar loss function changes with respect to our parameters**.

5.1.1 The Gradient: Derivatives of Scalars with Respect to Vectors

Consider a scalar function L that depends on a vector $\mathbf{w} = [w_1, w_2, \ldots, w_n]^T$. The **gradient** $\nabla_{\mathbf{w}} L$ is simply the vector of all partial derivatives:

$$\nabla_{\mathbf{w}} L = \begin{bmatrix} \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial w_2} \\ \vdots \\ \frac{\partial L}{\partial w_n} \end{bmatrix}$$

Example: If $L = w_1^2 + 3w_2$, then $\nabla_{\mathbf{w}} L = \begin{bmatrix} 2w_1 \\ 3 \end{bmatrix}$.

5.1.2 Derivatives of Scalars with Respect to Matrices

When our parameters form a matrix $\mathbf{W}$ (as in the case of our weight matrices), the derivative $\frac{\partial L}{\partial \mathbf{W}}$ is a matrix *of the same shape* as $\mathbf{W}$, where each element is:

$$\left(\frac{\partial L}{\partial \mathbf{W}}\right)_{ij} = \frac{\partial L}{\partial W_{ij}}$$

This is intuitive, since each parameter gets its own partial derivative, arranged in the same structure as the original matrix, and thus forms a pretty natural extension of the gradient, and we can then denote $\frac{\partial L}{\partial \mathbf{W}} = \nabla_{\mathbf{W}} L$.

Example: Let $\mathbf{W}$ be a matrix defined as:

$$\mathbf{W} = \begin{bmatrix} w_1 & w_2 \\ w_3 & w_4 \end{bmatrix}$$

and let $L = w_1^3 + 3w_2^2 + \frac{w_3}{w_4}$. Then the gradient is

$$\nabla_{\mathbf{W}} L = \begin{bmatrix} 3w_1^2 & 6w_2 \\ \frac{1}{w_4} & -\frac{w_3}{w_4^2} \end{bmatrix}$$

Essentially, the gradient of a matrix is like asking "how does the loss change when I vary *each* entry of the matrix?" Each entry gives how the loss changes when you change just *that* particular entry.

5.1.3 Derivatives of Tensors with Respect to other Tensors

Vectors are first-order tensors, and matrices are second-order tensors. As an intuition pump, first let us consider the Jacobian matrix, which is the derivative of a vector (say $\mathbf{a} \in \mathbb{R}^m$) with respect to another vector (say $\mathbf{b} \in \mathbb{R}^n$):

$$\frac{\partial \mathbf{a}}{\partial \mathbf{b}} = \begin{bmatrix} \frac{\partial a_1}{\partial b_1} & \cdots & \frac{\partial a_1}{\partial b_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial a_m}{\partial b_1} & \cdots & \frac{\partial a_m}{\partial b_n} \end{bmatrix} \in \mathbb{R}^{m \times n}$$

Essentially, this is what is happening:

$$\frac{\partial \mathbf{a}}{\partial \mathbf{b}} = \frac{\partial a_i}{\partial b_j}$$

This is well-defined and manageable. However, in deep learning, especially in back-propagation through time or in back-propagation in general, we are frequently working with matrices, and even though directly we want the derivative of a scalar function with respect to the matrices, which is manageable, while using the chain rule (as you'll see later), we may need to calculate the derivative of a tensor with respect to a tensor.

Let us suppose we want to take the derivative of a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ with respect to another tensor $\mathbf{B} \in \mathbb{R}^{p \times q}$. Then the full derivative is, in general, a 4th-order tensor:

$$\frac{\partial \mathbf{A}_{ij}}{\partial \mathbf{B}_{uv}} \quad \text{for all } i, j, u, v$$

This quickly becomes intractable:

- The number of elements in the Jacobian-like object grows multiplicatively with the dimensions.
- Visualising or storing these high-rank derivatives is infeasible in practice.
- The notation and indexing quickly become unreadable and error-prone.

Therefore, in practice, we rarely compute full tensor-tensor derivatives explicitly. Instead, we rely on the following techniques, which is by no means an exhaustive list.

- **Vectorization**: Flattening tensors into vectors and using matrix calculus identities to compute derivatives more compactly.
- **Gradient contraction**: Often we are only interested in contractions like $\frac{\partial L}{\partial \mathcal{B}}$, where L is a scalar loss function. In such cases, we compute gradients directly using chain rule structures that avoid constructing higher-order tensors altogether. **This is what we will be focusing on here.**
- **Index notation and Einstein summation**: Writing derivatives in terms of indexed expressions or Einstein notation allows precise yet compact representation, especially useful when dealing with batched operations and broadcasting.

In summary, while the definition of derivatives between higher-order tensors exists mathematically, machine learning relies on structural simplifications and computational tricks to make such derivatives practical.

5.2 The Golden Rule for Machine Learning

In neural networks, we follow this fundamental principle:

> **The derivative of a scalar with respect to any tensor has the same shape as that tensor.**

This means:

- $\frac{\partial L}{\partial \mathbf{w}} \in \mathbb{R}^n$ if $\mathbf{w} \in \mathbb{R}^n$
- $\frac{\partial L}{\partial \mathbf{W}} \in \mathbb{R}^{m \times n}$ if $\mathbf{W} \in \mathbb{R}^{m \times n}$
- $\frac{\partial L}{\partial b} \in \mathbb{R}$ if $b \in \mathbb{R}$

This rule ensures that gradient updates make sense: we can subtract $\eta \frac{\partial L}{\partial \mathbf{W}}$ from $\mathbf{W}$ during gradient descent. We will use this rule liberally throughout to maintain shapes and dimensions, especially when we are utilising the chain rule.

5.3 Essential Rules for Neural Networks

5.3.1 Chain Rule for Matrices

The chain rule extends not-so-naturally to matrix operations. If L depends on $\mathbf{z}$ (which may be a matrix or a vector), which depends on $\mathbf{W}$ (a matrix or a vector):

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{W}}$$

However, when dealing with matrices, we must be careful about dimensions and use appropriate matrix multiplication. In general, try to make the commutation of the multiplications and the transposes of each matrix so that the shape of the product of the two matches the required shape, that is, that of $\mathbf{W}$.

A special case given in The Matrix Cookbook [5], when say, L is a scalar function of $\mathbf{u}$ and $\mathbf{u}$ is a function of a matrix say $\mathbf{X}$, then

$$\left(\frac{\partial L}{\partial \mathbf{X}}\right)_{ij} = \mathrm{Tr}\left[\left(\frac{\partial L}{\partial \mathbf{u}}\right)^T \frac{\partial \mathbf{u}}{\partial \mathbf{X}_{ij}}\right]$$

This formula is sometimes useful and allows us to use the chain rule without explicitly calculating the higher-order tensors.

5.3.2 Derivatives of Common Operations

For RNN backpropagation, we need these key derivatives:

1. Linear Transformation: If $\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$, then:

$$\frac{\partial \mathbf{z}}{\partial \mathbf{W}} = \mathbf{x}^T$$
$$\frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \mathbf{W}^T$$
$$\frac{\partial \mathbf{z}}{\partial \mathbf{b}} = \mathbf{I}$$

An important point to note is that these derivatives are all contextually dependent, and we make it our aim to follow the Golden Rule. For example, the form of these derivatives can change, for example, $\frac{\partial \mathbf{z}}{\partial \mathbf{W}}$ may equal $\mathbf{x}^T$ or $\mathbf{x}$ depending on the shape that we want to maintain. For this purpose, we try to keep our calculations as general as possible to avoid any errors.

Why does this happen? This is because we are trying to avoid tensor calculus, and thus, we need to make use of a few general rules, especially for machine learning.

2. Element-wise Functions: If $\mathbf{h} = \sigma(\mathbf{a})$ where σ is applied element-wise:

$$\frac{\partial \mathbf{h}}{\partial \mathbf{a}} = \text{diag}(\sigma'(\mathbf{a}))$$

The above expression is actually the full Jacobian Matrix - giving a 2^{nd} order tensor as described earlier. However, when using this in chain rule applications, where we will frequently need it, we typically perform the element-wise (Hadamard) product. (And if you think intuitively, during multiplication with a diagonal matrix, this is what is happening anyway!) While the above is technically correct, we use it in the following form:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}} \odot \sigma'(\mathbf{a})$$

Here, $\odot$ denotes the Hadamard (element-wise) product.

5.4 Why These Rules Work: A Dimensional Analysis

A useful sanity check is to verify that dimensions match. For example, when we will be deriving BPTT, we will be checking some dimensions of the following form:

- $\mathbf{W}_{xh} \in \mathbb{R}^{m \times n} \Rightarrow \frac{\partial L}{\partial \mathbf{W}_{xh}} \in \mathbb{R}^{m \times n}$
- $\frac{\partial L}{\partial \mathbf{a}_t} \in \mathbb{R}^{m \times 1}, \quad \mathbf{x}_t^T \in \mathbb{R}^{1 \times n}$
- Outer product: $\mathbb{R}^{m \times 1} \cdot \mathbb{R}^{1 \times n} = \mathbb{R}^{m \times n}$ ✓

This dimensional consistency provides confidence that our derivations are correct.

5.5 The Path Forward

With these matrix calculus foundations, we're ready to tackle the full derivation of Back-propagation Through Time (BPTT). The key insight is that BPTT applies these same rules, but we must carefully track how gradients flow backward through time, accumulating contributions from multiple time steps.

In the next section, we'll see how the recurrent connections in RNNs create additional gradient paths that don't exist in feedforward networks, leading to the characteristic challenges and solutions of training recurrent architectures.

6 Backpropagation Through Time (BPTT)

Training a Recurrent Neural Network (RNN) involves computing the gradients of a loss function with respect to all of our trainable parameters (that is, our weights and biases) over an entire input sequence. This is known as **Backpropagation Through Time (BPTT)**.

In a standard feedforward neural network, we generally backpropogate our errors through the layers and try to reduce those errors through gradient descent. However, in our RNN, since we are not only passing in an input we are also trying to train a netowrk through *time*, we also have to *backpropagate the errors through* ***time***. We'll see how that is done below, with sufficient diagrams for explanation.

6.1 Loss Function and Accumulation

Since throughout the majority of this paper we have been trying to go for regression, let us continue forward and define the loss function that we will be using. A probabilistic interpretation of this loss function will be given in a section later.

The loss at *a particular time step* t is defined as:

$$\mathcal{L}_t = \frac{1}{2}\|\hat{\mathbf{y}}_t - \mathbf{y}_t\|_2^2$$

Here $\hat{y}_t$ is the predicted output and y_t is the ground truth. We square the L_2 norm to keep the loss differentiable and smooth.

Let's quickly implement this:

```
def loss_t(y_pred_t, y_t):
    return np.sum((y_pred_t - y_t) ** 2) / 2
```

Typically, we want to train the model over *all* time steps to ensure the model learns the entire sequence, so we sum over all the losses at each time step. Since the RNN produces an output at each time step, the total loss $\mathcal{L}$ is the **sum of the losses** at each time step:

$$\mathcal{L} = \sum_{t=1}^{T} \mathcal{L}_t = \sum_{t=1}^{T} \frac{1}{2}\|\hat{\mathbf{y}}_t - \mathbf{y}_t\|_2^2$$

To train our RNN, we must compute the gradients of this total loss $\mathcal{L}$ with respect to the weights and biases, and update them using gradient descent.

Since the parameters $W_{xh}, W_{hh}, W_{hy}, b_h, b_y$ are shared across all time steps, we **accumulate the gradients from each time step**. This means that for each parameter θ, the total gradient is:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t}{\partial \theta}$$

We will not be showing this explicitly after every derivation, but it will be shown clearly in the code and explained along with it.

6.1.1 Why do we need to accumulate gradients?

Each parameter (like W_{xh}) is used *at every time step* because the weights are shared across the sequence. Therefore, the total loss depends on the parameter's effect at every time step. Hence, we must sum the gradients:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t}{\partial \theta}$$

This is exactly why in code, as we will see further on, you loop backward from T to 1 and add up the gradients.

6.2 Activation Function Derivatives

We will need these derivatives when we are performing the derivations for the back-propagation, so it is best to implement them now.

We had already seen in the forward pass that our activation function was the sigmoid activation function applied element-wise, that is:

$$\mathbf{h}_t = \sigma(\mathbf{a}_t) = \frac{1}{1 + e^{-\mathbf{a}_t}}$$

The derivative of the sigmoid activation function as given in 5.3.2:

$$\sigma'(\mathbf{a}_t) = \sigma(\mathbf{a}_t) \odot (1 - \sigma(\mathbf{a}_t)) = \mathbf{h}_t \odot (1 - \mathbf{h}_t)$$

and this itself is applied element-wise whenever we will use the chain rule, as in further up ahead.

If we were using tanh instead, its derivative would be:

$$\tanh'(\mathbf{a}_t) = 1 - \tanh^2(\mathbf{a}_t)$$

where 1 is accordingly broadcasted to the size of $\mathbf{a}_t$, inthis case, $\mathbb{R}^m$.

6.2.1 Implementation

We can swiftly implement these two lines in our code:

```
def sigmoid_diff(x):
    return sigmoid(x) * (1 - sigmoid(x))
```

```
def tanh_diff(x):
    return 1 - (tanh(x)) ** 2
```

6.3 Backpropagation Equations

Now we get to the real equations for backpropagation! We'll try to keep this as clear as possible, showing how the errors are flowing through the computational graph to get the gradients of the parameters, and also include sanity checks like those of matrix shapes as in 5.4. We'll keep a very minimal set of diagrams to enable you to show how the error is actually propagating backward. Starting from the loss at time t, define:

$$\delta_t^y = \frac{\partial \mathcal{L}_t}{\partial \hat{\mathbf{y}}_t} = \hat{\mathbf{y}}_t - \mathbf{y}_t$$

How? Well,

$$\mathcal{L}_t = \frac{1}{2}\|\hat{\mathbf{y}}_t - \mathbf{y}_t\|_2^2 = \frac{1}{2}(\hat{\mathbf{y}}_t - \mathbf{y}_t)^T(\hat{\mathbf{y}}_t - \mathbf{y}_t) = \sum_i^k \frac{1}{2}(\hat{\mathbf{y}}_t - \mathbf{y}_t)_i^2$$

since $(\hat{\mathbf{y}}_t - \mathbf{y}_t)^T(\hat{\mathbf{y}}_t - \mathbf{y}_t)$ is just a scalar. Then, since this is just the inner product, we can write:

$$\left(\frac{\partial \mathcal{L}_t}{\partial \hat{\mathbf{y}}_t}\right)_i = (\hat{\mathbf{y}}_t - \mathbf{y}_t)_i$$

and this gives the form we see above.

What's the interpretation for δ_t^y? This derivative tells us how much the predicted output $\hat{\mathbf{y}}_t$ differs from the true output $\mathbf{y}_t$, and it gives the direction in which the prediction must move to reduce the loss. This will also be used further when we compute the gradients of our parameters.

Let's see where this δ_t^y lives on the computational graph - this can be considered as a *variation* of sorts from the true output, and we will continue to hold onto this δ notation further, showing variations and errors in the backpropagation.

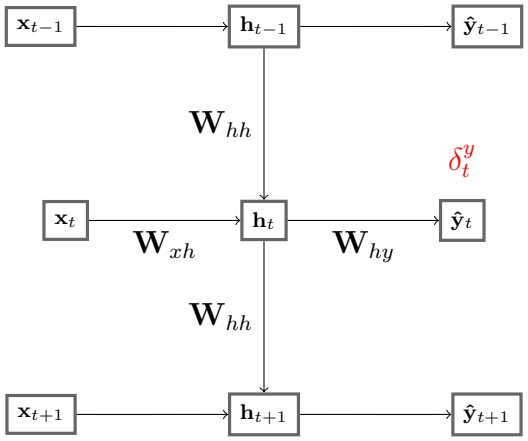

Figure 4: Variation in $\mathbf{y}_t$

This variation δ_t^y lives at the $\hat{\mathbf{y}}_t$ node on the computational node, showing the variation in $\mathbf{y}_t$ from the true value.

Quick Sanity Check: This variation in $\mathbf{y}_t$ is exactly the shape we expect, from both intuition and mathematics, as $\mathbb{R}^k$.

6.3.1 Output Layer Gradients

Let's now calculate the output layer gradients using the chain rule for gradients and our golden rule.

$$\begin{aligned}\frac{\partial \mathcal{L}_t}{\partial \mathbf{W}_{hy}} &= \frac{\partial \mathcal{L}_t}{\partial \hat{\mathbf{y}}_t} \cdot \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{W}_{hy}} \\ &= \delta_t^y \cdot \frac{\partial \left(\mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y\right)}{\partial \mathbf{W}_{hy}} \\ &= \delta_t^y \cdot \mathbf{h}_t^T\end{aligned}$$

where the second equality is arrived at from 5.3.2.
Similarly we can derive

$$\begin{aligned}\frac{\partial \mathcal{L}_t}{\partial \mathbf{b}_y} &= \frac{\partial \mathcal{L}_t}{\partial \hat{\mathbf{y}}_t} \cdot \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{b}_y} \\ &= \delta_t^y \cdot \frac{\partial \left(\mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y\right)}{\partial \mathbf{b}_y} \\ &= \delta_t^y\end{aligned}$$

where the second equality is again derived from 5.3.2.
Quick Sanity Check: These gradients $\nabla_{\mathbf{W_{hy}}}\mathcal{L}_t$ and $\nabla_{\mathbf{b_y}}\mathcal{L}_t$ are the shapes we expect, from both intuition and mathematics, as $\mathbb{R}^{k\times m}$, since we have an outer product between two vectors of shape $\mathbb{R}^k$ and $\mathbb{R}^m$ for $\nabla_{\mathbf{W_{hy}}}\mathcal{L}_t$ and we have $\mathbf{b}_y$ and δ_t^y are the same shape, $\mathbb{R}^k$.

6.3.2 Hidden Layer Gradients

The gradient with respect to $\mathbf{h}_t$, δ_t^h comes from flowing the variation from the future time state, δ_{t+1}^a and the current time state's output (via recurrence), as can be seen in the below figure, where the red-coloured terms represent the error signals. Thus,

$$\delta_t^h = \mathbf{W}_{hy}^T\delta_t^y + \mathbf{W}_{hh}^T\delta_{t+1}^a$$

where δ_{t+1}^a is the backpropagated error from the next time step.
Now apply the chain rule through the activation function:

$$\delta_t^a = \delta_t^h \odot \sigma'(\mathbf{a}_t)$$

How did we even arrive at this? Let's go through the derivation, step-by-step.
We have, for **only the current time step**:

$$\begin{aligned}\frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t} &= \frac{\partial \mathcal{L}_t}{\partial \hat{\mathbf{y}}_t} \cdot \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} \\ &= \mathbf{W}_{hy}^T\delta_t^y\end{aligned}$$

Why did we change the order? It's because of our golden rule 5.2 to maintain the shape of the derivative of a scalar function with respect to a matrix. In this case, δ_t^h can be interpreted similarly to δ_t^y, that is the *variation* or error in the real $\mathbf{h}_t$ compared to what the model predicts. Let's go through our sanity check once again, then through a quick intuitive understanding of why this should even be this way in the first place.

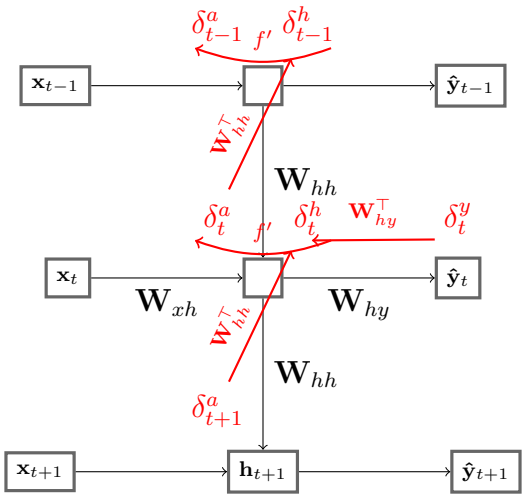

Figure 5: Error in hidden states and Pre-Activation States

Sanity Check: The shape of δ^h_t must be $\mathbb{R}^m$ - here, the shape of $\mathbf{W}^T_{hy}$ is $\mathbb{R}^{m\times k}$ and δ^y_t is $\mathbb{R}^k$, giving us our $\mathbb{R}^m$ shaped variation/error.

The intuition: This is the core part of the intuition behind the backpropagation and how we are making the errors flow backwards both through the layers and through time. Consider the state of the hidden layer, $\mathbf{h}_t$. We compute the state of the *next* hidden layer, that is, $\hat{\mathbf{y}}_t$ by multiplying the weight matrix $\mathbf{W}_{hy}$ (and also adding a bias but that is a property of the layer $\hat{\mathbf{y}}$) - it would only make sense, then, that going backwards, we would multiply $\mathbf{W}_{hy}$ to the error signal that we are generating to generate the new error signal! Hopefully this makes it intuitive, especially for when we are propagating the error back through the same layer.

Now, for the second term in δ^h_t, $\mathbf{W}^T_{hh}\delta^a_{t+1}$. Consider the same intuition we just applied - only in this case, since the current state $\mathbf{h}_t$ gets passed on to the pre-activation of the next state, $\mathbf{a}_{t+1}$ through the weight matrix $\mathbf{W}_{hh}$, it only makes sense for the error signal coming from the pre-activation state of the next time state, δ^a_{t+1}, to flow *back* through the same weight matrix, just transposed, as we had done earlier, and contribute to the error signal of the *post-activation* state of the current time step δ^h_t through the weight matrix $\mathbf{W}^T_{hh}$. Hopefully this makes sense. While the formal derivation for this second term can be derived through calculus, we leave that as an exercise to the reader.

Error Signal for the pre-activation state, δ^h_t: Let's go through this intuitively first this time, then move onto the formal derivation. Again, since the pre-activation state in the feedforward passes through the activation function $\sigma(\mathbf{a}_t)$, and is applied element-wise, it only makes sense for the **error signal** δ^h_t to be passed through its derivative to arrive at the error signal δ^a_t for the pre-activation state. Let's move onto the formal definition, now:

$$\begin{aligned}\delta^a_t &= \frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t}\cdot\frac{\partial \mathbf{h}_t}{\partial \mathbf{a}_t}\\ &= \delta^h_t \odot \sigma'(\mathbf{a}_t)\\ &= \delta^h_t \odot \sigma(\mathbf{a}_t)\odot(1-\sigma(\mathbf{a}_t))\end{aligned}$$

Why did we suddenly shift to the Hadamard (element-wise) product? Here's some intuition for you: Since the activation function is applied *element-wise* to the pre-activation

function, when we are propagating the errors backward, it makes sense to apply the same derivatives element-wise. You may also consult with section 5.3.2, where we explained why and how the element-wise product comes into play anyway.

You may ask why no terms from the future? This is because the pre-activation term $\mathbf{a}_t$ does not have any effect on the future states directly, only the post-activation state $\mathbf{h}_t$ does, and we have already included that through the error terms δ_t^h.

6.3.3 Input and Recurrent Weight Gradients

We are now ready to finally get the gradients for the remaining parameters, through the use of the error signals we have derived.

Let's first begin by finding the gradient for $\mathbf{W}_{xh}$: (this should be eerily similar to how we derived the output layer gradients - see 6.3.1).

$$\begin{aligned}\frac{\partial \mathcal{L}_t}{\partial \mathbf{W}_{xh}} &= \frac{\partial \mathcal{L}_t}{\partial \mathbf{a}_t} \cdot \frac{\partial \mathbf{a}_t}{\partial \mathbf{W}_{xh}} \\ &= \delta_t^a \cdot x_t^T\end{aligned}$$

Now we move onto $\mathbf{W}_{hh}$:

$$\begin{aligned}\frac{\partial \mathcal{L}_t}{\partial \mathbf{W}_{hh}} &= \frac{\partial \mathcal{L}_t}{\partial \mathbf{a}_t} \cdot \frac{\partial \mathbf{a}_t}{\partial \mathbf{W}_{hh}} \\ &= \delta_t^a \cdot \mathbf{h}_{t-1}^T\end{aligned}$$

And finally, we find the gradient for out last parameter, $\mathbf{b}_h$:

$$\begin{aligned}\frac{\partial \mathcal{L}_t}{\partial \mathbf{b}_h} &= \frac{\partial \mathcal{L}_t}{\partial \mathbf{a}_t} \cdot \frac{\partial \mathbf{a}_t}{\partial \mathbf{b}_h} \\ &= \delta_t^a\end{aligned}$$

We trust that by now, the reader is familiar with the concept of the sanity check and checking for shapes, and can verify for themselves that these multiplications are indeed correct.

6.4 Final forms of all error signals and gradients

This is essentially a summary section, where you can refer to all the error signals and gradients that we have derived in one convenient place.

$$\mathcal{L}_t = \frac{1}{2}\|\hat{\mathbf{y}}_t - \mathbf{y}_t\|_2^2 \tag{1}$$

$$\sigma'(\mathbf{a}_t) = \sigma(\mathbf{a}_t) \odot (1 - \sigma(\mathbf{a}_t)) = \mathbf{h}_t \odot (1 - \mathbf{h}_t) \tag{2}$$

$$\delta_t^y = \frac{\partial \mathcal{L}_t}{\partial \hat{\mathbf{y}}_t} = \hat{\mathbf{y}}_t - \mathbf{y}_t \tag{3}$$

$$\frac{\partial \mathcal{L}_t}{\partial \mathbf{W}_{hy}} = \delta_t^y \cdot \mathbf{h}_t^T \tag{4}$$

$$\frac{\partial \mathcal{L}_t}{\partial \mathbf{b}_y} = \delta_t^y \tag{5}$$

$$\delta_t^h = \mathbf{W}_{hy}^T \delta_t^y + \mathbf{W}_{hh}^T \delta_{t+1}^a \tag{6}$$

$$\delta_t^a = \delta_t^h \odot \sigma'(\mathbf{a}_t) \tag{7}$$

$$\frac{\partial \mathcal{L}_t}{\partial \mathbf{W}_{xh}} = \delta_t^a \cdot x_t^T \tag{8}$$

$$\frac{\partial \mathcal{L}_t}{\partial \mathbf{W}_{hh}} = \delta_t^a \cdot \mathbf{h}_{t-1}^T \tag{9}$$

$$\frac{\partial \mathcal{L}_t}{\partial \mathbf{b}_h} = \delta_t^a \tag{10}$$

We are now ready to implement this in code.

6.5 Code Implementation

Here is the corresponding Python code that implements the above equations:

```
def bptt(y_pred_t, y_t, h_t, W_hy, W_hh, delta_tplus1a,
         a_t, x_t, h_tminus1):
    delta_tplus1a = delta_tplus1a.reshape(-1, 1)
    delta_ty = (y_pred_t - y_t).reshape(-1, 1)
    h_t = h_t.reshape(-1, 1)
    h_tminus1 = h_tminus1.reshape(-1, 1)
    x_t = x_t.reshape(-1, 1)
    a_t = a_t.reshape(-1, 1)

    delta_th = W_hy.T @ delta_ty + W_hh.T @ delta_tplus1a
    delta_ta = delta_th * sigmoid_diff(a_t)

    dW_hy = delta_ty @ h_t.T
    dW_hh = delta_ta @ h_tminus1.T
    dW_xh = delta_ta @ x_t.T
    db_h = delta_ta
    db_y = delta_ty

    return dW_xh, dW_hh, dW_hy, db_h, db_y, delta_ta
```

This function will be called for each time step in reverse, and the gradients are accumulated across all t to get the final parameter updates.

What is this function doing? This function takes in as arguments the next value $\hat{\mathbf{y}}_t$, the true ground truth $\mathbf{y}_t$, the post-activation state $\mathbf{h}_t$, the current weight parameters (not the bias parameters! Have a look in the above equations, nowhere for any computation did we require any bias parameters), the error signal coming from the future pre-activation state δ^a_{t+1}, the current pre-activation state $\mathbf{a}_t$, the current input $\mathbf{x}_t$, and the previous post-activation state, $\mathbf{h}_{t-1}$. Where will we be getting these inputs from? In 4.3.1 we had saved all these input values in a particular list, apart from the error terms - these error terms will be gotten from the `rnn_backward()` function, which we will define next.

Note that we are explicitly reshaping all the input vectors to match their particular shapes - again, as mentioned previously, this is to avoid all the nasty errors that come along when we don't multiply matrices with correct dimensions correctly.

Let's now move on to the final piece of core code, the complete backward pass.

6.6 Wrapping it up: The Complete Backward Pass

Let's now focus on wrapping up the final piece of core code, the complete backward pass over all time.

```
def rnn_backward(x_list, a_list, h_list, y_pred_list, Y,
                 W_xh, W_hy, W_hh, b_h, b_y):
    T = len(Y) # The length of the time sequence

    dW_xh_total = np.zeros_like(W_xh)
    dW_hh_total = np.zeros_like(W_hh)
    dW_hy_total = np.zeros_like(W_hy)
    db_h_total  = np.zeros_like(b_h)
    db_y_total  = np.zeros_like(b_y)

    delta_tplus1a = np.zeros((W_hh.shape[0], 1))  # always shape (m, 1)

    for t in reversed(range(T)): # Since we are propogating backwards
        y_pred_t = y_pred_list[t]
        y_t = Y[t]
        h_t = h_list[t + 1]
        a_t = a_list[t]
        x_t = x_list[t]
        h_tminus1 = h_list[t]
        dW_xh, dW_hh, dW_hy, db_h,
        db_y, delta_ta = bptt(y_pred_t, y_t, h_t, W_hy, W_hh,
                              delta_tplus1a, a_t, x_t, h_tminus1)

        dW_xh_total += dW_xh
        dW_hh_total += dW_hh
        dW_hy_total += dW_hy
        db_h_total  += db_h
        db_y_total  += db_y

    # We are accumulating the gradients for all the losses

```

```
        delta_tplus1a = delta_ta

    return dW_xh_total, dW_hh_total, dW_hy_total, db_h_total, db_y_total
```

What are the arguments we pass to this function? Remember in 6.5 in `bptt()`, where we had to pass in the values, and we said we would get those values from our saved lists? We pass these same lists, that is, `x_list, a_list, h_list, y_pred_list` which we had created during our `rnn_forward()` in 4.3.1.

We then initialise our **total** gradient terms to zero, and we will continue to add the gradient terms for each time step as we traverse the time loop in *reverse*. We also initialise the initial pre-activation error signal δ_t^a to the zero vector $0 \in \mathbb{R}^m$, since as we start from the last time step, there is no error signal to be taken from further time steps because they do not exist.

As we traverse the time loop in reverse, we gather all the values we need for a particular backward pass at a particular time step from our lists (remember, again, that lists in Python are 0-indexed!) and pass all our collected values into the `bptt()` function, which gives us all of our gradient terms and error terms for a particular time step. We then accumulate all of our gradients, as explained in 6.1.1, and then finally set $\delta_{t+1}^a \leftarrow \delta_t^a$ to be used in the next time step $t-1$, since its job in the current time step at t is complete, and we continue until we reach the last time step, $t = 0$ ($t = 1$ mathematically), at which point we exit the loop.

Finally, we exit the function by returning the total accumulated values for all the gradients.

With BPTT complete, we are now equipped to train our RNN and move on to implementation and testing. Congratulations on making it this far!

7 Training the RNN

We now have all of our functions ready - it's time to create the training loop, which takes in as input all of our $\mathbf{x}_t$'s and tries to predict the according next value. We are providing it the next values in the sequence, that is, $\mathbf{y}_t$ - essentially, these are just our $\mathbf{x}_t$s but shifted forward one time step, and we will be predicting the values for the *next* time step. We will repeat the example given earlier.

Suppose that you are given your $\mathbf{x}_t$'s as $\sin(0.1), \sin(02), \ldots, \sin(T)$. Your $\mathbf{y}_t$'s will be the next predictions for this value - that is, $\sin(0.2), \sin(03), \ldots, \sin(T+1)$ after training. Or, after training, if you train it on a similar sine sequence from some other temporal period, the model should be able to figure out that it is following the sine wave and start predicting the next outputs. However, for reasons you will soon see, we will not actually use this sine sequence, but rather a toggle sequence, to train our simple RNN model. This toggle sequence will be given as:

$$\mathbf{x}_t = 0, 1, 0, 1, 0, 1, \ldots$$
$$\mathbf{y}_t = 1, 0, 1, 0, 1, 0, \ldots$$

We will attempt to see if our model can learn this simple sequence through time and output.

Let's first define our training loop:

```python
def train_rnn(X_seq, Y_seq, n, m, k, epochs=100, learning_rate=0.01):
    # Initialise weights and biases
    W_xh, W_hh, W_hy, b_h, b_y = initialise(n, m, k)
    h_0 = np.zeros((m, 1))  # column vector shape (m,)

    loss_history = [] # Store the loss history to plot later

    # Run the forward and backward passes per epoch
    for epoch in range(epochs):
        # Forward pass
        x_list, a_list,
        h_list, y_pred_list = rnn_forward(X_seq, h_0, W_xh,
                                          W_hh, W_hy, b_h, b_y)

        # Compute total loss
        total_loss = 0
        for t in range(len(Y_seq)):
            total_loss += loss_t(y_pred_list[t], Y_seq[t])

        # Backward pass (BPTT)
        dW_xh, dW_hh, dW_hy,
        db_h, db_y = rnn_backward(x_list, a_list, h_list,
                                  y_pred_list, Y_seq, W_xh, W_hy,
                                  W_hh, b_h, b_y)

        # Update weights
        W_xh -= learning_rate * dW_xh
        W_hh -= learning_rate * dW_hh
        W_hy -= learning_rate * dW_hy
        b_h  -= learning_rate * db_h
        b_y  -= learning_rate * db_y

        if (epoch + 1) % 10 == 0 or epoch == 0:
            print(f"Epoch {epoch + 1}, Loss: {total_loss:.4f}")
        loss_history.append(total_loss)

    return W_xh, W_hh, W_hy, b_h, b_y, loss_history
```

Let's quickly go through what this code is doing.

This code takes in your input data and the corresponding outputs, as `X_seq, Y_seq`. It also takes in the model parameters n, m, k, and the parameters that one sets for themselves for training, the number of epochs (passes over the training data) and the learning rate, η. We typically initialise the η to 0.01 in most cases.

We then initialise our model parameters through the `initialise()` function 4.2.1. Then, for each epoch, we run input data through `rnn_forward` 4.3.1 and then compute the total loss 6.1 after comparing each predicted value $\hat{\mathbf{y}}_t$ to its true value $\mathbf{y}_t$. This will allow us to keep a track of our loss function as we are training our model - ideally, as the number of epochs increase, our loss should tend to 0. We then back-propagate our errors and try to find out by how much we should vary our model parameters so that it fits the

training data better, and so we run `rnn_backward()` 6.6, and then update the weights. Finally, at convenient intervals, we print our loss and try to see whether our loss is going down or not, as expected. We finally return the parameters of our model to be used for testing purposes.

7.1 Training using the Toggle Sequence

Let us first generate a simple toggle sequence - this will be sufficient for training our RNN.

```
X_seq = [np.array([[0]]), np.array([[1]]), np.array([[0]]), np.array([[1]])]
Y_seq = [np.array([[1]]), np.array([[0]]), np.array([[1]]), np.array([[0]])]
```

And finally, let us train our RNN on this sequence.

```
W_xh, W_hh, W_hy,
b_h, b_y, loss_history = train_rnn(X_seq, Y_seq, n=1, m=5,
                                   learning_rate=0.1)
```

Assuming everything goes correctly, we should see something like this in the console:

```
Epoch 1, Loss: 1.5608
Epoch 10, Loss: 0.4905
Epoch 20, Loss: 0.4408
Epoch 30, Loss: 0.3826
Epoch 40, Loss: 0.3123
Epoch 50, Loss: 0.2340
Epoch 60, Loss: 0.1583
Epoch 70, Loss: 0.0966
Epoch 80, Loss: 0.0537
Epoch 90, Loss: 0.0279
Epoch 100, Loss: 0.0141
Epoch 110, Loss: 0.0072
Epoch 120, Loss: 0.0038
Epoch 130, Loss: 0.0022
Epoch 140, Loss: 0.0014
Epoch 150, Loss: 0.0009
Epoch 160, Loss: 0.0007
Epoch 170, Loss: 0.0005
Epoch 180, Loss: 0.0004
Epoch 190, Loss: 0.0003
Epoch 200, Loss: 0.0002
```

If you see something like above, congratulations! You've successfully trained your RNN. We can plot this loss over time to visualise easily how exactly our model is learning, for which we will need to import two packages: `matplotlob.pyplot, seaborn`, which are both data visualisation packages.

```
import seaborn as sns
import matplotlib.pyplot as plt
```

```

sns.lineplot(x=range(len(loss_history)), y=loss_history, label="Loss")

plt.title("RNN Training Loss Curve")
plt.xlabel("Epoch")
plt.ylabel("Total Loss")
plt.legend()
plt.show()
```

The above code gives us a curve as shown:

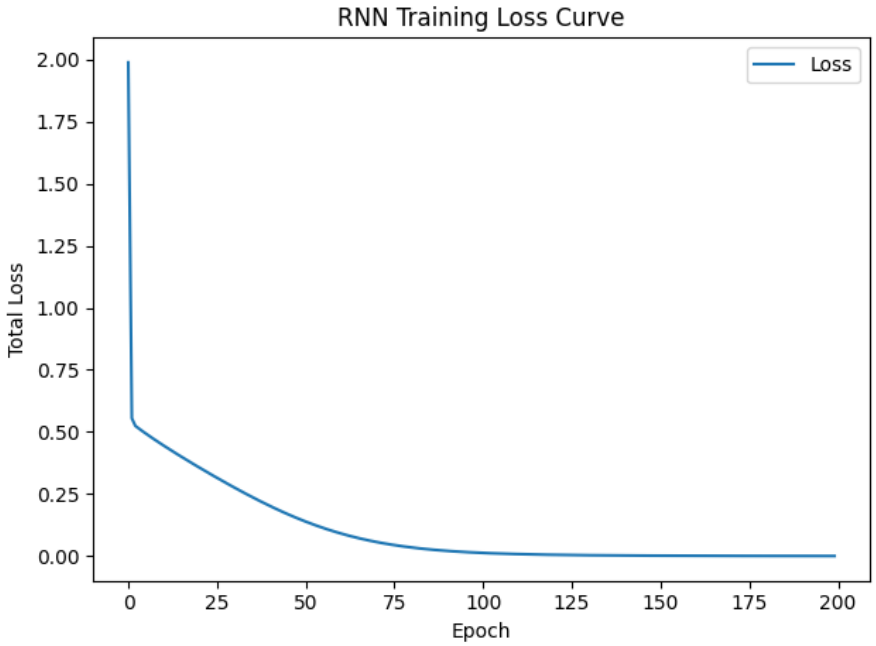

Figure 6: Training loss curve for the Elman RNN over epochs.

It is clear that the loss decreases sharply after the first epoch, showing that the model learns pretty well in the first epoch, and learns smoothly within the next few epochs until the loss reaches almost zero.

Let's test it on a new toggle sequence, this time, starting from 1 - let's try to test the model on the sequence $1, 0, 1, 0$. For this, we need to define a `predict_rnn()` function which takes in our sequence of $\mathbf{x}_t$'s and uses our estimated parameters.

```
def predict_rnn(X_seq, h_0, W_xh, W_hh, W_hy, b_h, b_y):
    h_t = h_0
    y_preds = []

    for x_t in X_seq:
        x_t = x_t.reshape(-1, 1)
        a_t = W_xh @ x_t + W_hh @ h_t + b_h
        h_t = sigmoid(a_t)
        y_pred_t = W_hy @ h_t + b_y
        y_preds.append(y_pred_t)
```

```
11
12      return y_preds
```

This function returns our list of predictions, $\hat{\mathbf{y}}_t$'s. It does this be keeping on re-defining $\mathbf{h}_t$ when needed, that is, at line 8 above, and that gets rolled over for the next iteration. Let's test this out on our trained model now and print out the list of predictions:

```
X_test=[np.array([[1]]), np.array([[0]]), np.array([[1]]), np.array([[0]])]
h_0_test = np.zeros((5, 1))  # same m as before

y_preds = predict_rnn(X_test, h_0_test, W_xh, W_hh, W_hy, b_h, b_y)

print("Predictions:")
for i, y in enumerate(y_preds):
    print(f"t={i}: {y.flatten()[0]:.4f}")
```

You may get an output as below:

```
Predictions:
t=0: 0.1414
t=1: 0.9364
t=2: 0.0160
t=3: 0.9601
```

Why is the first prediction that far off? Remember, our model was trained on a sequence starting from 0, that is, 1, 0, 1, 0, ...

Our test makes it start from 1. For each time the model had been trained to predict the next value as 0, that is, our input value has been $\mathbf{x}_t = 1$, it had some previous value $\mathbf{h}_t$ which was most likely *not* 0 being passed from the previous time iteration. However, when starting from 1 to predict 0, our model is passing the $\mathbf{h}_0$ as 0 to predict $\hat{\mathbf{y}}_t$, and that is causing the slight difference. After a while though, the model does learn to predict the sequence from the previous iterations and it passes the correct $\mathbf{h}_t$ (or somewhat correct) to predict the next value.

And with that, we finish with the training of our simple RNN Model.

7.2 Training using the Sinusoidal sequence

This section is optional, but will introduce the concepts of vanishing and exploding gradients, gradient clipping, and how we can model more complex periodic time sequences with our vanilla RNN.

Let's start by creating our training sequences:

```
X_seq = [np.array([[np.sin(t / 100)]]) for t in range (1000)]
Y_seq = [np.array([[np.sin((t + 1) / 100)]]) for t in range (1000)]
```

And now, let's run our `train_rnn()` on these sequences and see what we get.

```
W_xh, W_hh, W_hy,
b_h, b_y, loss_history = train_rnn(X_seq, Y_seq, n=1, m=5,
                                   k=1, epochs=200, learning_rate=0.1)
```

What's the output we get?

```
Epoch 1, Loss: 291.3229
/tmp/ipython-input-210-2947873235.py:8: RuntimeWarning: overflow encountered in exp
  return 1 / (1 + np.exp(-x))
Epoch 10, Loss: 873191392849790827235386626611955630080000
Epoch 20, Loss: 714189298061930997352254763519630621078849184180320713343855598557578
Epoch 30, Loss: 584140381642480358875934303288995460460013110556070607077187136109845
Epoch 40, Loss: 477772470676105502932908080965450202740961051947088572900810193834124
Epoch 50, Loss: 390773418358949832390216192617307937886837543200878462318239641457993
Epoch 60, Loss: 319616289904416898853532904385639957823497404937836255703666722189476
Epoch 70, Loss: 261416380881883563568494629296765621189480030833119189531881632070029
/tmp/ipython-input-214-1970746919.py:17: RuntimeWarning: overflow encountered in scal
  total_loss += loss_t(y_pred_list[t], Y_seq[t])
/tmp/ipython-input-211-3145719394.py:2: RuntimeWarning: overflow encountered in squar
  return np.sum((y_pred_t - y_t) ** 2) / 2
Epoch 80, Loss: inf
Epoch 90, Loss: inf
Epoch 100, Loss: inf
Epoch 110, Loss: inf
Epoch 120, Loss: inf
Epoch 130, Loss: inf
Epoch 140, Loss: inf
/tmp/ipython-input-211-3145719394.py:18: RuntimeWarning: overflow encountered in matm
  delta_th = W_hy.T @ delta_ty + W_hh.T @ delta_tplus1a
/tmp/ipython-input-211-3145719394.py:19: RuntimeWarning: invalid value encountered in
  delta_ta = delta_th * sigmoid_diff(a_t)
Epoch 150, Loss: nan
Epoch 160, Loss: nan
Epoch 170, Loss: nan
Epoch 180, Loss: nan
Epoch 190, Loss: nan
Epoch 200, Loss: nan
```

Well, this is obviously both bad and concerning. What went wrong?

7.2.1 The Problem: Vanishing and Exploding Gradients

So, what exactly went wrong? The answer lies in a common problem that plagues the training of Recurrent Neural Networks: the phenomenon of **vanishing and exploding gradients**.

During BPTT, gradients are passed backward through many time steps. If the magnitude of the gradients is consistently less than 1 (e.g., due to small weights or activation function derivatives), they can shrink exponentially and become so small that they effectively become zero — this is the **vanishing gradient problem**. On the other hand, if the gradients are consistently greater than 1, they can grow exponentially large, leading to **exploding gradients**.

This is precisely what we are seeing in the above experiment — the loss suddenly increases to extremely large values (overflow warnings), then eventually becomes ∞, and

finally becomes `nan`. At this point, the model cannot learn anything meaningful, because the gradients that we computed caused our model to overshoot so far that it could not come back to the optimal point, even if we put a small enough learning rate η.

Mathematically, let's go back to our δ_t^h 6.4. Recall that

$$\delta_t^h = \mathbf{W}_{hy}^T \delta_t^y + \mathbf{W}_{hh}^T \delta_{t+1}^a \text{ and}$$
$$\delta_t^a = \delta_t^h \odot \sigma'(\mathbf{a}_t)$$

This means that the error signal δ_t^a continually takes contributions from δ_{t+1}^a, or we can say that

$$\delta_t^a \sim \mathbf{W}_{hh}^T \delta_{t+1}^a$$

And the gradients *directly* depend on this error signal δ_t^a 6.4. What do you think will happen if we have as many data points as we do (1000) and we are backpropagating these errors 1000 times?

If you unroll this across many time steps — say, hundreds or thousands — we see that δ_t^a becomes a function of repeated matrix multiplications by $\mathbf{W}_{hh}^T$ and elementwise multiplications by the derivative of the activation $\sigma'(\mathbf{a}_t)$.

Now here's the critical point:

- If the eigenvalues of $\mathbf{W}_{hh}$ are **greater than 1**, then this repeated multiplication leads to **exploding gradients** — the norm of δ_t^a increases exponentially, causing overflows.
- If the eigenvalues of $\mathbf{W}_{hh}$ are **less than 1**, then the gradients **vanish** — the signal becomes exponentially smaller, and no useful gradient is left to update the earlier weights.

Additionally, the derivative of sigmoid and tanh activations lie in the range $(0, 0.25)$ and $(0, 1)$ respectively — both of which are less than 1. This further contributes to the gradient shrinking in magnitude as we backpropagate through time. Therefore, we have an unfortunate situation:

> **The gradients either shrink so much that they vanish, or explode so rapidly that training becomes unstable.**

This is what we call the **Vanishing and Exploding Gradients Problem**, and it is a fundamental challenge when training RNNs over long sequences.

How do we fix this? In the next section, we will look at one of the most commonly used practical techniques: *Gradient Clipping.*

7.2.2 The Quick and Dirty Solution: Gradient Clipping

Gradient clipping is a simple yet effective trick to stabilise training when the gradients start to explode.

The idea is straightforward: *if the norm of the total gradient vector exceeds a certain threshold, we scale it down to lie within that threshold, without changing its direction.*

Let us suppose that we have computed all the gradients across time, and they are stored in parameters like:

$$\mathbf{g} = \left\{ \frac{\partial \mathcal{L}}{\partial \mathbf{W}_{xh}}, \frac{\partial \mathcal{L}}{\partial \mathbf{W}_{hh}}, \frac{\partial \mathcal{L}}{\partial \mathbf{W}_{hy}}, \frac{\partial \mathcal{L}}{\partial \mathbf{b}_h}, \frac{\partial \mathcal{L}}{\partial \mathbf{b}_y} \right\}$$

We now compute the global norm (global since we want to include *all* parameters and give them the same weightage:

$$N = \sqrt{\sum_i \|\mathbf{g}_i\|^2}$$

and check whether this exceeds a certain threshold τ (usually somewhere between 1 and 5). If it does, we scale every parameter gradient proportionally:

$$\mathbf{g}_i \leftarrow \mathbf{g}_i \cdot \frac{\tau}{N} \quad \text{(only if } N > \tau\text{)}$$

This preserves the direction of the gradient (and hence the learning dynamics), but prevents it from growing arbitrarily large and destabilising training. Essentially, this allows us to continue learning.

Let us implement this in Python. We can insert this function anywhere before the `train_rnn()` function.

```
def clip_gradients(gradients, threshold):
    total_norm = 0
    for grad in gradients:
        total_norm += np.sum(grad ** 2)
    total_norm = np.sqrt(total_norm)

    if total_norm > threshold:
        for i in range(len(gradients)):
            gradients[i] = gradients[i] * (threshold / total_norm)
    return gradients
```

And inside the training loop, just before updating our weights, we can call this function:

```
gradients = [dW_xh, dW_hh, dW_hy, db_h, db_y]
dW_xh, dW_hh, dW_hy, db_h, db_y = clip_gradients(gradients, threshold=5.0)
```

This one change is enough to prevent exploding gradients in practice, and is widely used in almost every deep learning framework under the hood when training RNNs or other recurrent architectures.

In future papers, we will explore more principled alternatives — like using more expressive RNN variants, like the LSTM — but this fix is remarkably effective in most vanilla RNN implementations.

Let's try training our model once more.

```
X_seq = [np.array([[np.sin(t / 100)]]) for t in range (1000)]
Y_seq = [np.array([[np.sin((t + 1) / 100)]]) for t in range (1000)]

W_xh, W_hh, W_hy,
b_h, b_y, loss_history = train_rnn(X_seq, Y_seq, n=1, m=5, k=1,
```

We have increased the number of outputs to allow the model to sufficiently learn in this case, since we have capped the gradients to a maximum threshold, and large gradients which are actually beneficial to learning are capped.

We will not show the output here (you should observe that your loss begins to approach 0 towards the end).

If you attempt to plot the loss over the epochs as in 7.1, you should get a figure like so:

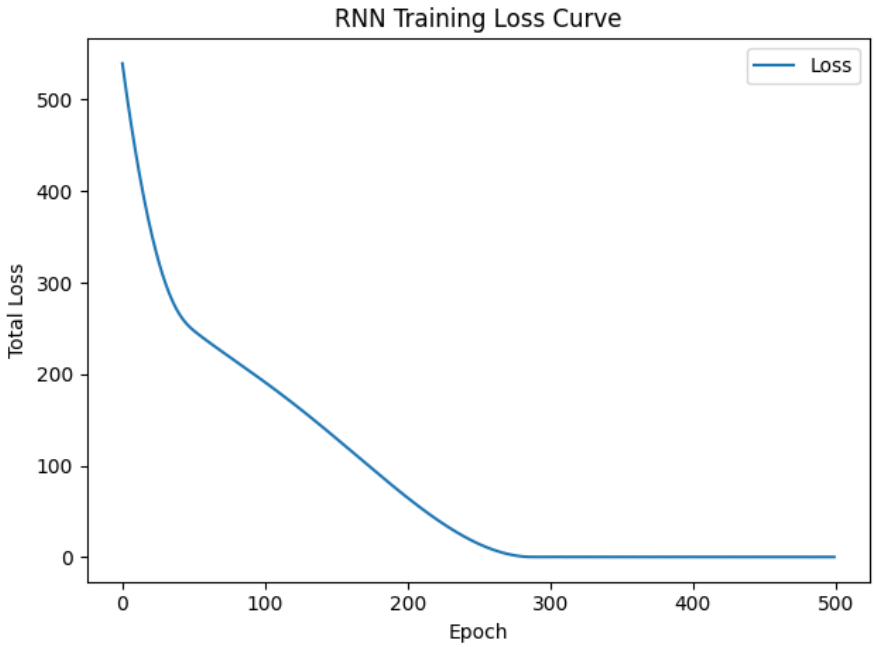

Figure 7: Training loss curve for the Elman RNN over epochs for Sinusoidal Training.

Let's try to test our model now! We generate the test sequence (you may generate it over however many examples you like - in my case, I decided to do it over 2000 steps of 0.01 radians each):

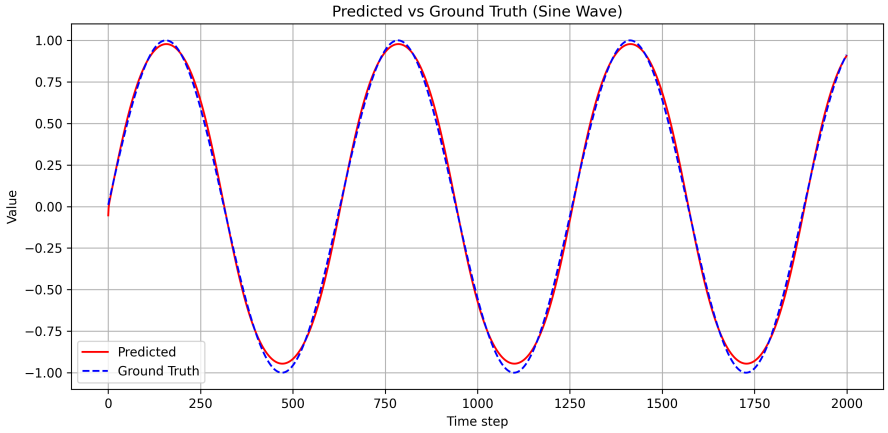

Figure 8: Predicted vs Ground Truth values for sinusoidal training

This seems absolutely phenomenal - a true testament of the power of the ability of this RNN to learn and generalise.

The code for this, if you are interested, is as given below:

```
X_test = [np.array([[np.sin(t / 100)]]) for t in range(2000)]
Y_test = [np.array([[np.sin((t + 1) / 100)]]) for t in range(2000)]

h_0_test = np.zeros((5, 1))  # assuming hidden size m = 5
y_preds = predict_rnn(X_test, h_0_test, W_xh, W_hh, W_hy, b_h, b_y)

preds = [y.flatten()[0] for y in y_preds]
ground_truth = [y.flatten()[0] for y in Y_test]

plt.figure(figsize=(10, 5))
plt.plot(preds, label='Predicted', color='r')
plt.plot(ground_truth, label='Ground Truth', color='b', linestyle='dashed')
plt.title('Predicted vs Ground Truth (Sine Wave)')
plt.xlabel('Time step')
plt.ylabel('Value')
plt.legend()
plt.show()
```

You may also notice initially that the value seems a little off, at around $t = 0$ - this is due to the same reason mentioned in 7.1 towards the end, of not having a previous, correct $\mathbf{h}_0$ value for $\mathbf{h}_1$.

8 A Quick Probabilistic Perspective to RNNs and ML in General

So far, we've defined the loss function for our RNN as the squared error:

$$\mathcal{L}_t = \frac{1}{2}\|\hat{y}_t - y_t\|_2^2$$

and we've used it for training without questioning where it comes from. In this section, we build a simple probabilistic foundation to justify this loss function more formally — and thereby show how training an RNN can be seen as a process of *maximum likelihood estimation*.

8.1 A Quick Primer on Maximum Likelihood Estimation (MLE)

Suppose you have a random variable Y whose probability distribution depends on some set of parameters θ, and you observe data $y_1, y_2, \ldots, y_T$ drawn from it. The idea of **Maximum Likelihood Estimation (MLE)** is to choose the value of θ that makes the observed data most probable.

Mathematically, if the observations are independent and identically distributed (i.i.d.), the likelihood of the dataset is:

$$P(y_1, y_2, \ldots, y_T \mid \theta) = \prod_{t=1}^{T} P(y_t \mid \theta)$$

We then choose θ to maximise this likelihood:

$$\hat{\theta}_{\text{MLE}} = \arg\max_{\theta} \prod_{t=1}^{T} P(y_t \mid \theta)$$

This essentially means that we want to choose the value of θ as the estimator $\hat{\theta}$ that maximises the value of the expression on the right.

For convenience, we typically maximise the **log-likelihood**, since the argument of θ that maximises the both will be the same:

$$\log P(y_1, y_2, \ldots, y_T \mid \theta) = \sum_{t=1}^{T} \log P(y_t \mid \theta)$$

8.2 Connecting this to Neural Networks

In neural networks — and in our RNN — we model the output $\hat{y}_t$ as a function of the input and parameters (i.e., weights and biases). If we now assume that the *errors* of our model follow a Gaussian (normal) distribution, then we can write:

$$y_t = \hat{y}_t + \epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0, \sigma^2)$$

That is, the true output y_t is the predicted value plus some zero-mean Gaussian noise.

Thus, the likelihood of observing y_t given $\hat{y}_t$ is:

$$P(y_t \mid \hat{y}_t) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y_t - \hat{y}_t)^2\right)$$

Taking the log-likelihood:

$$\log P(y_t \mid \hat{y}_t) = -\frac{1}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}(y_t - \hat{y}_t)^2$$

Now summing over all time steps $t = 1, \ldots, T$, we get:

$$\log P(Y \mid \hat{Y}) = \sum_{t=1}^{T} \log P(y_t \mid \hat{y}_t) = \text{const} - \frac{1}{2\sigma^2} \sum_{t=1}^{T} (y_t - \hat{y}_t)^2$$

Since σ^2 is a constant, **maximising the log-likelihood is the same as minimising the squared error!**

Hence, the total loss:

$$\mathcal{L} = \sum_{t=1}^{T} \frac{1}{2}(y_t - \hat{y}_t)^2$$

can be viewed as arising from maximum likelihood estimation under the assumption that the model errors are Gaussian-distributed and i.i.d.

8.3 Clarifying a Common Confusion: Are the Targets Gaussian?

It is important to note that we are *not* assuming that the targets y_t are drawn from a Gaussian distribution directly. Rather, we are assuming that *given the model's prediction* $\hat{y}_t$, the **error** (or noise) in prediction is Gaussian. That is, our model is deterministic, but we imagine a probabilistic noise model on top of it.

This subtle distinction is critical: the targets themselves may be structured and sequential (as in our sinusoidal RNN), and are certainly not i.i.d. — but the error residuals are assumed to be i.i.d. Gaussian to justify our loss function mathematically.

8.4 Conclusion

This perspective allows us to justify squared loss from first principles. Under the assumption of Gaussian errors, training an RNN to minimise squared loss is equivalent to performing maximum likelihood estimation. This bridges our deterministic model and the probabilistic viewpoint. Most ML models can also be modelled this way, as a probabilistic model - you will see this if you ever study Gaussian Discriminant Analysis and Naive Bayes.

If you didn't understand this yet, not to worry - this just shows that the choice of our loss function is not *arbitrary*, and that there is mathematical reasoning behind everything.

9 Conclusion and Future Work

In this paper, we have attempted to build a complete understanding of Recurrent Neural Networks (RNNs) from first principles — beginning from their forward propagation mechanics, to the intricacies of Backpropagation Through Time (BPTT), the matrix calculus that underlies gradient computation, and finally culminating in a working, trainable implementation from scratch. We explored the intuition and derivation behind every mathematical expression, verified the dimensions and sanity of our gradients, and applied our implementation to tasks such as predicting toggle sequences and modelling sinusoidal data.

We also provided a probabilistic interpretation of the squared loss function via maximum likelihood estimation, thereby bridging the gap between deterministic and probabilistic views of training neural networks. This helped us understand why squared error loss is commonly used in regression and how it arises naturally from basic probabilistic assumptions.

Additionally, we encountered and addressed one of the most important challenges of training RNNs — the issue of vanishing and exploding gradients — and discussed how gradient clipping can be used as a simple yet effective strategy to stabilise training.

9.1 Where Do We Go From Here?

While our vanilla RNN implementation is a powerful educational tool, it suffers from inherent limitations — most notably the difficulty in learning long-term dependencies due to gradient instability. This motivates the development of more robust architectures such as:

- **Long Short-Term Memory (LSTM)** Networks: Introduce gating mechanisms that allow the model to explicitly learn what information to retain, forget, or pass forward, enabling stable training over long sequences.
- **Gated Recurrent Units (GRU)**: A simplified version of LSTMs that achieves comparable performance with fewer parameters.
- **Bidirectional RNNs**: Process sequences both forward and backward to better capture context, especially in tasks like sequence labelling or translation.
- **Sequence-to-Sequence Models and Attention**: Extend RNNs to model full sequence mappings (e.g., machine translation), and introduce attention mechanisms to dynamically focus on relevant time steps.

In future work, we aim to build an LSTM from scratch — mirroring the level of depth and intuition presented in this paper — and demonstrate its superiority on long-range sequence modelling tasks. This will involve not just implementation, but also a detailed derivation of the LSTM update equations and their training dynamics.

9.2 Final Thoughts

Understanding RNNs from scratch is not just an academic exercise. It demystifies the black-box nature of deep learning, sharpens mathematical intuition, and gives one the confidence to modify or build custom architectures for new problems. As the field moves toward Transformers and other paradigms, mastering the foundations remains essential — and that's what we have attempted here.

References

[1] J. L. Elman, *Finding Structure in Time*, Cognitive Science, 1990.

[2] Ian Goodfellow, Yoshua Bengio, Aaron Courville, *Deep Learning*, MIT Press, 2016.

[3] Xavier Glorot and Yoshua Bengio. *Understanding the difficulty of training deep feedforward neural networks.* In Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS), 2010.

[4] Bengio, Y., Simard, P., and Frasconi, P. *Learning long-term dependencies with gradient descent is difficult.* IEEE Transactions on Neural Networks, 1994.

[5] Kaare Brandt Petersen and Michael Syskind Pedersen, *The Matrix Cookbook*, Version November 15, 2012. Available at: `https://www.math.uwaterloo.ca/~hwolkowi/matrixcookbook.pdf`

[6] Ralf C. Staudemeyer and Eric Rothstein Morris, *Understanding LSTM — A Tutorial into Long Short-Term Memory Recurrent Neural Networks*, Schmalkalden University of Applied Sciences, Germany, and Singapore University of Technology and Design, Singapore, September 23, 2019.

[7] S. Hochreiter and J. Schmidhuber, *Long Short-Term Memory*, Neural Computation, vol. 9, no. 8, pp. 1735–1780, 1997.

[8] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton, *On the importance of initialization and momentum in deep learning*, In Proceedings of the 30th International Conference on Machine Learning (ICML), 2013.