RNNS: FORMULATION AND APPLICATION IN NLP

Assignment 5, CS-587

Author
Nikolaos Barmparousis
csd4690
June 7, 2024

Contents

1 Introduction

Recurrent Neural Networks (RNNs) are essential for sequence modeling tasks, allowing them to maintain information across time steps, which is crucial for applications like text generation and language modeling. This assignment focuses on understanding and implementing RNNs using the Numpy framework.

The assignment is structured into three parts:

- **Part A: Overview of RNNs** - This part delves into the core concepts of RNNs, including their structure, forward propagation, hidden state management, different RNN configurations based on input and output sequences, deep RNNs, and modern RNN architectures like LSTMs and GRUs. It provides a comprehensive understanding of RNNs and their various applications.
- **Part B: Backpropagation through Time (BPTT)** - This part focuses on the mathematical derivation of gradients for the backward pass of RNNs. It also explains the reasoning behind the main drawbacks present in vanilla RNNs, such as exploding/vanishing gradients and difficulty in capturing long-term dependencies.
- **Part C: Implementation and Application** - This part involves implementing and running an RNN model on two datasets for character-level text generation. The datasets include names of chemical elements and movie titles. This section covers the setup, results, and discussion of the generated outputs.

This structure ensures a thorough exploration of RNNs from theoretical foundations to practical applications, providing a deep understanding of their functionality, limitations, and potential solutions.

2 Part A: Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are a specialized type of neural network designed to handle sequential data. Unlike traditional neural networks, RNNs maintain a hidden state that captures information from previous time steps, allowing them to model temporal dependencies. This capability makes RNNs particularly suitable for tasks such as language modeling, text generation, and time series prediction.

2.1 RNN Fundamentals and Mechanics

2.1.1 The Mechanics of an RNN Cell

At the core of an RNN is the RNN cell, which processes input sequences one element at a time while maintaining an evolving hidden state. Figure 1 illustrates the structure of an RNN cell and its forward propagation steps.

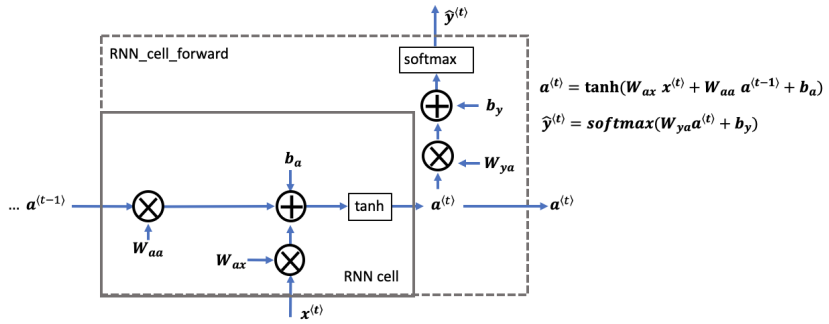


Figure 1: RNN Cell Forward Propagation

In the figure, the RNN cell receives an input $x^{(t)}$ at time step t and the hidden state from the previous time step $a^{(t-1)}$. The cell computes the current hidden state $a^{(t)}$ and the output prediction $\hat{y}^{(t)}$ using the following equations:

$$a^{(t)} = \tanh(W_{ax}x^{(t)} + W_{aa}a^{(t-1)} + b_a)$$
$$\hat{y}^{(t)} = \text{softmax}(W_{ya}a^{(t)} + b_y)$$

- **Hidden State Calculation**: The hidden state $a^{(t)}$ is computed using the tanh activation function. This state is a combination of the input $x^{(t)}$ and the previous hidden state $a^{(t-1)}$, weighted by W_{ax} and W_{aa} respectively, and shifted by the bias b_a.
- **Output Prediction**: The output $\hat{y}^{(t)}$ is generated using the softmax function, which converts the hidden state $a^{(t)}$ into a probability distribution over possible outputs. This involves weights W_{ya} and bias b_y.

2.1.2 Unrolling RNNs Across Time

When processing a sequence of inputs, the RNN cell is unrolled across the entire sequence. This means that the cell's computations are repeated for each time step, passing the hidden

state from one step to the next. The unrolling process can be visualized as a chain of RNN cells, each contributing to the final output.

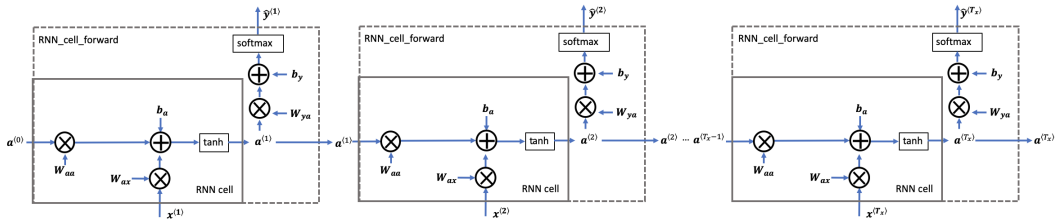


Figure 2: Unrolling RNN Cells Across Time Steps

Here, T represents the length of the input sequence. At each step, the RNN cell takes the input and the previous hidden state, computes the new hidden state, and produces an output. The initial hidden state a_0, is usually set to 0 or as a variable parameter that the model learns during the training process.

The same cell is used at each time step, meaning that the computation of each new state and output relies on the shared weight matrices W_{ax}, W_{aa}, W_{ya} and shared biases b_x, b_y. This parameter sharing is crucial for several reasons. It significantly reduces the computational resources required. More importantly, it allows the model to generalize across sequences of different lengths, as the same rules are applied consistently throughout the sequence. This reflects the fact that similar tasks are performed at each step, enabling the model to learn consistent patterns, regardless of the specific position within the sequence.

2.1.3 Forward Pass and Loss Computation

As we have mentioned, at each time step t, the inputs x_t and a_{t-1} are used to calculate the output of the model at time t, alongside the hidden state a_t that will be passed on to the next time step. The output of the model, denoted as $\hat{y}_t$, is passed through a loss function with the ground truth label y_t: $l(\hat{y}_t, y_t)$. After unrolling the RNN cell for each time step $1, 2, \ldots, T$, the loss function of the forward pass is computed as the sum of each time step's loss value:

$$L = \sum_{t=1}^{T} l(\hat{y}_t, y_t)$$

Figure 3 illustrates the forward pass of an RNN cell unrolled for an input of three time steps.

Starting from the bottom row, x_1 is passed through the RNN alongside h_0 (in our notation a_0) through the W_{hx} (W_{ax}) and W_{hh} (W_{aa}) weight matrices to produce h_1 (a_1). That h_1 (a_1) in the same row is passed through the W_{qh} (W_{ya}) weight matrix to produce o_1 ($\hat{y}_1$). The bias terms are omitted for simplicity. The h_1 (a_1) is also passed as input to the second row (the second time step) alongside x_2, and so on. The predictions of the model for each time step are passed through the l loss function in individual pairs, and the sum of all of them is the total loss function L.

[1]Figure taken from, https://www.d2l.ai/chapter_recurrent-neural-networks/bptt.html#backpropagation-through-time-in-detail.

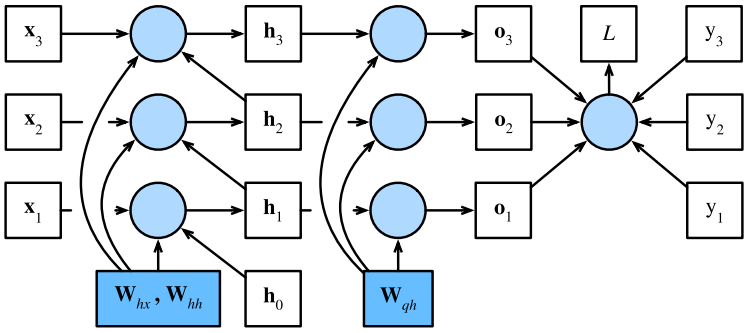


Figure 3: Unrolling RNN Cells Across Time Steps [1]

2.2 Types of RNNs based on Input/Output

So far, we have discussed RNNs that take a sequence of inputs and produce a synced sequence of outputs, such as in video classification where each frame has a corresponding label that the model predicts. However, there are several other types of RNNs that process different input/output configurations.

RNN models can be categorized based on their input and output sequences:

- **One to One**: This is the vanilla mode of processing without an RNN, where a fixed-sized input produces a fixed-sized output (e.g., image classification).
- **One to Many**: A single input leads to a sequence of outputs (e.g., image captioning, where an image is described with a sequence of words).
- **Many to One**: A sequence of inputs results in a single output (e.g., sentiment analysis, where a sentence is classified as expressing positive or negative sentiment).
- **Many to Many**: A sequence of inputs is transformed into a sequence of outputs (e.g., machine translation, where an RNN reads a sentence in one language and outputs a sentence in another language).
- **Synced Many to Many**: Each input in a sequence corresponds to an output (e.g., video classification, where each frame of the video is labeled).

Figure 4 illustrates these different types of RNN models and how the input and output sequences are processed.

In each case, there are no pre-specified constraints on the lengths of the sequences because the recurrent transformation (green) is fixed and can be applied as many times as needed. This flexibility allows RNNs to handle a wide variety of sequence-based tasks effectively.

2.3 Deep RNNs

So far, we have focused on RNNs with a sequence input, a single hidden RNN layer, and an output layer. These networks can influence outputs many time steps away, making them deep in a temporal sense.

[2]Figure taken from, https://karpathy.github.io/2015/05/21/rnn-effectiveness/

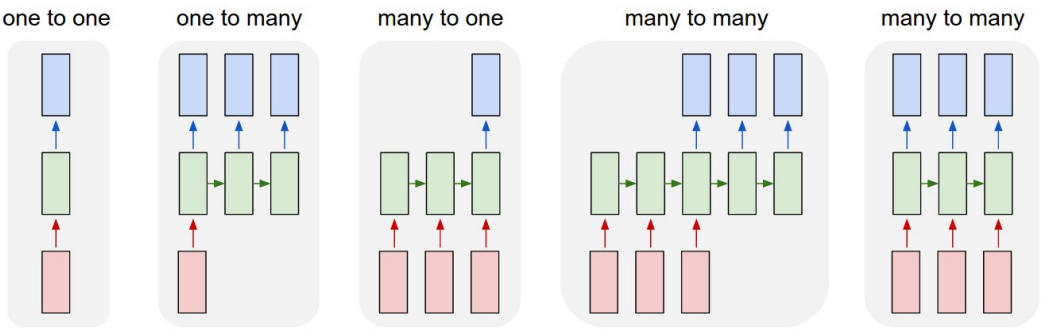


Figure 4: Different types of RNN models based on input and output sequences. Input vectors are in red, output vectors are in blue, and green vectors hold the RNN's state. [2]

To express complex relationships between inputs and outputs at the same time step, we construct RNNs that are deep in both the time direction and the input-to-output direction. This is similar to the depth in MLPs and CNNs.

The standard method for building deep RNNs is to stack RNN layers on top of each other. The first RNN layer processes the input sequence and produces an output sequence, which serves as the input to the next RNN layer. This stacking allows for more complex representations and deeper learning. Figure 5 illustrates a deep many-to-many RNN with multiple hidden layers.

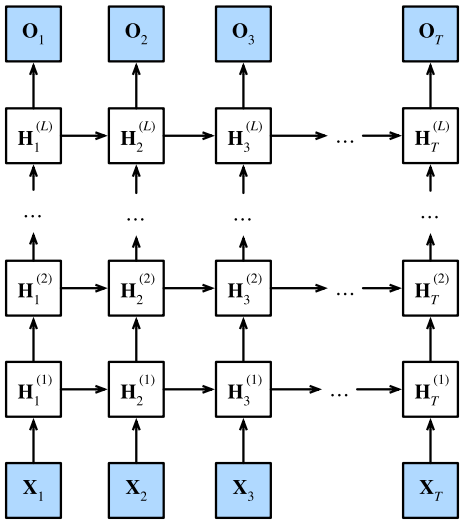


Figure 5: Illustration of a Deep many-to-many RNN with multiple hidden layers. Each hidden state operates on a sequential input and produces a sequential output, influenced by the same layer's value at the previous time step and the previous layer's value at the same time step. [3]

[3] Figure taken from, https://www.d2l.ai/chapter_recurrent-modern/deep-rnn.html#deep-recurrent-neural-networks

2.4 Modern RNN Architectures

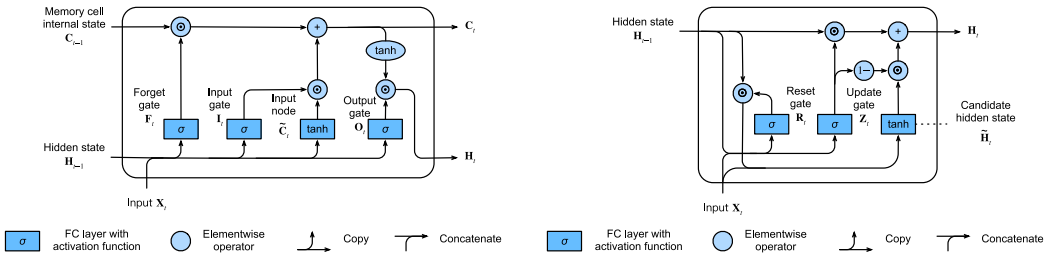
Vanilla RNNs face significant challenges, particularly when it comes to retaining long-range dependencies. This limitation arises because the information must be passed through many steps, causing it to diminish over time. For the same reason, vanilla RNNs suffer from the gradient vanishing/exploding problem during training, which can make it difficult for the network to learn effectively. We will better understand this problem when we define the gradient update terms for the backward pass.

To address these issues, more sophisticated models like Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs) were developed. These models incorporate gating mechanisms that regulate the flow of information, helping to retain long-term dependencies and mitigate the gradient vanishing/exploding problems.

LSTMs introduce three types of gates: the input gate, forget gate, and output gate. These gates control the cell state, allowing the network to remember or forget information as needed. This gating mechanism helps LSTMs to manage long-term dependencies more effectively than vanilla RNNs.

GRUs simplify the architecture by combining the input and forget gates into a single update gate, and they have an additional reset gate. This simplified gating mechanism makes GRUs computationally more efficient than LSTMs while still addressing the same fundamental issues.

Figure 6 illustrates the architectures of LSTMs and GRUs.



(a) LSTM Architecture

(b) GRU Architecture

Figure 6: Architectures of modern RNN models: LSTMs (left) and GRUs (right).

2.5 Theoretical Questions

A. Explain briefly the general functionality of RNN cells (how do we achieve memorization, the purpose of each function on the memorization task).

RNN cells process input sequences one element at a time while maintaining a hidden state. Memorization is achieved by passing the hidden state forward, allowing the network to retain information over time. As discussed in Section 2.1.1, the hidden state $a^{(t)}$ is computed using previous hidden states $a^{(t-1)}$ and current input $x^{(t)}$. The tanh activation function bounds the hidden state values to avoid exploding values and to introduce non-linearity, while the softmax function converts the hidden state into a probability distribution for output prediction.

B. List the limitations of RNNs, and briefly mention why these issues arise.

RNNs face several limitations:

- **Vanishing Gradients**: Gradients become very small during backpropagation, making it difficult to learn long-term dependencies.
- **Exploding Gradients**: Gradients grow excessively, causing numerical instability.
- **Sequential Computation**: Training time is slower due to the inability to parallelize computations effectively.

These issues arise due to the nature of gradient propagation over many time steps, as discussed in Section 2.4.

C. Consider the two following problems, (a) Recognizing images between 10 different species of birds, and (b) Recognizing whether a movie review says that the movie is worth watching or not. For which of the two problems can we use an RNN-based model? Justify your answer.

RNNs are suitable for recognizing whether a movie review says the movie is worth watching (NLP task). RNNs are designed to handle sequence data and context, making them suitable for analyzing the sentiment of a review. As mentioned in Section 2.2, the many-to-one model is appropriate for sentiment analysis tasks.

D. While training an RNN, you observe an increasing loss trend. Upon looking, you find out that at some point the weight values increase suddenly very much and finally take the value NaN. What kind of learning problem does this indicate? What can be a solution?

This indicates an exploding gradient problem. To solve this, apply gradient clipping to limit the values of gradients. Another solution that handles the vanishing gradients problem as well, is switching to more modern RNN architectures, as mentioned in Section 2.4.

E. Gated Recurrent Units (GRUs) have been proposed as a solution to a specific problem of RNNs. What is the problem they solve? And how?

GRUs address the vanishing gradient problem and difficulty in learning long-term dependencies. They use gating mechanisms to control the flow of information:

- **Reset Gate**: Determines how much of the past information to forget.
- **Update Gate**: Decides the amount of new information to be added to the hidden state.

Compared to LSTMs, GRUs have a simplified architecture with fewer gates, making them computationally more efficient. Details can be found in Section 2.4 and Figure 6.

3 Part B: Backpropagation through Time (BPTT)

Backpropagation through Time (BPTT) is a fundamental algorithm for training Recurrent Neural Networks (RNNs). It extends the backpropagation algorithm to handle the sequential nature of RNNs by computing gradients of the loss function, L, with respect to the learnable parameters W_{aa}, W_{ax}, W_{ya}, b_a, and b_y.

In this section, we will first derive these gradients for a single RNN cell and then extend our reasoning to the entire computational graph of a many-to-many RNN sequence architecture to better understand how the whole end-to-end training procedure is executed.

3.1 BPTT for a single RNN cell

The computational graph for a single RNN cell is illustrated in Figure 7.

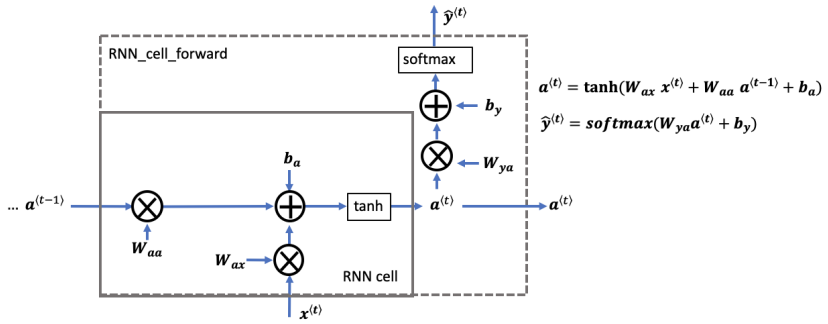


Figure 7: Computational graph for a single RNN cell.

The forward pass equations for the RNN are:

$$a_t = \tanh(W_{ax} \cdot x_t + W_{aa} \cdot a_{t-1} + b_a) \tag{1}$$

$$\hat{y}_t = \text{softmax}(W_{ya} \cdot a_t + b_y) \tag{2}$$

The functions are decomposed to make the backpropagation chains easier to understand:

$$a_t = \tanh(z_t) \tag{3}$$

$$z_t = W_{ax} \cdot x_t + W_{aa} \cdot a_{t-1} + b_a \tag{4}$$

$$y_t = \text{softmax}(o_t) \tag{5}$$

$$o_t = W_{ya} \cdot a_t + b_y \tag{6}$$

To compute the gradients for the single cell, the gradient $\frac{\partial \hat{y}_t}{\partial o_t}$ can be analytically computed. However, when paired with the Cross Entropy loss function, we can directly calculate $\frac{\partial L}{\partial o_t}$ in one step using the formula:

$$\frac{\partial L_i}{\partial o_j} = \text{softmax}(o_j) - y_j$$

where y is the one-hot encoded ground truth label, and o represents the logits.

Using the chain rule, we can calculate the gradient of $\hat{y}_t$ with respect to W_{ya}:

$$\frac{\partial \hat{y}_t}{\partial W_{ya}} = \left(\frac{\partial \hat{y}_t}{\partial o_t}\right) \cdot \left(\frac{\partial o_t}{\partial W_{ya}}\right) = \left(\frac{\partial \hat{y}_t}{\partial o_t}\right) \cdot a_t^T \quad \square \tag{7}$$

The bias term b_y is computed analogously:

$$\frac{\partial \hat{y}_t}{\partial b_y} = \frac{\partial \hat{y}_t}{\partial o_t} \cdot \frac{\partial o_t}{\partial b_y} = \frac{\partial \hat{y}_t}{\partial o_t} \quad \square \tag{8}$$

If this is the only cell in the unrolled BPTT, the $\nabla_{a_t} \hat{y}_t$ is calculated as:

$$\frac{\partial \hat{y}_t}{\partial a_t} = \frac{\partial \hat{y}_t}{\partial o_t} \cdot \frac{\partial o_t}{\partial a_t} = \left(\frac{\partial \hat{y}_t}{\partial o_t}\right) \cdot W_{ya} \quad \square \tag{9}$$

After that, we have to traverse through the tanh function to calculate $\frac{\partial \hat{y}_t}{\partial z_t}$. This is the upstream gradient we just calculated $\frac{\partial \hat{y}_t}{\partial a_t}$ multiplied by $\frac{\partial a_t}{\partial z_t}$.

For the tanh non-linearity backpropagation, use the following formula:

$$\frac{\partial a_t}{\partial z_t} = 1 - a_t^2 \tag{10}$$

Combining these, we get:

$$\frac{\partial \hat{y}_t}{\partial z_t} = \left(\frac{\partial \hat{y}_t}{\partial a_t}\right) \cdot \left(1 - a_t^2\right) \quad \square \tag{11}$$

Using the gradient we just computed, $\nabla_{a_t} \hat{y}_t$, we can calculate the derivative of $\hat{y}_t$ w.r.t W_{aa}, W_{ax}, b_a.

These are derived as follows:

$$\frac{\partial \hat{y}_t}{\partial W_{aa}} = \frac{\partial \hat{y}_t}{\partial z_t} \cdot \frac{\partial z_t}{\partial W_{aa}} = \left(\frac{\partial \hat{y}_t}{\partial z_t}\right) \cdot a_{t-1}^T \quad \square \tag{12}$$

$$\frac{\partial \hat{y}_t}{\partial W_{ax}} = \frac{\partial \hat{y}_t}{\partial z_t} \cdot \frac{\partial z_t}{\partial W_{ax}} = \left(\frac{\partial \hat{y}_t}{\partial z_t}\right) \cdot x_t^T \quad \square \tag{13}$$

$$\frac{\partial \hat{y}_t}{\partial b_a} = \frac{\partial \hat{y}_t}{\partial z_t} \cdot \frac{\partial z_t}{\partial b_a} = \left(\frac{\partial \hat{y}_t}{\partial z_t}\right) \quad \square \tag{14}$$

For the last timestamp in the unrolled RNN, its state, a_t, is only routed to its output, $\hat{y}_t$. However, this is not the case for all the previous timestamps in the forward pass (a_{t-1}, a_{t-2}, etc.). For the previous timestamp a_{t-1}, the state will consider two routes: one directly leading to its output, $\hat{y}_{t-1}$, and one indirectly leading to the next timestamp's output, $\hat{y}_t$, as a_{t-1} is used to compute a_t which will produce $\hat{y}_t$.

Thus, during the backwards pass, the gradient of the previous timestamp, a_{t-1}, will be influenced by these 2 gradients, one from its output,$\nabla_{a_{t-1}} \hat{y}_{t-1}$, and one from the next

state's gradient, $\nabla_{a_{t-1}} a_t$.

We will expand on this in the next subsection, where we will extend our calculations to the entire computational graph and compute the BPTT from the total loss function L with respect to the parameters.

3.2 BPTT for the entire RNN

In this subsection, we will expand on the gradients computed for a single cell and calculate the gradients of the total loss function L with respect to the parameters for all timestamps. We will use a synced many-to-many RNN of depth 1 for an unrolled input of 3 timestamps, as illustrated in Figure 8.

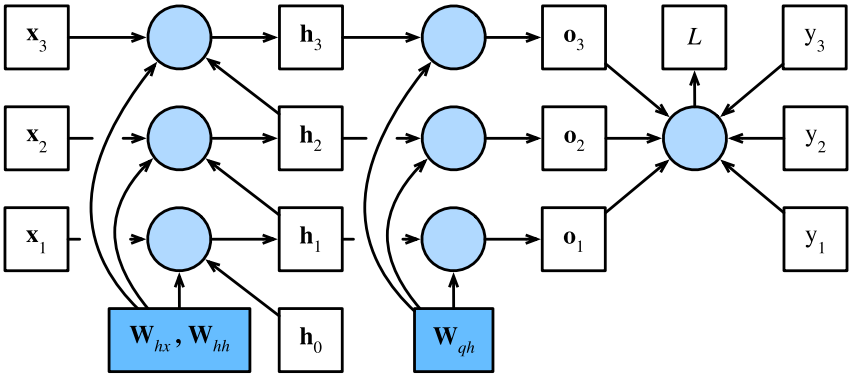


Figure 8: Computational graph for a 3-timestamp many-to-many RNN. Note that we have omitted the tanh and the softmax functions for ease of readability.

Our loss function is defined as:

$$L = \sum_{t=1}^{T} l(\hat{y}_t, y_t) \tag{15}$$

To extend the gradients we computed in the previous subsection to the total loss function, L, we denote $\nabla_{\hat{y}_t} L$ using the chain rule:

$$\frac{\partial L}{\partial \hat{y}_t} = \frac{\partial L}{\partial l} \cdot \frac{\partial l}{\partial \hat{y}_t} \tag{16}$$

To take this one step further, we define $\nabla_{o_t} L$, for ease of notation, as:

$$\frac{\partial L}{\partial o_t} = \frac{\partial L}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial o_t} \tag{17}$$

Now, we will recompute our gradients based on the formulas for a single timestamp but extended for all of them.

The gradient $\frac{\partial L}{\partial W_{ya}}$ is computed as:

$$\frac{\partial L}{\partial W_{ya}} = \sum_{t=1}^{T} \left(\frac{\partial L}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial W_{ya}} \right) = \sum_{t=1}^{T} \left(\frac{\partial L}{\partial o_t} \cdot a_t^T \right) \quad \square \tag{18}$$

Analogously, the bias term b_y is the sum of the first term only:

$$\frac{\partial L}{\partial b_y} = \sum_{t=1}^{T} \left(\frac{\partial L}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial b_y} \right) = \sum_{t=1}^{T} \left(\frac{\partial L}{\partial o_t} \right) \quad \square \tag{19}$$

Next, we compute $\frac{\partial L}{\partial a_t}$. For the last timestamp $t = T$, the state a_T is only passed to the final output $\hat{y}_T$.

Thus:

$$\frac{\partial L}{\partial a_T} = \frac{\partial L}{\partial \hat{y}_T} \cdot \frac{\partial \hat{y}_T}{\partial a_T} = \frac{\partial L}{\partial o_T} \cdot W_{ya} \tag{20}$$

For earlier timestamps ($t < T$), we need to account for the gradient from the subsequent state. Therefore:

$$\frac{\partial L}{\partial a_t} = \frac{\partial L}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial a_t} + \frac{\partial L}{\partial a_{t+1}} \cdot \frac{\partial a_{t+1}}{\partial a_t} = \frac{\partial L}{\partial o_t} \cdot W_{ya} + \frac{\partial L}{\partial a_{t+1}} \cdot W_{aa} \tag{21}$$

These two scenarios can be combined into one equation:

$$\frac{\partial L}{\partial a_t} = \sum_{i=t}^{T} \left(W_{aa}^{\top}\right)^{T-i} W_{ya}^{\top} \frac{\partial L}{\partial o_{T+t-i}} \quad \square \tag{22}$$

Looking closer at this equation, we can now understand the biggest fallback RNNs face during training, the gradient vanishing/exploding problem. If we unroll the RNN for a large input sequence, during the backwards pass, the update term of a_t will involve large powers of the term Waa. Raising this matrix to a large power will potentially lead to the values approaching either 0 or infinity. This makes training an RNN for large inputs difficult; if not impossible.

Gradient clipping is sometimes used as a half-solution to the problem, by clipping the calculated terms to a range like [5,5]. This prevents gradients from exploding, but updates the weights by a value that does not correspond to the true gradient, and does not address the gradient vanishing problem.

This was one of the motivations behind more modern RNN architectures, such as LSTMs and GRUs, which alleviate the problem, by introducing gates that regulate the gradient flow much better.

Having this gradient term computed, we can backpropagate through the tanh function:

$$\frac{\partial L}{\partial z_t} = \frac{\partial L}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial z_t} = \frac{\partial L}{\partial a_t} \cdot \left(1 - a_t^2\right) \tag{23}$$

Given this, we can finally compute the gradients of L w.r.t W_{aa}, W_{ax}, b_a as:

$$\frac{\partial L}{\partial W_{aa}} = \sum_{t=1}^{T} \left(\frac{\partial L}{\partial z_t} \cdot a_{t-1}^{T} \right) \quad \square \tag{24}$$

$$\frac{\partial L}{\partial W_{ax}} = \sum_{t=1}^{T} \left(\frac{\partial L}{\partial z_t} \cdot x_t^T \right) \quad \square \tag{25}$$

$$\frac{\partial L}{\partial b_a} = \sum_{t=1}^{T} \left(\frac{\partial L}{\partial z_t} \right) \quad \square \tag{26}$$

4 Part C: Application to Generative Datasets

In this section, we apply the Recurrent Neural Network (RNN) model developed in Part A to a practical task of character-level text generation. The goal is to train the RNN to learn patterns in sequences of characters and generate new sequences based on the learned patterns. We use two datasets for this purpose: names of existing chemical elements and movie titles.

The process involves the following steps:

- Load the datasets and preprocess them by splitting the names into single characters to create an alphabet of unique characters.
- Define the forward and backward propagation operations to calculate the loss and gradients for updating the model parameters.
- Implement gradient clipping to facilitate stable learning and generalization.
- Train the model to generate new names for chemical elements and movie titles.

We will discuss the setup, results, and observations from training the model on these datasets, as well as answer related theoretical questions to provide insights into the model's performance and potential improvements.

4.1 Setup

For this experiment, we use a single-layer RNN trained over 70,000 iterations with a learning rate of 0.1. The model is designed to minimize the cross-entropy loss function, which is well-suited for classification tasks involving multiple classes. Gradient clipping is implemented to maintain stable learning and prevent the exploding gradient problem.

The weight matrices and vector sizes are configured as follows: W_{aa} is 100x100, W_{ax} is vocab_size x vocab_size, and W_{ya} is vocab_size x 100. Each hidden state a_t has a size of 100x1, the input vector x_t is vocab_size x 1, and the output vector y_t is vocab_size x 1. The bias terms b_a and b_y are vectors of sizes 100x1 and vocab_size x 1, respectively.

The value 100 is a hyperparameter that refers to the dimension of the hidden state at each timestamp. Bigger hidden states allow the network to retain more complex information as we unroll the input, with the downside being the increasing risk of overfitting. As with all hyperparameters, the optimal value is application dependent and must be searched for using Grid Search, Bayesian Optimization, etc.

4.2 Generating New Chemical Element Names

The goal of this experiment is to train the RNN model to generate new names for chemical elements by learning patterns from a dataset of existing chemical element names. The vocabulary size for this dataset is 25.

During training with a hidden state dimension of 100, the best results were observed at iteration 12,000 with a loss of 12.529147. When training with a hidden state dimension of 50, the best results were observed at iteration 30,000 with a loss of 6.129626. The generated names are presented in the table below:

	Hidden State 50	**Hidden State 100**
Loss	6.129626	12.529147
Iteration	30,000	12,000
Generated Names	Targon Tadgium Tine Tadcium Tine Oadhium Tine	Piupspemium On Pturice Peicker Toorium Hadium Ton

Table 1: Generated Chemical Names with Different Hidden State Dimensions

Question C: What do you observe about the new generated chemical names? How do you interpret the format of the new generated chemical element names?

The generated names often end in "-ium" and resemble real chemical elements. However, names from the larger hidden state dimension sometimes contain unrealistic sequences, indicating potential overfitting. Names generated with a hidden state dimension of 50 appear more consistent and realistic, suggesting better generalization during training.

4.3 Generating New Movie Titles

In this subsection, we aim to generate new movie titles using the RNN model. The vocabulary size for this task is 43. We experimented with two different hidden state dimensions: 30 and 50. The best results are summarized in table 2.

	Hidden State 30	**Hidden State 50**
Loss	33.902076	37.672859
Iteration	60,000	18,000
Generated Titles	Thitog nais The 400nd and the wilds Toy speanday The doud Ursent on wator Ga The lerl	The tramedfathe taringot Peaf Rysuran The grin- Ving Ie The of thieris

Table 2: Generated Movie Titles with Different Hidden State Dimensions

Question D: What happens when you train your model for the movie titles dataset? What do you observe and how do you interpret this observation?

The generated movie titles exhibit some patterns and structures similar to real titles, but are for the most part nonsense. There are inconsistencies and nonsensical sequences, particularly with a larger hidden state dimension. This suggests that while the model captures some patterns in the data, it struggles with coherence and plausibility, possibly due to overfitting or insufficient data diversity.

Question E: Can you think of ways to improve the model for the case of the movie title dataset?

To improve the model, consider the following approaches:

- **Increase Dataset Diversity**: Use a more diverse dataset to provide the model with a broader understanding of movie titles.
- **Pre-train on Diverse Text Data**: Train the model on a large, diverse text dataset to help it understand the structure and flow of the English language.
- **Tokenize by Words**: Instead of characters, use word-level tokenization to capture more meaningful patterns. This will increase the vocabulary size and computational complexity but improve coherence.
- **Use Advanced Models**: Implement more sophisticated RNN variants like LSTMs and GRUs to better manage long-term dependencies and improve performance.
- **Transformer Models**: Consider using transformer models, which are state-of-the-art for NLP tasks and can handle long-range dependencies and complex patterns more effectively than RNNs.