Recurrent Deep Learning Models and its applications

Ngai Ho Wang

February 19, 2025

Contents

1 Introduction

With the rise of the Generative Artificial Intelligence, the development of AI has already made remarkable strides in processing sequential data. In understanding and producing sequential data. It has applications ranging from Natural Language Processing (NLP) to music composition to video generation. Especially NLP, has emerged as a pivotal field in artificial intelligence, enable machines to understand, interpret and generate in human readable format. Some famous Artificial Intellience assistence for example, Siri, Alexa and Bixby have shown the possibility. Everyone can communicate with those machines, which make the reasonable response back to user.

Recurrent Neural Networks (RNNs) have been a foundational architecture in this domain, Unlike the traditional Artificial Neural Network, RNNs do not treat each input independently, RNNs handle each input by considering the information from previous inputs. Conceptually this architecture able to retain the information. Thus, this architecture is suitable for handling sequential data. Unfortunately, early RNNs had limitation in training of networks over long sequence. vanishing and exploding gradient problems significantly affect the training process of RNN (Bengio et al., 1994). Eliminating many practical applications of RNNs. After that, (Hochreiter & Schmidhuber, 1997) introduced Long Short-Term Memory (LSTM) networks and are responsible for the breakthrough in how to solve these challenges. Specificized gating mechanisms were introduced in LSTMs to regulate the flow of the information, minimize the vanishing gradient problem and learn the long-term dependencies. This advanced made RNNs much more performant on tasks like a language modeling, machine translation and speech recognition tasks.

Further improvements were achieved with Gated Recurrent Units (GRUs) by (Cho et al., 2014) which diminished the LSTM architecture's complexity, but still provided the same performance. GRUs performed comparably but used fewer parameters, making it computationally and more tractably trainable.

Since the craze of AI has been revived by generative AI, natural language processing to time series prediction and speech recognition have once again aroused people's interest in RNN. This report aims to:

- Explore the theoretical foundations of recurrent deep learning models.
- Investigate their diverse applications in solving sequential data tasks.
- Analyze their performance, strengths, and inherent limitations.

2 Background and Literature Review

2.1 Evolution of Deep Learning and Recurrent Neural Networks

In the past few decades, thank to the rapidly development of technology, the computing resource has a incredible increase. Thus, substantially deep learning architecture have improved, from simple architectures, which only able to capture simple information from data to sophisticated models that are able to learn complex, abstract representations. This was before the early neural networks like perceptrons and multilayer perceptrons (MLPs) laid the footwork of neural computation that first came in the picture, but were

burdened by the lack of ability to model sequential dependencies. This however imposed a limit on the feed forward paradigm, which prompted the development of recurrent neural networks (RNNs) that extend the old stalactite of feed forward paradigm with cyclic connections. Through these connection, RNNs are capable to keep a hidden state that represents information over time steps thereby effectively capture temporal dynamics. RNNs have been a decisive step in the evolution of deep learning, as they are able to do tasks that require memory of previous events, including problems of natural language processing and time series modeling. Despite that, early RNNs models suffered from serious problems for example, vanishing gradients and exploding gradients, which prevented these RNN models from learning long ranged dependencies. This stimulated the building of more refined architectures intended to side step these obstacles.

2.2 Literature Review

2.2.1 Backpropagation Through Time

BPTT is one of the most important algorithms used for training RNNs. Dating back to the original effort to expand the typical backpropagation algorithm, BPTT has been formulated to handle the difficulties of temporal sequences that are inherent in sequential data (Werbos, 1990). This algorithm allows RNNs in learning sequence dependent data by unfold the network over time steps and then updating weights matrix through the gradient of loss function with respect to the variable (Rumelhart et al., 1986).

Conceptual Framework of BPTT

BPTT works based on the technique of treating an RNN as a deep feedforward network for across multiple time steps. In the forward pass, the RNN, like other artificial neuronal network, applies operation over the data input in sequence, bringing changes in its own state variables at every time step, depending on the input and the previous state of its general working state or hidden state. This sequential processing produces outputs and stores the internal states of the network in any period (Werbos, 1990).
This unfolds the RNN to construct a traditional Feedforward Neural Network where we can apply backpropagation through time. Below is the conceptual idea of BPTT in RNN.

Notation	Meaning	Dimension
U	Weight matrix for input to hidden state	$input\ size \times hidden\ unites$
W	Weight matrix for hidden to hidden state	$hidden\ units \times hidden\ unites$
V	Weight matrix for hidden state to output state	$hidden\ units \times number\ of\ class$
x_t	Input vector at time t	$input\ size \times 1$
h_t	Hidden state output at time t	$hidden\ units \times 1$
b_h	Bias term for hidden state	$hidden\ units \times 1$
b_y	Bias term for output state	$number\ of\ class \times 1$
$\hat{o}_y$	Output at time t	$number\ of\ class \times 1$
$\hat{y}_t$	Output at time t	$hidden\ units \times 1$
$\mathcal{L}$	Loss at time t	$scalar$

Table 1: Unfolded RNN

Forward Pass

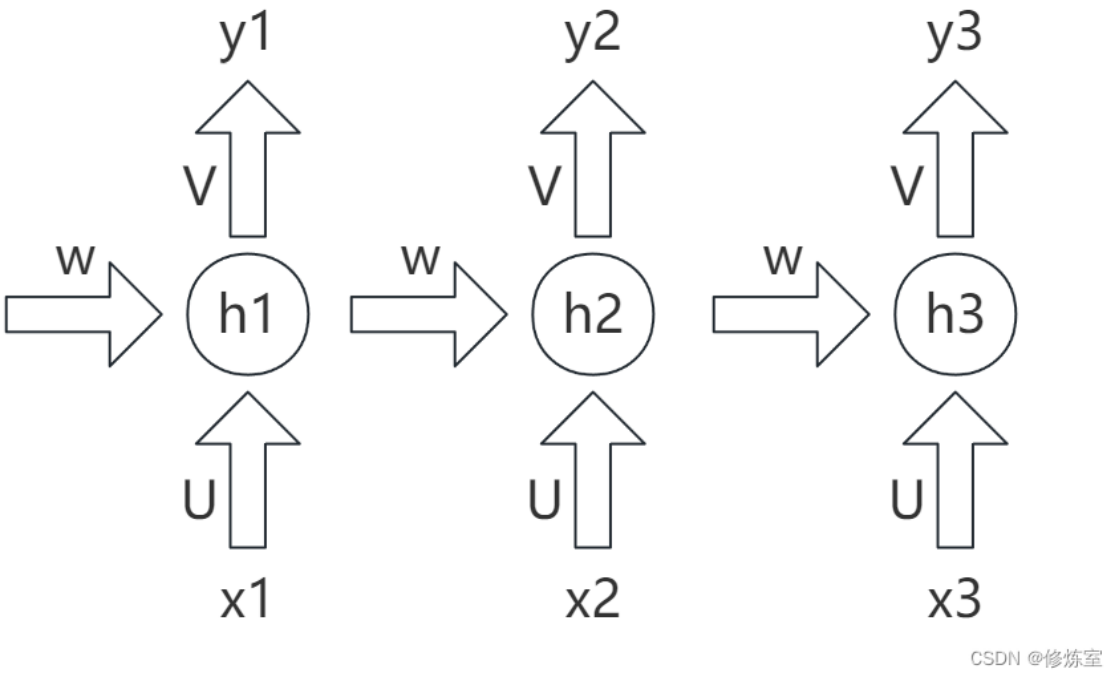


Figure 1: Unfolded RNN

During the forward pass, the RNN processes the input sequence sequentially, computing hidden states and output at each timestep:

$$h_t = f(U^T x_t + W^T h_{t-1} + b_h) \tag{1}$$

$$\hat{y}_t = f(V^T h_t + b_y) \tag{2}$$

Computing the loss function

Assuming the loss is computed only at the final timestep t:

$$\mathcal{L}_t = L(y_t, \hat{y}_t) \tag{3}$$

In order to do backpropagation through time to tune the parameters in RNN, we need to calculate the partial derivative of loss function $\mathcal{L}$ with respect to the differently parameters.

Backward pass using the chain rule

Using the chain rule for computing the gradient.
Partial derivative of loss function $\mathcal{L}$ with respect to W (hidden to hidden state) at time 2.

$$\frac{\partial \mathcal{L}_2}{\partial W} = \frac{\partial \mathcal{L}_2}{\partial y_2} \cdot \frac{\partial y_2}{\partial h_2} \cdot \frac{\partial h_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial W} \tag{4}$$

By mathematic induction

$$\frac{\partial \mathcal{L}_t}{\partial W} = \frac{\partial \mathcal{L}_t}{\partial y_t} \cdot \frac{\partial y_t}{\partial h_t} \cdot (\sum_{i=1}^{t} \frac{\partial h_t}{\partial h_i} \cdot \frac{\partial h_i}{\partial W}) \tag{5}$$

Where

$$\frac{\partial h_t}{\partial h_i} = \prod_{j=i+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \tag{6}$$

Partial derivative of loss function $\mathcal{L}$ with respect to U (input to hidden state) at time 2.

$$\frac{\partial \mathcal{L}_2}{\partial U} = \frac{\partial \mathcal{L}_2}{\partial y_2} \cdot \frac{\partial y_2}{\partial h_2} \cdot \frac{\partial h_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial U} \tag{7}$$

By mathematic induction

$$\frac{\partial \mathcal{L}_t}{\partial U} = \frac{\partial \mathcal{L}_t}{\partial y_t} \cdot \frac{\partial y_t}{\partial h_t} \cdot (\sum_{i=1}^{t} \frac{\partial h_t}{\partial h_i} \cdot \frac{\partial h_i}{\partial U}) \tag{8}$$

Where

$$\frac{\partial h_t}{\partial h_i} = \prod_{j=i+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \tag{9}$$

Partial derivative of loss function $\mathcal{L}$ with respect to V (hidden to output state) at time 2.

$$\frac{\partial \mathcal{L}_2}{\partial V} = \frac{\partial \mathcal{L}_2}{\partial y_2} \cdot \frac{\partial y_2}{\partial h_2} \cdot \frac{\partial h_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial V} \tag{10}$$

By mathematic induction

$$\frac{\partial \mathcal{L}_t}{\partial V} = \frac{\partial \mathcal{L}_t}{\partial y_t} \cdot \frac{\partial y_t}{\partial h_t} \cdot (\sum_{i=1}^{t} \frac{\partial h_t}{\partial h_i} \cdot \frac{\partial h_i}{\partial V}) \tag{11}$$

Where

$$\frac{\partial h_t}{\partial h_i} = \prod_{j=i+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \tag{12}$$

Partial derivative of loss function $\mathcal{L}$ with respect to b_h (bias term in hidden state) at time 2.

$$\frac{\partial \mathcal{L}_2}{\partial b_b} = \frac{\partial \mathcal{L}_2}{\partial y_2} \cdot \frac{\partial y_2}{\partial h_2} \cdot \frac{\partial h_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial b_h} \tag{13}$$

By mathematic induction

$$\frac{\partial \mathcal{L}_t}{\partial b_h} = \frac{\partial \mathcal{L}_t}{\partial y_t} \cdot \frac{\partial y_t}{\partial h_t} \cdot (\sum_{i=1}^{t} \frac{\partial h_t}{\partial h_i} \cdot \frac{\partial h_i}{\partial b_h}) \tag{14}$$

Where

$$\frac{\partial h_t}{\partial h_i} = \prod_{j=i+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \tag{15}$$

Partial derivative of loss function $\mathcal{L}$ with respect to b_y (bias term in output state) at time 2.

$$\frac{\partial \mathcal{L}_2}{\partial b_y} = \frac{\partial \mathcal{L}_2}{\partial y_2} \cdot \frac{\partial y_2}{\partial h_2} \cdot \frac{\partial h_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial b_y} \tag{16}$$

By mathematic induction

$$\frac{\partial \mathcal{L}_t}{\partial b_y} = \frac{\partial \mathcal{L}_t}{\partial y_t} \cdot \frac{\partial y_t}{\partial h_t} \cdot (\sum_{i=1}^{t} \frac{\partial h_t}{\partial h_i} \cdot \frac{\partial h_i}{\partial b_y}) \tag{17}$$

Where

$$\frac{\partial h_t}{\partial h_i} = \prod_{j=i+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \tag{18}$$

parameters updates

$$W \leftarrow W - \alpha \frac{\partial \mathcal{L}}{\partial W} \tag{19}$$

$$U \leftarrow U - \alpha \frac{\partial \mathcal{L}}{\partial U} \tag{20}$$

$$V \leftarrow V - \alpha \frac{\partial \mathcal{L}}{\partial V} \tag{21}$$

$$b_h \leftarrow b_h - \alpha \frac{\partial \mathcal{L}}{\partial b_h} \tag{22}$$

$$b_y \leftarrow b_y - \alpha \frac{\partial \mathcal{L}}{\partial b_y} \tag{23}$$

Pseudocode of BPTT (Wikipedia, 2023)

Algorithm 1 Backpropagation Through Time (BPTT)

Input:
  Sequence of input data $\{x_1, x_2, \ldots, x_T\}$
  Sequence of target outputs $\{y_1, y_2, \ldots, y_T\}$
  Learning rate η
  Number of time steps to unroll N
Initialize: Model parameters θ, hidden state $h_0 = 0$
Forward Pass:
for $t = 1$ to T **do**
  Compute hidden state: $h_t = f(h_{t-1}, x_t; \theta)$
  Compute output: $\hat{y}_t = g(h_t; \theta)$
  Compute loss for time step t: $L_t = \mathcal{L}(\hat{y}_t, y_t)$
end for
Backward Pass (BPTT):
Set total loss: $L = \sum_{t=1}^{T} L_t$
for $t = T$ down to 1 **do**
  Compute gradient of loss with respect to output: $\frac{\partial L_t}{\partial \hat{y}_t}$
  Backpropagate through output layer to obtain: $\frac{\partial L_t}{\partial h_t}$
  Accumulate gradients for parameters: $\frac{\partial L}{\partial \theta}$
  for $k = 1$ to N **do**
    Backpropagate through time for N steps:
    Compute gradient contribution from step $t - k$: $\frac{\partial L_t}{\partial h_{t-k}}$
  end for
end for
Update Parameters:
$\theta = \theta - \eta \cdot \frac{\partial L}{\partial \theta}$
Output: Updated parameters θ

2.2.2 Activation Function

Activation functions, particularly the sigmoid function, are fundamental components of recurrent neural networks (RNNs). They transform input data into output data. A key property of these functions is their differentiability. Differentiability is crucial for the backpropagation through time (BPTT) algorithm, enabling the application of the chain rule during training.

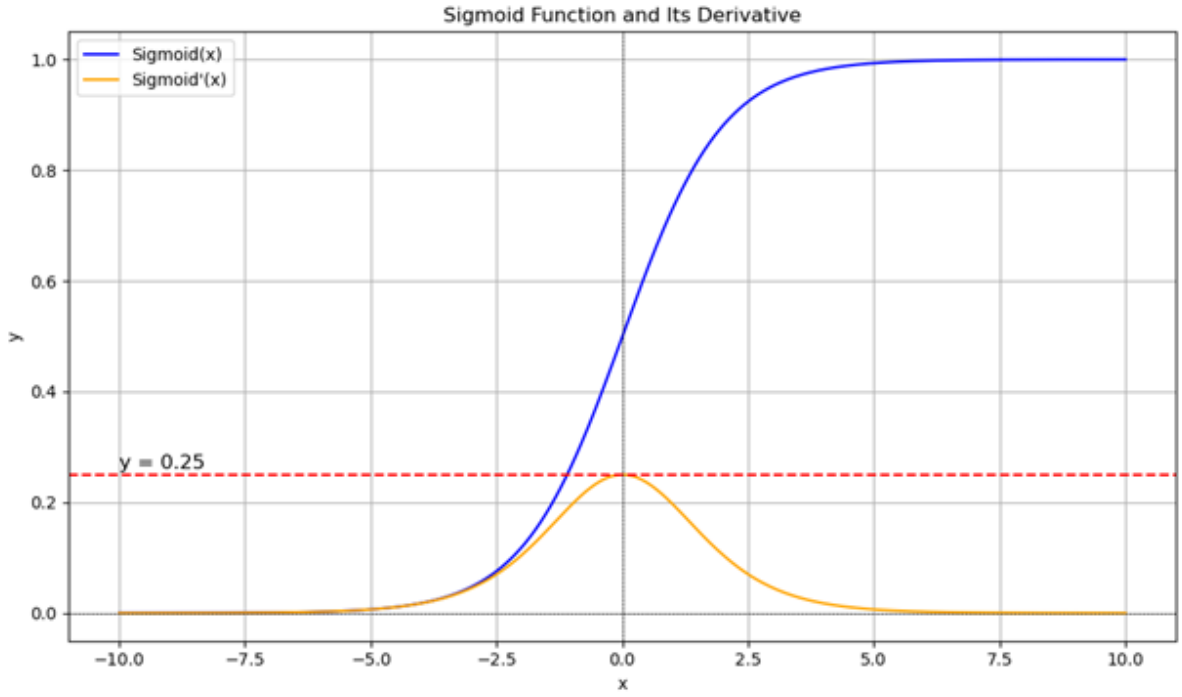
Sigmoid activation function

The main role of the sigmoid activation function is to normalize candidate values and convert the cell state to a hidden state when performing cell state updates. It limits the output between [0,1] because it has a smooth gradient, which is important for discovering long-range dependencies.

$$Sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{24}$$

$$Sigmoid'(x) = Sigmoid(x)(1 - Sigmoid(x)) \tag{25}$$

Below is the sigmoid function and its derivative.



$Domain(Sigmoid(x)) = \mathbb{R}, \quad Codomain(Sigmoid(x)) = (0, 1)$
$Domain(Sigmoid'(x)) = \mathbb{R}, \quad Codomain(Sigmoid'(x)) = [0, 0.5]$

Hyperbolic tangent activation function

The main role of the hyperbolic tangent (tanh) activation function is to normalize candidate values and convert the cell state to a hidden state when performing cell state updates. It limits the output between [-1,1] because it has a stable gradient, which is important for discovering long-range dependencies.

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{26}$$

$$tanh'(x) = 1 - tanh^2(x) \tag{27}$$

Below is the Hyperbolic tangent activation function and its derivative.

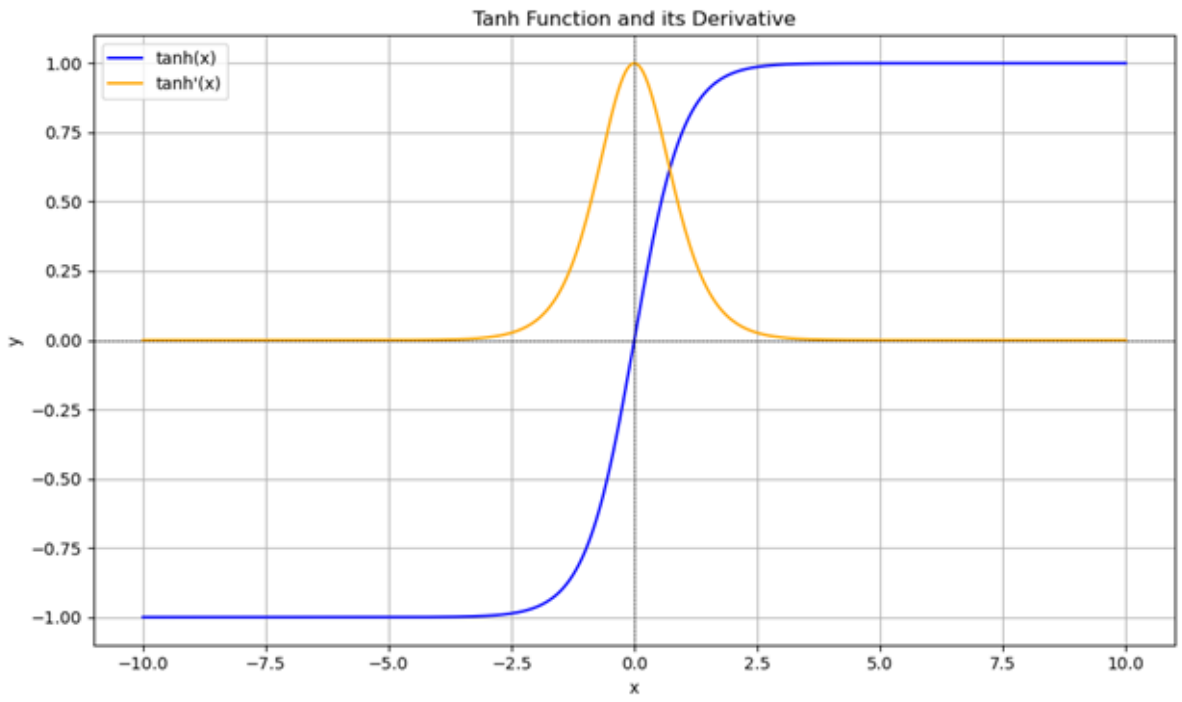
$Domain(tanh(x)) = \mathbb{R}, \quad Codomain(tanh(x)) = [-1, 1]$
$Domain(tanh'(x)) = \mathbb{R}, \quad Codomain(tanh'(x)) = [0, 1]$

2.2.3 Gradient vanishing and gradient exploring

When training the RNN, BPTT was used to update the weight matrix. As the number of time steps increase, the problem of gradient instability of often encountered, and this problem is gradient vanishing and gradient exploring (Bengio et al., 1994).

Vanishing Gradients

Generally, sigmoid activation function is used commonly in RNNs, has a maximum derivative of 0.25. When doing BPTT in long time steps, this multiplication results in exponentially diminishing gradients as the sequence length increases. Consequently, the shallow



neural receive very small gradient updates, making it difficult to adjust the parameters effectively. This leads to the model struggling to learn long time dependencies.

Exploding Gradients

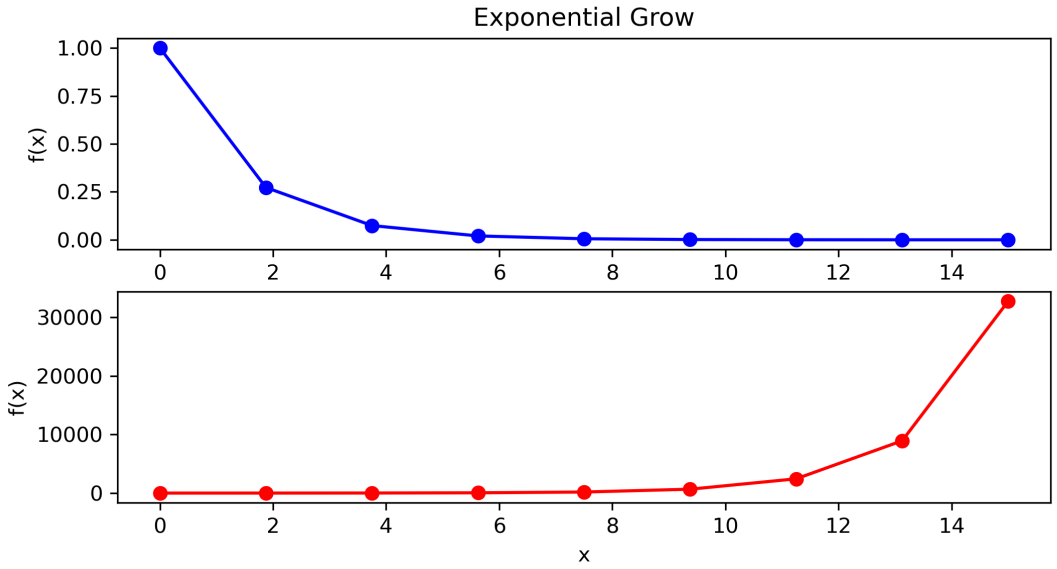
When we are doing the feedforward and get super large value computed by loss function. Then when updating the parameters. The updates to the weights will also be large. Resulting in higher loss and larger gradients in the next iterations. This will lead to exploding gradients.

We have introduce the backpropagation through time. This is the method to update the parameters in RNNs. When calculating, for example, the partial derivative of loss function with respect to W. Assume the time t go to infinity large. We will get this term. $\prod_{j=i+1}^{t} \frac{\partial h_j}{\partial h_{j-1}}$, and it will lead to exponential problem. if $\frac{\partial h_j}{\partial h_{j-1}} > 1$. Then the product of all term will increase exponentially, then exploding gradients occur. On the contrary, if $\frac{\partial h_j}{\partial h_{j-1}} < 1$. Then the result will decrease exponentially, then vanishing gradients occur.

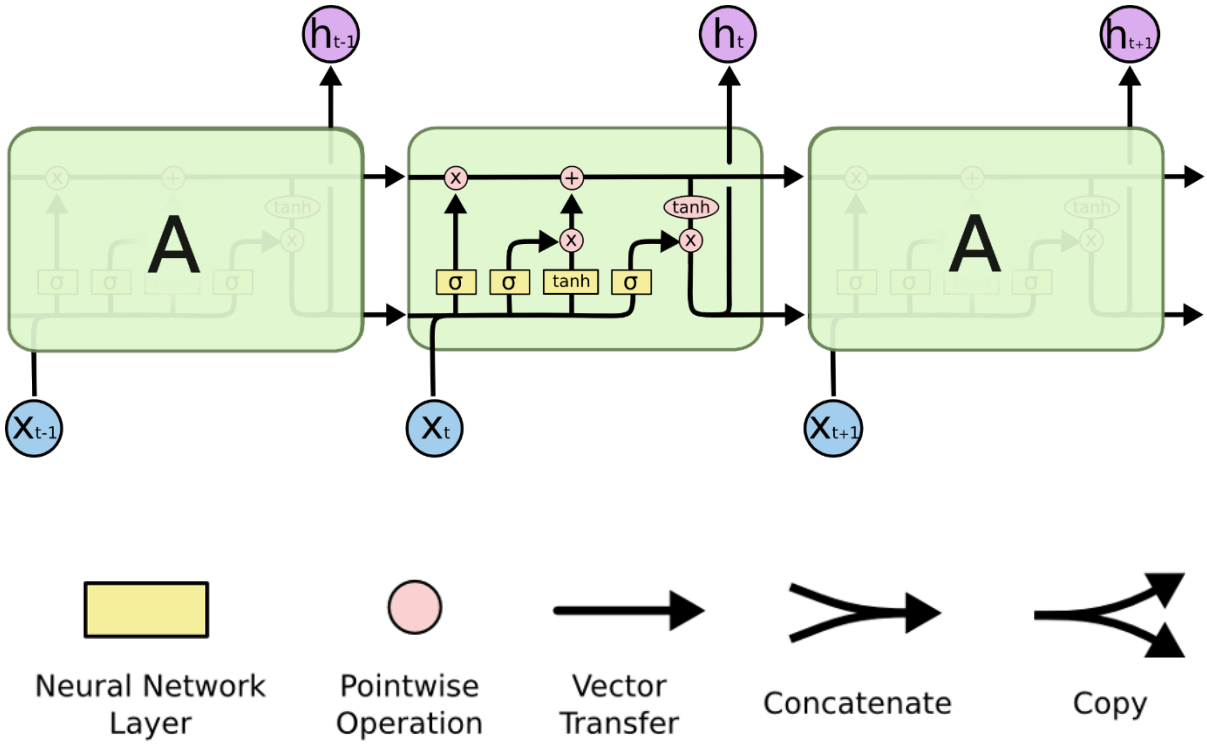
2.2.4 Long short-term memory (LSTM)

Long short-term memory proposed by (Hochreiter & Schmidhuber, 1997). LSTM is designed for handling long time step problems. The architecture of LSTM can prevent vanishing gradient and exploring gradient. The main difference between LSTM and RNN is the number of gates. LSTM introduced input, forget and output gates. This allows LSTM to manage the flow of information more effectively, retaining important information over longer sequences.

Architecture



LSTMs introduce a memory cell that can maintain information over long time steps the cell is controlled by three gates, input gate, output gate, and forget gate. Each cell of LSTMs inside has 3 sigmoid and 1 tanh layer. Below graph unfolds the LSTM hence we can analyze different gates.



Forget gate:

The forget gate is a component of the LSTM, designed to manage the flow of information within the cell state. The function of forget gate is to determine which information should be retained in memory cell (Hochreiter & Schmidhuber, 1997).

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{28}$$

Input gate:

The input gate controls how much new information from the current time step is allowed to enter the cell (Hochreiter & Schmidhuber, 1997). For the $\widetilde{C}_t$, the purpose is to suggest updates for the cell state.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_c) \tag{29}$$

$$\widetilde{C}_t = tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \tag{30}$$

Cell State Update:

The forget gate will drop the meaningless information and add some potential information.

$$C_t = f_t * C_{t-1} + i_t * \widetilde{C}_t \tag{31}$$

Output gate:

The output gate is able to control how much or what information from the cell state should be passed to the next layer or used in predictions.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{32}$$

Hidden State Update:

The hidden state is influenced by output value and current cell state.

$$h_t = o_t * tanh(C_t) \tag{33}$$

n = number of features in the input vector x_t.
m = number of units in LSTM.

Notation	**Meaning**	**Dimension**
x_t	Input vector at time t	$n \times 1$
h_t	Hidden state output at time t	$m \times 1$
C_t	Cell state at time t	$m \times 1$
f_t	Forget gate output at time t	$m \times 1$
i_t	Input gate output at time t	$m \times 1$
o_t	Output gate output at time t	$m \times 1$
$\widetilde{C}_t$	Candidate memory cell at time t	$m \times 1$
W_f	Weight matrix for the forget gate	$m \times (m+n)$
W_i	Weight matrix for the input gate	$m \times (m+n)$
W_C	Weight matrix for the candidate memory cell	$m \times (m+n)$
W_o	Weight matrix for the output gate	$m \times (m+n)$
b_f	Bias vector for the forget gate	$m \times 1$
b_i	Bias vector for the input gate	$m \times 1$
b_C	Bias vector for the candidate memory cell	$m \times 1$
b_o	Bias vector for the output gate	$m \times 1$

Table 2: Unfolded RNN

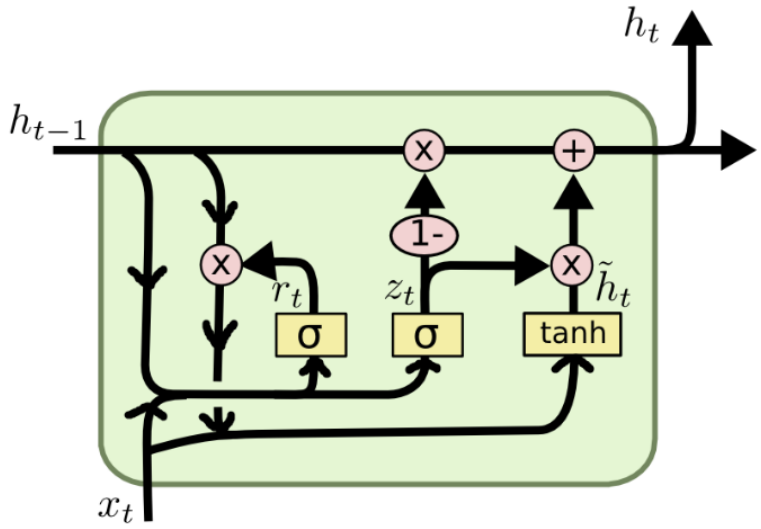
Number of parameters:

1. Weights matrix for the input
    - Forget gate: $n \times m$
    - Input gate: $n \times m$
    - Cell gate: $n \times m$
    - Output gate: $n \times m$
2. Weight matrix for the hidden state
    - Hidden state for forget gate: $m \times m$
    - Hidden state for input gate: $m \times m$
    - Hidden state for cell gate: $m \times m$
    - Hidden state for output gate: $m \times m$
3. Bias term
    - Bias for forget gate: $1 \times m$
    - Bias for input gate: $1 \times m$
    - Bias for cell gate: $1 \times m$
    - Bias for output gate: $1 \times m$

Total parameters: $4 \times (n + m + 1) \times m$

2.2.5 Gated Recurrent Unit (GRU)

The gated recurrent unit (GRU) was proposed by (Cho et al., 2014) to make each recurrent unit to adaptively capture dependencies of different time scales. The GRU has 2 gates, update gate and reset gate. Update gate:



The update gate determines how much of the past information should be retained in the current hidden state.

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z) \tag{34}$$

Reset gate:

The reset gate is similar with the update gate, but the candidate hidden state is influenced by the reset gate.

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r) \tag{35}$$

Candidate hidden state:

The candidate hidden state combined with previous hidden state and current input to form the potential new information that can be added to the current hidden state.

$$\widetilde{h_t} = tanh(W_h \cdot [r_t * h_{t-1}, x_t] + b_h) \tag{36}$$

Final hidden state

The final hidden state of the GRU at time t is a linear interpolation between the previous final hidden state and the candidate hidden state.

$$h_t = (1 - z_t) * h_{t-1} + z_t * \widetilde{h_t} \tag{37}$$

Notation	**Meaning**	**Dimension**
x_t	Input vector at time t	$n \times 1$
h_t	Hidden state output at time t	$m \times 1$
r_t	Reset gate output at time t	$m \times 1$
z_t	Update gate output at time t	$m \times 1$
W_z	Weight matrix for the update gate	$m \times (m+n)$
W_r	Weight matrix for the candidate memory cell	$m \times (m+n)$
W_h	Weight matrix for the output gate	$m \times (m+n)$
b_z	Bias vector for the input gate	$m \times 1$
b_r	Bias vector for the candidate memory cell	$m \times 1$
b_h	Bias vector for the output gate	$m \times 1$

Number of parameters:

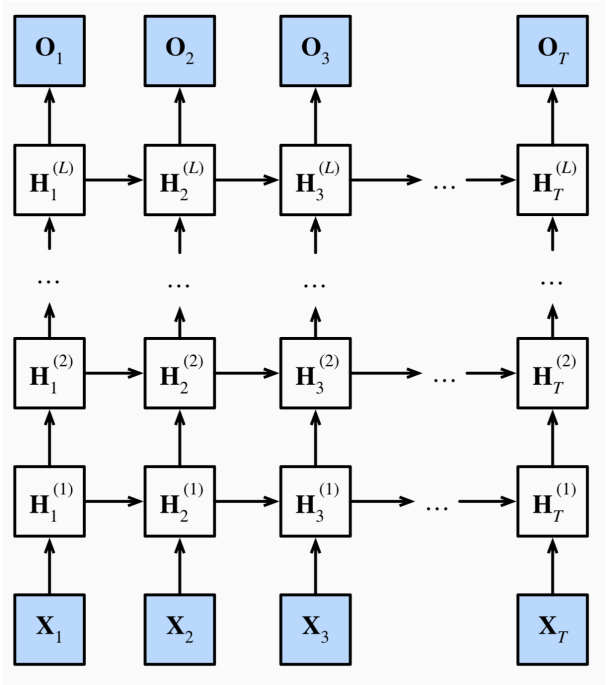
1. Weights matrix for the input
   - Update gate: $n \times m$
   - Reset gate: $n \times m$
   - Candidate hidden state: $n \times m$
2. Weight matrix for the hidden state
   - Hidden state for update gate: $m \times m$
   - Hidden state for reset gate: $m \times m$
   - Hidden state candidate hidden state: $m \times m$
3. Bias term
   - Bias for update: $1 \times m$
   - Bias for reset: $1 \times m$
   - Bias for candidate hidden state: $1 \times m$

Total parameters: $3 \times (n + m + 1) \times m$

2.2.6 Deep recurrent neural networks (DRNNs)

Deep architecture networks with multiple layers that can hierarchically learn complex representations (Bengio, 2009). By extending of this concept, stacking recurrent layers to form Deep Recurrent Neural Networks aligns with this principle. A number of research papers have been prove that the performance of DRNNs is out-performance then conventional RNNs. (Delalleau & Bengio, 2011; Le Roux & Bengio, 2010; Pascanu et al., 2013)

The architecture of DRNNs are similar with conventional RNNs. We simply stack the recurrent layers vertically. For the first layer, this layer receives the input and combines it with its previous hidden state h_{t-1} (Equation 38). The second layer receive the hidden state of the first layer and treat as the input of the layer 2 (Equation 39). We can extend this concept to L layers (Equation 40).



$$h_t^{(1)} = f(W_{xh}^{(1)} x_t + W_{hh}^{(1)} h_{t-1}^{(1)} + b_h^{(1)}) \tag{38}$$

$$h_t^{(2)} = f(W_{xh}^{(2)} h_t^{(1)} + W_{hh}^{(2)} h_{t-1}^{(2)} + b_h^{(2)}) \tag{39}$$

$$h_t^{(L)} = f(W_{xh}^{(L)} h_t^{(L-1)} + W_{hh}^{(L)} h_{t-1}^{(L)} + b_h^{(L)}) \tag{40}$$

$$o_t = W_{hy} h_t^{(L)} + b_y \tag{41}$$

$$y_t = g(o_t) \tag{42}$$

Where x_t is the input at time t. $h_t^{(l)}$ is the hidden state for the l layer at time t. $W_{xh}^{(l)}, W_{hh}^{(l)}$ are the weight matrices for the input to hidden and hidden to hidden connections in layer l, respectively. W_{hy} is weight matrix for the output layer. $b_h^{(l)}$ is the bias vector for the l layer (except output layer), b_y is the bias vector for output layer. $g(\cdot)$ and $f(\cdot)$ are an activation function.

2.2.7 Hidden Markov Model

Before reviewing Hidden Markov Model (HMM), it is essential to understand what Markov models is, or Markov chains. Markov chains are fundamental models in probability theory and statistic, and it is a stochastic process describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event. This property known as the Markov property (L. Rabiner & Juang, 1986; Roberts & Rosenthal, 2004). In rigorous terms. Let the state space be defined as:

$$S = \{s_1, s_2, \ldots, s_N\}. \tag{43}$$

$$P(s_{t+1} \mid s_t, s_{t-1}, s_{t-2}, \cdots, s_1) = P(s_{t+1} \mid s_t) \tag{44}$$

The state transition probability is denoted by:

$$P_{ij} = P(s_{t+1} = s_j \mid s_t = s_i). \tag{45}$$

Given an initial state s_1 with probability $P(s_1)$, the joint probability of a state sequence $\{s_1, s_2, \ldots, s_T\}$ can be written as:

$$P(s_1, s_2, \ldots, s_T) = P(s_1) \cdot P(s_2 \mid s_1) \cdot P(s_3 \mid s_2) \cdots P(s_T \mid s_{T-1}) \tag{46}$$

We can simplfy the joint probability of a state sequence as equestion 47.

$$P(s_1, s_2, \ldots, s_T) = P(s_1) \prod_{t=1}^{T-1} P(s_{t+1} \mid s_t) \tag{47}$$

Then it is essential to discuss marginal probability in a Markov model because it provides critical information about the likelihood of being in particular state s_j at a specific time t. The marginal probability of being in state s_j at time t defined as:

$$\pi_t(s_j) = P(s_t = s_j) \tag{48}$$

It is the probability of the system being in state s_j at time t, irrespective of how the system transitioned to s_j. We can obtainde by summing over all possible transition probability P_{ij} where s_i belong in state space.

$$\pi_{t+1}(s_j) = \sum_{s_i \in S} \pi_t(s_i) \cdot P_{ij} \tag{49}$$

If Markov chain is irreducible and aperiodic, the stationary distribution defined as below:

$$\pi_j = \sum_{i=1}^{n} \pi_i P_{ij} \tag{50}$$

Hidden markov model

Hidden Markov Model (HMMs) is a statistical model that extends the Markov chains by introducting a layer of latent (unobservable) variables. And connecting them to observable outputs (emission probability). In contrast to standard Markov chains, hidden states are never be observed directly. Instead, we observe a sequence of data (observations) that are probabilitically generated by these hidden state. This structure has proven useful for applications in speech recognition, biological sequence analysis, and signal processing

(L. R. Rabiner, 1989). Lets the hidden states be denoted by X_t, where X_t is the hidden state at time t. The hidden state space is defined as:

$$\mathcal{X} = \{s_1, s_2, \cdots, s_n\} \tag{51}$$

where $\mathcal{X}$ is finite set of size n, and each $s_i \in \mathcal{X}$ represents a specific possible hidden state. Then for the observable space represents the possible outcomes (observations) that are generated by the hidden state, we have discuss previously. Let the observations at time t be denoted by Y_t. The observable space is defined as:

$$\mathcal{Y} = \{o_1, o_2, \cdots, o_m\} \tag{52}$$

where $\mathcal{Y}$ is finite set of size m, and each $o_i \in \mathcal{Y}$ represents a specific possible observation.

Then for the transition probability matrix (P), the probability of transitioning between hidden states denoted as:

$$P_{n\times n} = \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1n} \\ p_{21} & p_{22} & \cdots & p_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ p_{n1} & p_{n2} & \cdots & p_{nn} \end{bmatrix} \tag{53}$$

$$p_{ij} = P(X_{t+1} = s_j \mid X_t = s_i) \tag{54}$$

where $s_j, s_i \in \mathcal{X}$.

For emission probability matrix (B), the probabilities of generating an observation given a hidden state.

$$B_{n\times n} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mm} \end{bmatrix} \tag{55}$$

$$b_{ik} = P(Y_t = o_k \mid X_t = s_i) \tag{56}$$

where $s_i \in \mathcal{X}, o_k \in \mathcal{Y}$.

And the goal of hidden markov model is the maximize the joint probability $P(X, Y)$ given the initial probability $P(X_1)$.

$$P(X, Y) = P(X_1) \cdot \prod_{t=1}^{T-1} P(X_{t+1} = s_j \mid X_t = s_i) \cdot \prod_{t=1}^{T} P(Y_t = o_k \mid X_t = s_i) \tag{57}$$

For the NLP tasks, there may have some unobservable data, for example the implicit sentiment, user intent, emotional state. (L. R. Rabiner & Juang, 1993) proposes the hidden Markov model (HMM), the assumption is the existence of the latent process follows a Markov chain from which observations X are generated. In other word, there would exists an unobserved state sequence $Z = \{z_1, z_2 \cdots, z_T\}$ in observed sequence $X = \{x_1, x_2, \cdots, x_T\}$ (Sengupta et al., 2023). Where the hidden states, z_t belonging to state-space $Q = \{q_1, q_2, \cdots, q_M\}$ follow a Markov chain goverened by:

- A state-transition probability matrix $A = [a_{ij}] \in \mathbb{R}^{M\times M}$ where $a_{ij} = p(z_{t+1} = q_j | z_t = q_i)$

- Initial state matrix $\pi = [\pi_i] \in \mathbb{R}_{1\times M}$ with $\pi_i = p(z_1 = q_i)$

Furthermore, for the hidden state z_t, correspoinding to the observe data x_t is release by emission process $B = [b_j(x)]$ where $b_j(x) = p(x|z = q_i)$. We can assume $b_j(x)$ is follows the Gaussian mixture model (GMM).

$$p(x_t|z = q_j) = \sum_{l=1}^{k} c_{jl}\mathcal{N}(x_t|\mu_{jl}, \Sigma_{jl}) \tag{58}$$

where $\sum_{l=1}^{k} c_{jl} = 1$, $\forall j = \{1, \cdots, M\}$, k is the number of Gaussian mixture components and $\mathcal{N}(x_t|\mu_{jl}, \Sigma_{jl})$ denotes a Gaussian probability density with mean μ_{jl} and covariance Σ_{jl} for state j and mixture component l. The number of hidden states (M) and mixture component (k) are the two hyperparameters of the model which have to be provided apriori.

Therefore, the joint probability probability density function of the observation X can be expressed as:

$$p(X) = p(z_1)\prod_{t=1}^{T-1} p(z_{t+1}|z_t)\prod_{t=1}^{T} p(x_t|z_t) \tag{59}$$

The optimal parameters $[A, B, \pi]$, which maximize the likelihood of the observation sequence X (Equation 39), are determined using the Baum-Welch algorithm, an expectation-maximization method (L. R. Rabiner & Juang, 1993). Additionally, the probability of the system in a specific hidden state z_t corresponding to the observation x_t is calculated using the Viterbi algorithm.

2.2.8 Word representation

In RNN, the due to the architecture, the model can only handle vector, it is important that convert the word into machine readable format. By translating words into vectors, these representations capture semantic meanings, relationships and contexts. (Mikolov et al., 2013) proposed 2 model architectures for leaning distributed representations of words.

Continuous Bag of Words model

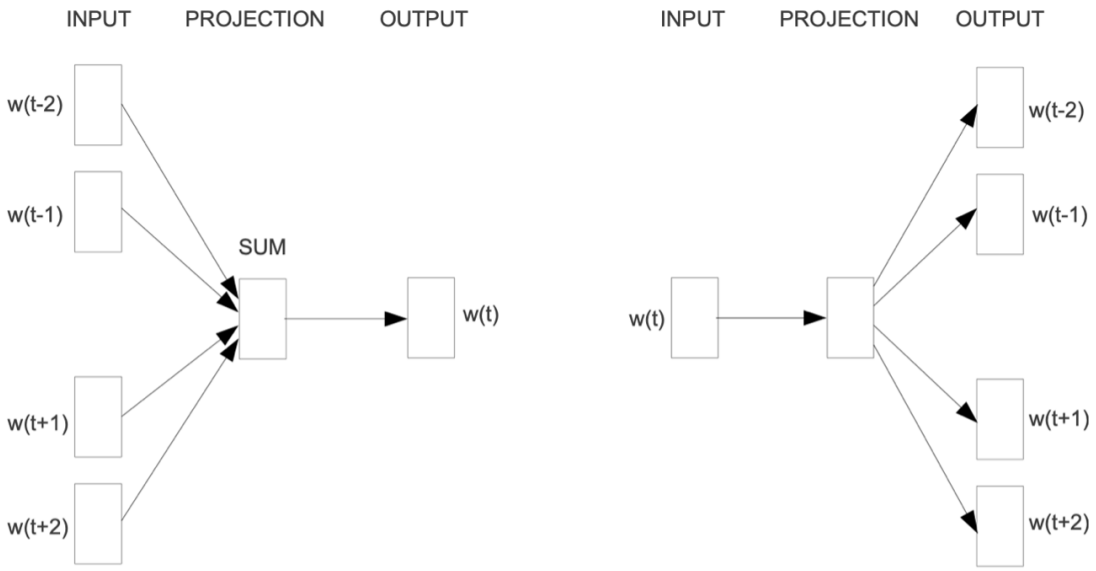
CBOW operates by taking context words around a target word, it is flexible to adjust the window size which mean that we can control how many words around our target word. Hence aggregating their embeddings and passing to hidden layer to produce a probability distribution. Capture the semantic meanings and the relationship more effectively.

$$Q = N \times D + D \times log_2(V) \tag{60}$$

Continuous Skip gram model

The continuous skip gram model operates in opposite direction compare with CBOW. The model takes the surrounding word to predict the target word. In other word, treat the current word as an input to a log-linear classifier with continuous projection layer, and predict the probability distribution of surrounding words.

$$Q = C \times (D + D \times log_2(V)) \tag{61}$$



3 Project Goals and Objectives

3.1 Project Goals

The primary goal of this report is to deeply review on RNN and how it can apply in natural language processing (NLP) tasks. And by comparing the performance of different models. this report aims to provide a detailed comparative analysis of the following RNN-based models:

- Vanilla RNN
- Long Short-Term Memory (LSTM)
- Gated Recurrent Unit (GRU)

3.2 Objectives

3.2.1 Implement Existing RNN Architectures

Develop models using state-of-the-art deep learning framework, PyTorch, to implement each of the above RNNs model. This includes setting up the network layers, loss functions, and appropriate optimization methods.

3.2.2 Apply the Models on a Benchmark NLP Dataset

Utilize a selected NLP dataset to train and evaluate the performance of each RNN-based model. The dataset will be preprocessed (e.g., tokenization, padding, embedding initialization) to suit sequential data modeling.

3.2.3 Compare Model Performances

Compare different RNN-based models on metrics relevant to the target NLP taskquantitatively and qualitatively. Accuracy, F1, training time scores be used.

3.2.4 Analyze the Impact of Architectural Differences

Identify and discuss the strengths and limitations of each model architecture through both experimental results and theoretical insights. For example, compare how LSTM and GRU architectures mitigate vanishing gradients and exploding gradients using gating mechanisms relative to a vanilla RNN.

4 Research Plan / Methodology

To achieve the project goals and objectives, this report will follow the step as below:

4.1 Literature Review

Since this report require some knowledge related on RNN and NLP. So it is necessary to review the foundemental of those area. Different architecture of RNNs-based model for instance, vanilla RNN, LSTM, GRU. And understand the limitation of RNN, vanishing/exploding gradients. Also, it is essential to understand natural language processing (NLP).

4.2 Data Collection and Preprocessing

In this report, ACL-IMDB (Maas et al., 2011) was experiment. This is a dataset for binary sentiment classification containing substantially more data than previous benchmark datasets. This dataset provide a set of 25,000 highly polar movie reviews for training, and 25,000 for testing. I preporcess the dataset by text cleaning and tokenization.

1. Lowercasing:
   All text was converted to lowercase to ensure consistency and avoid treadting words like "Good" and "good" as different tokens.

2. Handing Abbreviations: I hard code a dictionary in python, and marjority of common abbreviations were included and replaced with their full forms. For example:

   - "u" -> "you"
   - "pls" -> "please"

3. Removing HTML Tags: HTML tags were removed using regular expression

   ```
   import re
   re.sub(r'<.*?>', '', sentence)
   ```

   where variable "sentence" is movie review.

4. Removing Special Characters Non-alphanumeric characters (e.g. punctuation) were removed using the regular expression

```
import re
re.sub(r'[^a-zA-Z0-9\s]', '', sentence)
```

5. Removing Extra Whitespace Leading and trailing whitespaces were stripped.

6. Removing Step Words

4.3 Model Architecture

In this section, i will describe the model architecture, i will list out all of the model i have used and the detail information of those models. For mathematical notation, i will define all of the variables precisely and clearly. And because this project mainly focusing on natural language processing task. So that the classification model consists of an embedding layer, this is just like a look-up table to convert token to vector. First, the sequence of words $(w_1, w_2, \cdots, w_T)$ are passed through an embedding layer, which is the look-up table to convert those words in vector format. In other words, the look-up table map the word in a high-demensional space $(v_1, v_2, \cdots, v_T)$, where $v_t \in \mathbb{R}^d$. And the dimension of the vector space is depends on how we define the hidden size of the model. Next, the forward of RNN based model will processes these word vectors v_t in the forward directions, updating corresponding hidden states at each time step t.

4.3.1 Model 1 (Vanilla RNN)

This is the vanilla RNN model, with 1 layer. We just use the last output to calculate the loss function and do the backpropagation.

$$h_t = f(W_{ih}^T v_t + W_{hh} h_{t-1} + b_h) \tag{62}$$

where W_{ih} is input to hidden matrix, v_t is the word vector at time step t, W_{hh} is hidden to hidden matrix. h_t is hidden state at time t. b_h is biased term for hidden state. For the output at time t, we will pass the hidden state output to an activation function defined as below.

$$\hat{y}_t = f(W_{ho}^T h_t + b_y) \tag{63}$$

where y_t is the output at time step t and W_{ho} is the hidden to output matrix and b_y is biased term for output. And $f(\cdot)$ is an activation function. Then we calculate the loss function by the last output.

$$Loss = \mathcal{L}(\hat{y}, y) \tag{64}$$

4.3.2 Model 2 (Long short-term memory)

For the LSTM model, I use 1 layer, and the calculation step was already discuss.

4.3.3 Model 3 (Gated recurrent unit)

For the GRU model, I use 1 layer, and the calculation step was already discuss.

4.4 Training

4.4.1 Training Protocol

In order to ensure a fair comparison, i will use the similar training configurations (e.g., optimizer choice, learning rate schedule, batch size).
4. **Training, Hyperparameter Tuning, and Evaluation:** - **Training Protocol:** Use similar training configurations (e.g., optimizer choice, learning rate schedule, batch size) for all models to ensure a fair comparison. - **Hyperparameter Optimization:** Conduct grid search or Bayesian optimization for parameters such as hidden unit size, dropout rates, and learning rates. - **Evaluation Metrics:** Define metrics based on the nature of the task (e.g., classification accuracy, F1 score, perplexity). Monitor convergence behavior and computational resource usage. - **Cross-Validation and Benchmarking:** Use cross-validation and hold-out test sets to ensure robustness of performance assessments.

To achieve the project goals and objectives, the research will follow a structured approach combining theoretical understanding with empirical evaluation:
1. **Literature Review:** - **Background Study:** Conduct a detailed review of seminal and recent works on RNNs, LSTMs, GRUs, and RNN encoder-decoder architectures in NLP. Key references include: - Hochreiter & Schmidhuber (1997) on LSTM. - Cho et al. (2014) and Chung et al. (2014) on GRU. - Recent surveys on the application of RNNs in NLP. - **Problem Context:** Understand current best practices, challenges (e.g., vanishing/exploding gradients, sequence length handling), and evaluation standards in NLP.
2. **Data Collection and Preprocessing:** - **Dataset Selection:** Choose a benchmark NLP dataset appropriate to the target task (e.g., language modeling, text classification, or machine translation). - **Preprocessing Steps:** Perform text normalization, tokenization, vocabulary building, and sequence padding. Optional steps may include initializing word embeddings (either pre-trained or learned from scratch).
3. **Model Development and Experimental Implementation:** - **Model Architecture Design:** Implement the different RNN-based models: - **Vanilla RNN:** Define the recurrence as

$$h_t = \sigma(W_{hx}x_t + W_{hh}h_{t-1} + b_h), \quad y_t = f(W_{yh}h_t + b_y),$$

where x_t is the input vector at time t, h_t is the hidden state, and y_t is the output. - **LSTM:** Set up the LSTM cell with input, forget, and output gates to control information flow. - **GRU:** Implement the GRU cell, which uses update and reset gates as a simplified alternative. - **RNN Encoder-Decoder:** Develop an encoder to summarize the input sequence and a decoder to generate the target sequence. - **Framework and Tools:** Develop the models using PyTorch or TensorFlow. Leverage available libraries for reproducibility and efficiency.
4. **Training, Hyperparameter Tuning, and Evaluation:** - **Training Protocol:** Use similar training configurations (e.g., optimizer choice, learning rate schedule, batch size) for all models to ensure a fair comparison. - **Hyperparameter Optimization:** Conduct grid search or Bayesian optimization for parameters such as hidden unit size, dropout rates, and learning rates. - **Evaluation Metrics:** Define metrics based on the nature of the task (e.g., classification accuracy, F1 score, perplexity). Monitor convergence behavior and computational resource usage. - **Cross-Validation and Benchmarking:** Use cross-validation and hold-out test sets to ensure robustness of performance assessments.
5. **Result Analysis and Reporting:** - **Comparative Analysis:** Analyze and interpret the performance differences among the RNN, LSTM, GRU, and RNN encoder-decoder models. Assess the trade-offs in terms of processing speed, accuracy, and stability. - **Discussion of Findings:** Provide a discussion relative to theoretical expectations (e.g., how and why LSTMs may outperform vanilla RNNs in handling long-term dependencies). - **Documentation:** Compile the findings into a detailed technical report, including visualizations of loss curves, performance metrics over training epochs, and error analysis.
6. **Dissemination:** - **Presentation and Peer Review:** Present the results at seminars and workshops to receive feedback. - **Final Reporting:** Prepare a final research report and, if applicable, a manuscript for publication in a relevant conference or journal.

References

Bengio, Y. (2009). Learning deep architectures for ai. *Found. Trends Mach. Learn.*, *2*(1), 1–127. https://doi.org/10.1561/2200000006

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, *5*(2), 157–166. https://doi.org/10.1109/72.279181

Cho, K., van Merrienboer, B., Bahdanau, D., & Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint*. https://doi.org/10.48550/arxiv.1409.1259

Delalleau, O., & Bengio, Y. (2011). Shallow vs. deep sum-product networks. *In NIPS*.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, *9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Le Roux, N., & Bengio, Y. (2010). Deep belief networks are compact universal approximators. *FouNeural Computation*, *22*(8), 2192–2207. https://doi.org/10.1162/neco.2010.08-09-1081

Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., & Potts, C. (2011). Learning word vectors for sentiment analysis. *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, 142–150. http://www.aclweb.org/anthology/P11-1015

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint*. https://doi.org/10.48550/arxiv.1301.3781

Pascanu, R., Montufar, G., & Bengio, Y. (2013). On the number of response regions of deep feed forward networks with piece-wise linear activations. *In NIPS*. https://doi.org/10.48550/arxiv.1312.6098

Rabiner, L., & Juang, B. (1986). An introduction to hidden markov models. *IEEE ASSP Magazine*, *1*(3), 4–16. https://doi.org/10.1109/MASSP.1986.1165342

Rabiner, L. R. (1989). A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, *77*(2), 257–286. https://doi.org/10.1109/5.18626

Rabiner, L. R., & Juang, B. H. (1993). *Fundamentals of speech recognition*. PTR Prentice Hall.

Roberts, G. O., & Rosenthal, J. S. (2004). General state space markov chains and mcmc algorithms. *Probability Surveys*, *1*, 20–71. https://doi.org/10.1214/154957804100000024

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature (London)*, *323*(6088), 533–536. https://doi.org/10.1038/323533a0

Sengupta, A., Das, A., & Guler, S. I. (2023). Hybrid hidden markov lstm for short-term traffic flow prediction. *arXiv preprint*. https://doi.org/10.48550/arxiv.2307.04954

Werbos, P. J. (1990). Backpropagation through time: What it does and how to do it. *Proceedings of the IEEE*, *78*(10), 1550–1560. https://doi.org/10.1109/5.58337

Wikipedia. (2023). Backpropagation through time [In Wikipedia, The Free Encyclopedia. Retrieved November 12, 2024, from https://en.wikipedia.org/wiki/Backpropagation_through_time].