Recurrent Deep Learning Models and its applications

Ngai Ho Wang

March 30, 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.

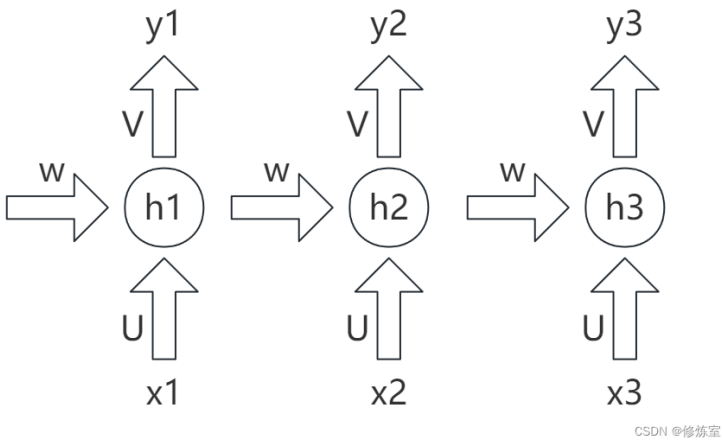


Figure 1: Unfolded RNN

Forward Pass

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

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

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

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

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

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

Notation	Meaning	Dimension
U	Weight matrix for input to hidden state	$input\ size \times hidden\ units$
W	Weight matrix for hidden to hidden state	$hidden\ units \times hidden\ units$
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

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.

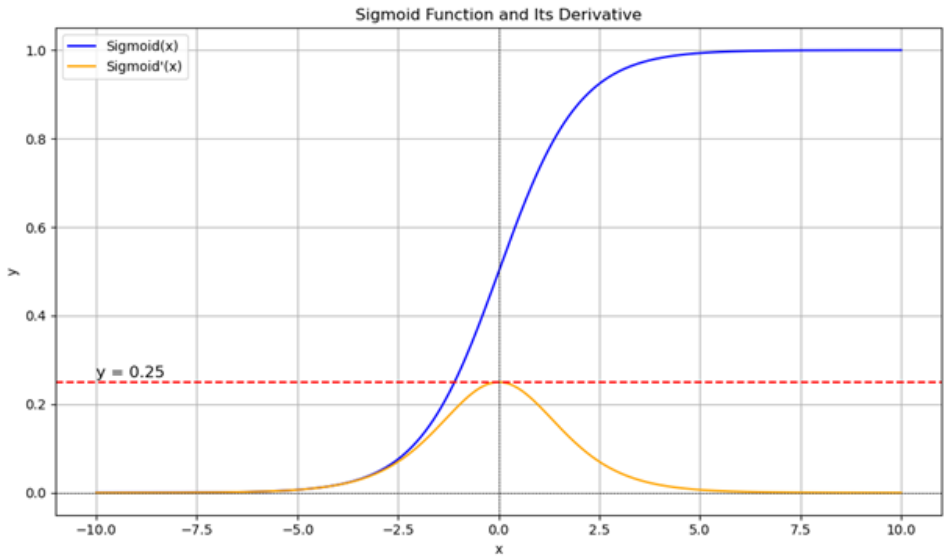
Sigmoid activation function

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.



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

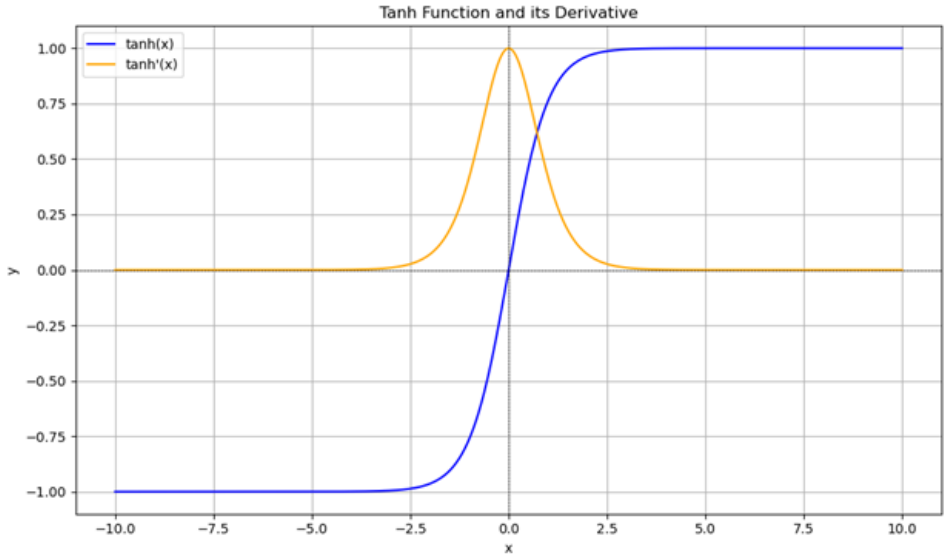
Hyperbolic tangent activation function

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).

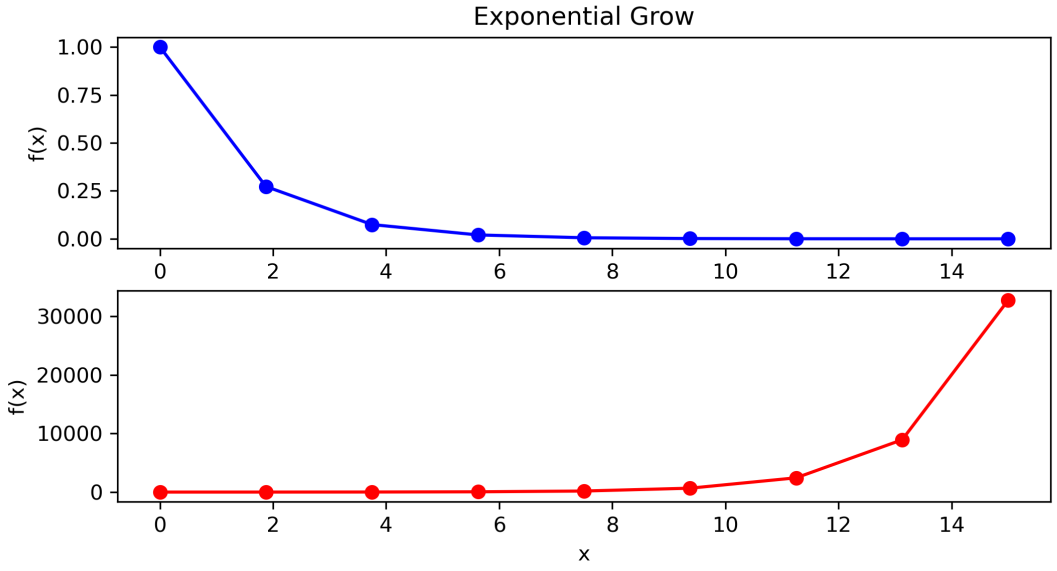
Vanishing Gradients

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.



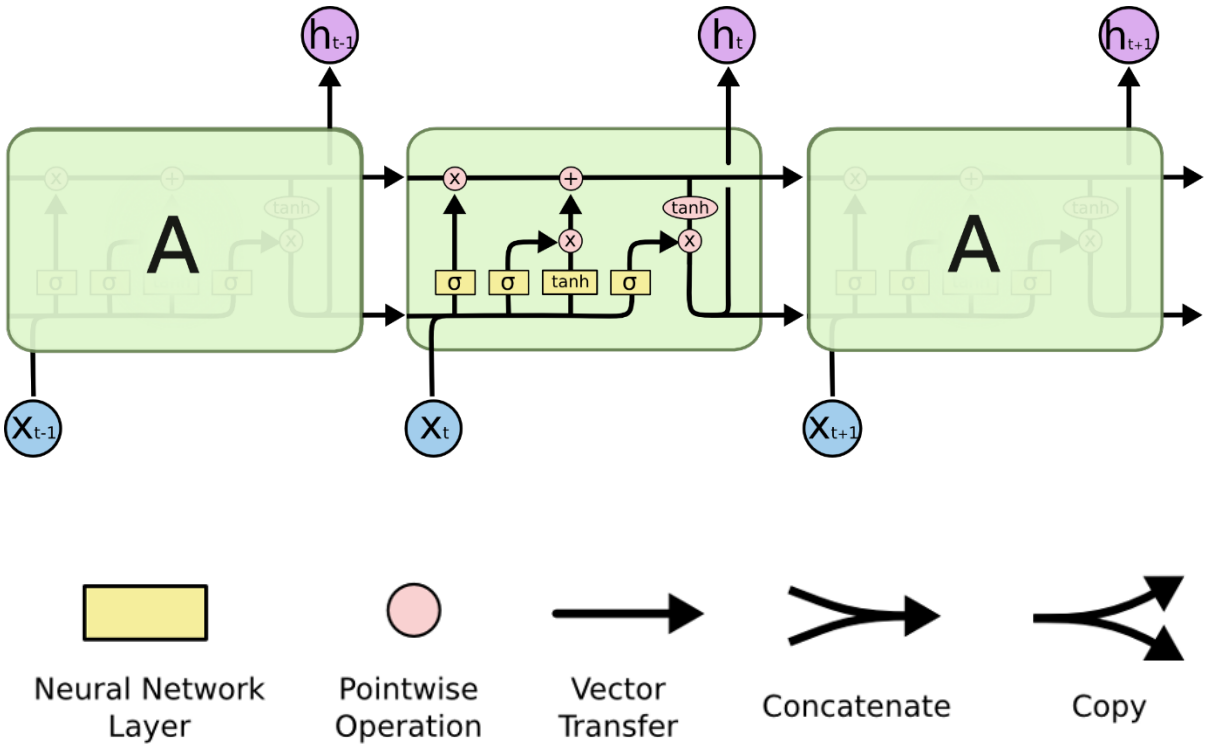
2.2.4 Long short-term memory (LSTM)

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:

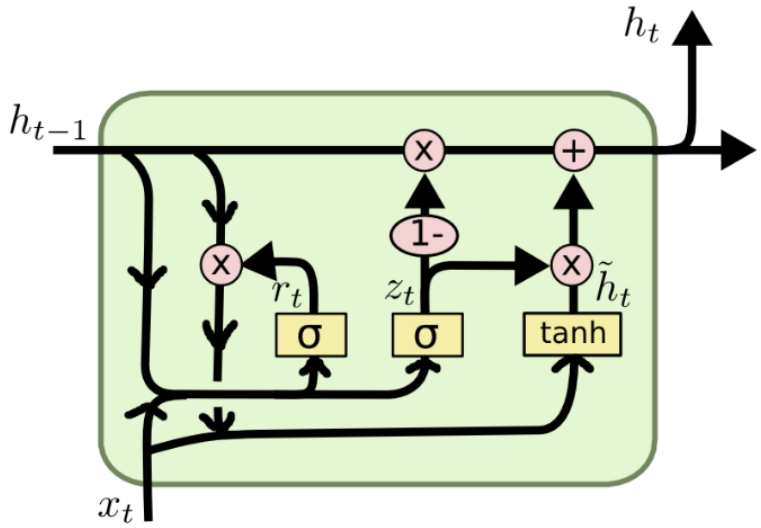
1. Weights matrix for the input
   - Forget gate: $n \times m$
   - Input gate: $n \times m$
   - Cell gate: $n \times m$
   - Output gate: $n \times m$

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:

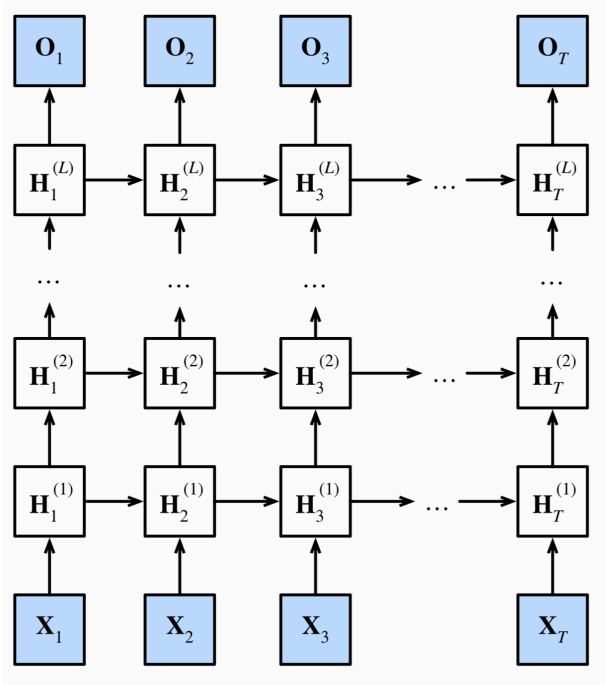
1. Weights matrix for the input
   - Update gate: $n \times m$
   - Reset gate: $n \times m$
   - Candidate hidden state: $n \times m$
2. Weight matrix for the hidden state
   - Hidden state for update gate: $m \times m$
   - Hidden state for reset gate: $m \times m$
   - Hidden state candidate hidden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



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

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

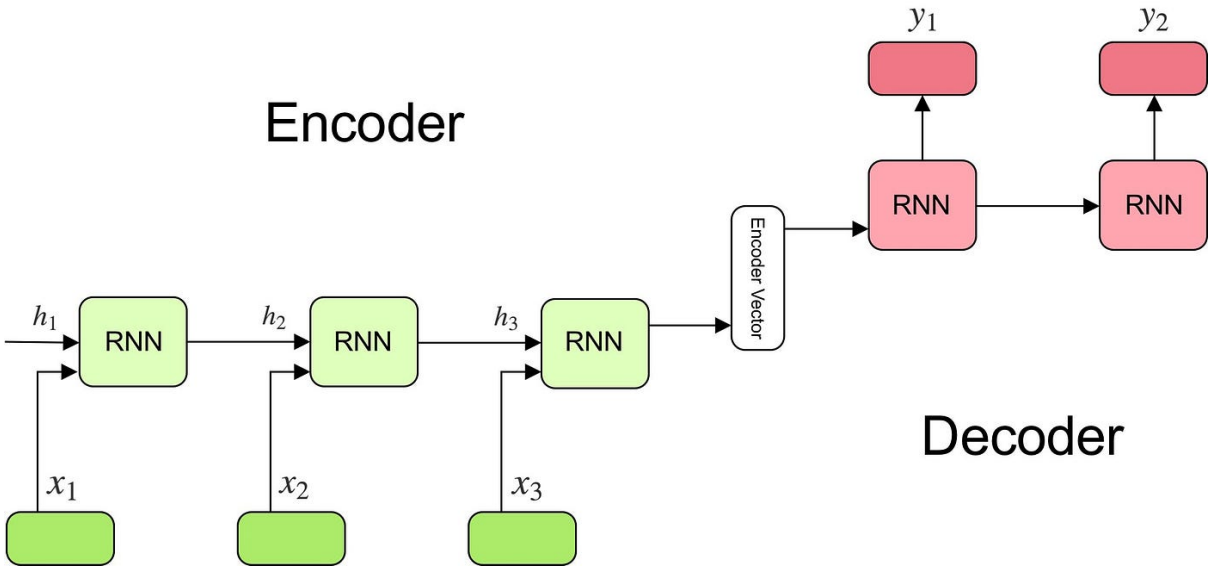
$$h_t^{(L)} = f(W_{xh}^{(L)} h_t^{(L-1)} + W_{hh}^{(L)} h_{t-1}^{(L)} + b_h^{(L)}) \tag{40}$$

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

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

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 RNN encoder decoder model



RNN Encoder-Decoderis a RNN based architecture which commonly used for handle sequence to sequence problem, for example machine translation, summarization, and question answering. The designed of this architecture is good for handle the input sequence and the output sequence is not the same length, leveraging the sequential nature of data.

In this architecture, there are two main components, the first component is rnn encoder, the usage of the encoder is to encode the input sequence into a fixed length vector. And the decoder is to decode the vector into the target sequence.

Encoder

The encoder is a RNN based block, which processes the input sequence step by step and calculate the hidden state representation for each input token. Let the input sequence be $X = [x_1, x_2, \cdots, x_T]$ where T is the length of the input sequence, and x_t is the input at time step t. We use the below formula to computes the hidden states h_t.

For the Vanilla RNN encoder

$$h_t^{enc} = tanh(W_{ih}^{enc} x_t + W_{hh}^{enc} h_{t-1} + b_h^{enc}) \tag{43}$$

Notation	Meaning	Dimension
W_{ih}^{enc}	Weight matrix for input to hidden state in encoder	*hidden units* × *input size*
W_{hh}^{enc}	Weight matrix for hidden to hidden state in encoder	*hidden units* × *hidden units*
b_h^{enc}	Bias vector for the hidden state in encoder	*hidden units* × *hidden units*
h_t^{enc}	Hidden state at timestep t in encoder	*hidden units* × *hidden units*

Table 4: Encoder

After the model processed the entire input sequence from $t = 1$ to $t = T$, the last hidden state h_T^{enc} treat as the context vector c.

$$c = h_T^{enc} \tag{44}$$

Decoder

The decoder is the second component of RNN encoder decoder, the usage of the decoder is to generates the output sequence step by step, conditioned on the context vector c and its previous hidden state. At each timestep t, the decoder updates its hidden state s'_t based on the previous hidden state $s_{t'-1}$ and the embedding of the previous output $y_{t'-1}$. And we will define the $s_0 = c$ first.

$$h_t^{dec} = tanh(W_{ih}^{dec} y_{t-1} + W_{hh}^{dec} + b_h^{dec}) \tag{45}$$

After the model computed the h_t^{dec}, the model will compute the output probability distribution.

$$P(y_t|y_{t-1}, c) = softmax(W_o h_t^{dec} + b_o^{dec}) \tag{46}$$

And the predicted token $y_t = argmax P(y_t)$, the process will repeat until the decoder predicts the end of sequence token ($< EOS >$).

Notation	Meaning	Dimension
W_{ih}^{dec}	Weight matrix for input to hidden	$hidden\ units \times input\ size$
W_{hh}^{dec}	Weight matrix for hidden to hidden	$hidden\ units \times hidden\ units$
b_h^{dec}	Bias vector for the hidden	$hidden\ units \times hidden\ units$
h_t^{dec}	Hidden state at timestep t	$hidden\ units \times hidden\ units$
b_o^{dec}	Bias vector for the output layer	$hidden\ units \times hidden\ units$
W_o^{dec}	Weight matrix for hidden to output	$output\ feature\ space \times hidden\ units$

Table 5: Decoder

2.2.8 Attention Mechanism

Bahdanau (Additive) Attention

This method proposed by (Bahdanau et al., 2016), in which this attention uses a feed-forward network to compute the attention scores.

$$e_{t',t} = v_a^T tanh(W_a h_{t'-1}^{dec} + U_a h_t^{enc} + b_a) \tag{47}$$

Luong (Multiplicative) Attention

This method is introduced by Luong et al (Luong et al., 2015). This method uses the dot product to compute the attention scores.

$$e_{t',t} = h_{t'-1}^{dec} \cdot h_t^{enc} \tag{48}$$

Scaled Dot-Product Attention

The scaled dot-product attention was introduced by (Vaswani et al., 2017), this is the key component in Transformer architecture. It improves from the Luong's dot-product attention (Luong et al., 2015) by addressing he numerical stability issues when working with the high dimensional vectors and making attention computation more efficient for parallelization. Below is the formula of attention scores.

$$S = \frac{QK^T}{\sqrt{d_k}} \tag{49}$$

Where $S \in \mathbb{R}^{n \times n}$ is pairwise similarity scores between all tokens.

2.2.9 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, which describe 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:

$$S = \{s_1, s_2, \ldots, s_N\}. \tag{50}$$

$$P(s_{t+1} \mid s_t, s_{t-1}, s_{t-2}, \cdots, s_1) = P(s_{t+1} \mid s_t) \tag{51}$$

The state transition probability is denoted by:

$$P_{ij} = P(s_{t+1} = s_j \mid s_t = s_i). \tag{52}$$

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:

$$P(s_1, s_2, \ldots, s_T) = P(s_1) \cdot P(s_2 \mid s_1) \cdot P(s_3 \mid s_2) \cdots P(s_T \mid s_{T-1}) \tag{53}$$

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

$$P(s_1, s_2, \ldots, s_T) = P(s_1) \prod_{t=1}^{T-1} P(s_{t+1} \mid s_t) \tag{54}$$

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:

$$\pi_t(s_j) = P(s_t = s_j) \tag{55}$$

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.

$$\pi_{t+1}(s_j) = \sum_{s_i \in S} \pi_t(s_i) \cdot P_{ij} \tag{56}$$

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

$$\pi_j = \sum_{i=1}^{n} \pi_i P_{ij} \tag{57}$$

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:

$$\mathcal{X} = \{s_1, s_2, \cdots, s_n\} \tag{58}$$

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:

$$\mathcal{Y} = \{o_1, o_2, \cdots, o_m\} \tag{59}$$

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:

$$P_{n\times n} = \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1n} \\ p_{21} & p_{22} & \cdots & p_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ p_{n1} & p_{n2} & \cdots & p_{nn} \end{bmatrix} \tag{60}$$

$$p_{ij} = P(X_{t+1} = s_j \mid X_t = s_i) \tag{61}$$

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

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

$$B_{n\times n} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mm} \end{bmatrix} \tag{62}$$

$$b_{ik} = P(Y_t = o_k \mid X_t = s_i) \tag{63}$$

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 $P(X_1)$.

$$P(X, Y) = P(X_1) \cdot \prod_{t=1}^{T-1} P(X_{t+1} = s_j \mid X_t = s_i) \cdot \prod_{t=1}^{T} P(Y_t = o_k \mid X_t = s_i) \tag{64}$$

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).

$$p(x_t|z = q_j) = \sum_{l=1}^{k} c_{jl}\mathcal{N}(x_t|\mu_{jl}, \Sigma_{jl}) \tag{65}$$

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:

$$p(X) = p(z_1)\prod_{t=1}^{T-1} p(z_{t+1}|z_t)\prod_{t=1}^{T} p(x_t|z_t) \tag{66}$$

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.

2.2.10 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

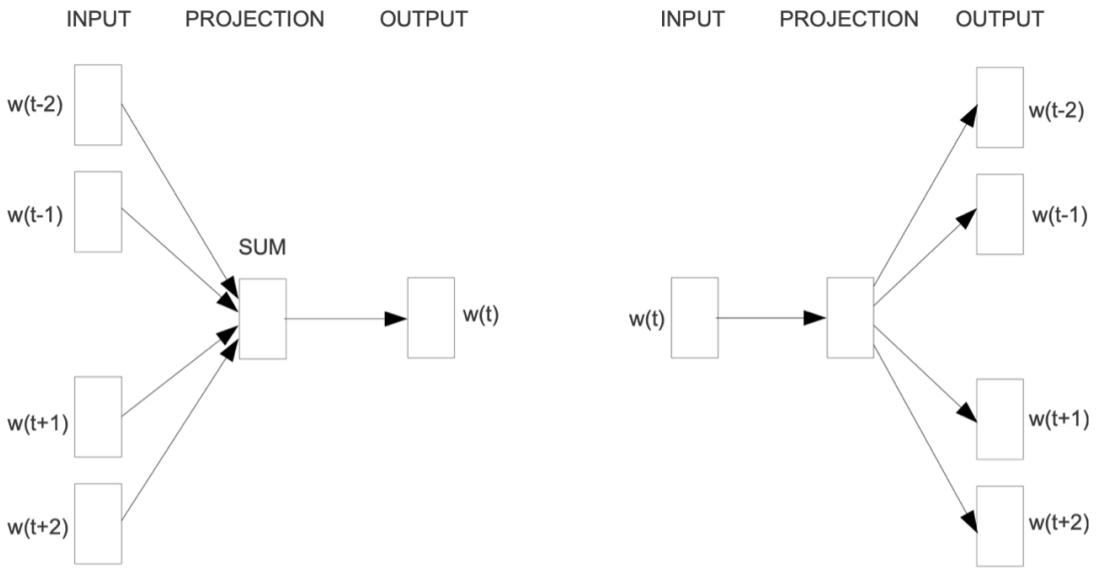
CBOW operates by taking context words around a target word, it is flexible to adjust the window size which mean that we can control how many words around our target word. Hence aggregating their embeddings and passing to hidden layer to produce a probability distribution. Capture the semantic meanings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

$$Q = N \times D + D \times log_2(V) \tag{67}$$

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.

$$Q = C \times (D + D \times log_2(V)) \tag{68}$$



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 by evaluating their performance on the IMDb dataset.

- Vanilla RNN
- Long Short-Term Memory (LSTM)
- Gated Recurrent Unit (GRU)
- RNN Encoder-Decoder
- RNN Encoder-Decoder with Attention Mechanism

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. For example accuracy.

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

4.2.1 Data Collection

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

4.2.2 Data Preprocessing

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 Stop Words
   Stop words (e.g., "is", "the", "and") were removed to reduce noise in the text. The NLTK library's stop word list was used. Words not in the stop word list were retained.

7. Tokenization
   The cleaned text was split into individual tokens, I use white space and the criterion to split the word.

   Once the text was preprocessed, a vocabulary was constructed from the dataset. The vocabulary mapping each word to a unique index. Words not found in the vocabulary are replaced with an "unknown" token. I reserved index 0 as unknown token. So that if the word can not map to correspoinding index. Then the token should be 0. This is a good way to handle Out-of-vocabulary words (Mikolov et al., 2013).

Word	Index
<UNK>	0
acting	1
good	2
movie	3
amazing	4
performance	5
bad	6
terrible	7
excellent	8

Table 6: Word to Index Mapping Table

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

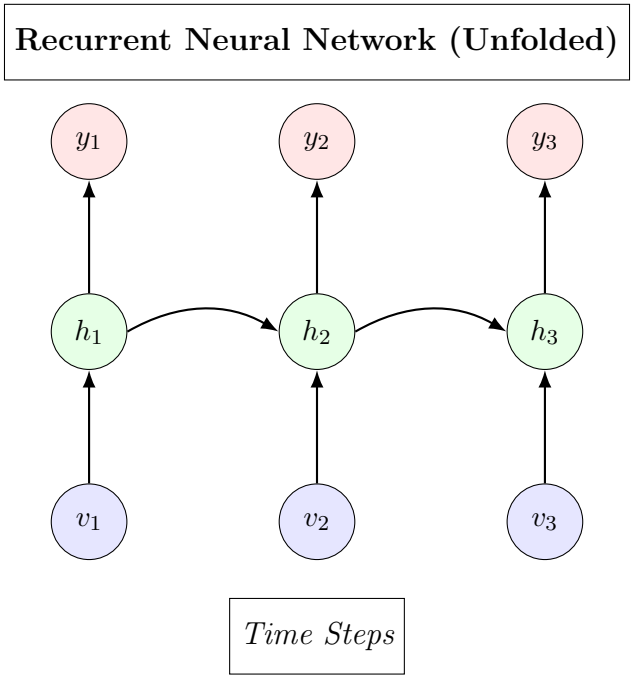
of an embedding layer, this is just like a look-up table to convert token to vector. First, the sequence of words $(w_1, w_2, \cdots, w_T)$ are passed through an embedding layer, which is the look-up table to convert those words in vector format. In other words, the look-up table map the word in a high-demensional space $(v_1, v_2, \cdots, v_T)$ (Table 5), 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.

Word	**Index**	**Embedding Vector**
<UNK>	0	$[0.01 \quad -0.02 \cdots 0.03 \quad 0.04]$
acting	1	$[0.12 \quad 0.23 \cdots -0.11 \quad 0.05]$
good	2	$[0.03 \quad -0.15 \cdots 0.08 \quad 0.07]$
movie	3	$[0.10 \quad 0.05 \cdots -0.09 \quad -0.01]$
amazing	4	$[0.25 \quad -0.12 \cdots 0.18 \quad 0.30]$
performance	5	$[-0.10 \quad -0.20 \cdots 0.15 \quad -0.05]$
bad	6	$[0.40 \quad 0.35 \cdots -0.10 \quad 0.50]$
terrible	7	$[0.30 \quad 0.25 \cdots -0.11 \quad 0.40]$
excellent	8	$[0.32 \quad 0.38 \cdots -0.90 \quad 0.70]$

Table 7: Word to Index Matching with Lookup Table Embeddings

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.



$$h_t = f(W_{ih}^T v_t + W_{hh} h_{t-1} + b_h) \tag{69}$$

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.

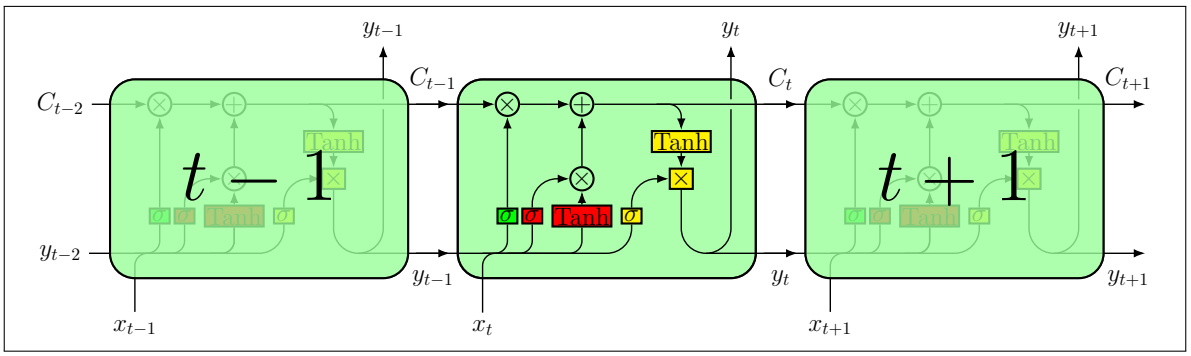
$$\hat{y}_t = f(W_{ho}^T h_t + b_y) \tag{70}$$

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.

$$Loss = \mathcal{L}(\hat{y}, y) \tag{71}$$

4.3.2 Model 2 (Long short-term memory)

For the LSTM model, I use 1 layer:



For an LSTM, the overall structure is similar to the Vanilla RNN shown in the diagram, but the hidden state computation is replaced by the LSTM's more complex gating mechanism. Using its gating mechanisms to control how information flows through time. Let us consider an input sequence $\{v_1, v_2, \ldots, v_T\}$, where v_t represents the input vector at time step t. Here's how the LSTM processes the input:

Step 1: Initialize Hidden and Cell States At the beginning of the sequence, the LSTM initializes the hidden state h_0 and cell state C_0. For our model, we use 0 matrix as h_0 and C_0:

$$h_0 = \mathbf{0}, \quad C_0 = \mathbf{0} \tag{72}$$

Step 2: Process Each Input Vector For each time step t, LSTM processes the current input v_t along with the previous hidden state h_{t-1} and cell state C_{t-1} to compute:

- The **forget gate** f_t, determine which information should be retained in memory cell C_{t-1}.
- The **input gate** i_t, controls how much new information from the current time step is allowed to enter the cell.
- The **cell candidate** $\tilde{C}_t$, represents the new information to be added to the cell state.
- The updated **cell state** C_t, which combines the retained information from C_{t-1} and the new candidate $\tilde{C}_t$.

- The **output gate** o_t, control how much or what information from the cell state should be passed to the next layer or used in predictions.
- The updated **hidden state** h_t, summarizes the information at time t.

Step 3: Compute Forget Gate The forget gate f_t is computed as:

$$f_t = \sigma(W_f v_t + U_f h_{t-1} + b_f) \tag{73}$$

where:

- σ is the sigmoid activation function.
- W_f is the input to hidden weight matrix for the forget gate.
- U_f is the hidden to hidden weight matrix for the forget gate.
- b_f is the bias term for the forget gate.

The forget gate determines which parts of the previous cell state C_{t-1} to retain.

Step 4: Compute Input Gate The input gate i_t is computed as:

$$i_t = \sigma(W_i v_t + U_i h_{t-1} + b_i) \tag{74}$$

where:

- W_i, U_i, and b_i are the weight matrices and bias for the input gate.

The input gate determines how much new information to add to the cell state.

Step 5: Compute Cell Candidate The cell candidate $\tilde{C}_t$ is computed as:

$$\tilde{C}_t = \tanh(W_c v_t + U_c h_{t-1} + b_c) \tag{75}$$

where:

- tanh is the hyperbolic tangent activation function.
- W_c, U_c, and b_c are the weight matrices and bias for the cell candidate.

The cell candidate represents potential new information to add to the cell state.

Step 6: Update Cell State The cell state C_t is updated as:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{76}$$

where $\odot$ represents element-wise multiplication. The updated cell state combines retained information from C_{t-1} (via f_t) and new information from $\tilde{C}_t$ (via i_t).

Step 7: Compute Output Gate The output gate o_t is computed as:

$$o_t = \sigma(W_o v_t + U_o h_{t-1} + b_o) \tag{77}$$

where W_o, U_o, and b_o are the weight matrices and bias for the output gate.

Step 8: Compute Hidden State The hidden state h_t is computed as:

$$h_t = o_t \odot \tanh(C_t) \tag{78}$$

The hidden state summarizes the information at time t and is passed to the next time step.

Step 9: Use Final Hidden State for Output After processing all time steps, the final hidden state h_T is used as the input to a fully connected layer for classification:

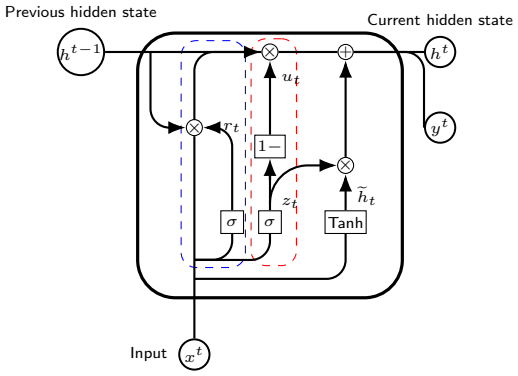
$$\hat{y} = f(W_{ho}^T h_T + b_y) \tag{79}$$

where W_{ho} and b_y are the weight matrix and bias for the output layer, and f is the activation function. Then we calculate the loss function by the last output.

$$Loss = \mathcal{L}(\hat{y}, y) \tag{80}$$

4.3.3 Model 3 (Gated recurrent unit)

For the GRU model, I use 1 layer. This architecture is similar with LSTM but reduce the among of parameters. Let us consider an input sequence $\{v_1, v_2, \ldots, v_T\}$, where v_t represents the input vector at time step t. Here's how the GRU processes the input:



Step 1: Initialize Hidden State At the beginning of the sequence, the GRU initializes the hidden state h_0 to zero:

$$h_0 = \mathbf{0} \tag{81}$$

Step 2: Process Each Input Vector For each time step t, the GRU processes the current input v_t along with the previous hidden state h_{t-1} to compute:

- The **reset gate** r_t, which determines how much of the previous hidden state h_{t-1} to forget.
- The **update gate** z_t, which determines how much of the previous hidden state h_{t-1} to retain.
- The **candidate hidden state** $\tilde{h}_t$, which represents the new information to be combined with the previous hidden state.
- The updated **hidden state** h_t, which summarizes the information at time t.

Step 3: Compute Reset Gate The reset gate r_t is computed as:

$$r_t = \sigma(W_r v_t + U_r h_{t-1} + b_r) \tag{82}$$

where:

- σ is the sigmoid activation function.
- W_r is the input to hidden weight matrix for the reset gate.
- U_r is the hidden to hidden weight matrix for the reset gate.
- b_r is the bias term for the reset gate.

The reset gate determines how much of the previous hidden state h_{t-1} to use when computing the candidate hidden state.

Step 4: Compute Update Gate The update gate z_t is computed as:

$$z_t = \sigma(W_z v_t + U_z h_{t-1} + b_z) \tag{83}$$

where:

- W_z is the input to hidden weight matrix for the update gate.
- U_z is the hidden to hidden weight matrix for the update gate.
- b_z is the bias term for the update gate.

The update gate determines how much of the previous hidden state h_{t-1} to retain and how much of the new candidate hidden state to use.

Step 5: Compute Candidate Hidden State The candidate hidden state $\tilde{h}_t$ is computed as:

$$\tilde{h}_t = \tanh(W_h v_t + U_h(r_t \odot h_{t-1}) + b_h) \tag{84}$$

where:

- tanh is the hyperbolic tangent activation function.
- $r_t \odot h_{t-1}$: The reset gate modulates the contribution of the previous hidden state h_{t-1}.
- W_h, U_h, and b_h are the weight matrices and bias for the candidate hidden state.

Step 6: Update Hidden State The updated hidden state h_t is computed as:

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t \tag{85}$$

where:

- z_t: The update gate decides how much of the previous hidden state h_{t-1} to retain.
- $(1 - z_t)$: The complement of the update gate decides how much of the candidate hidden state $\tilde{h}_t$ to use.

Step 7: Use Final Hidden State for Output After processing all time steps, the final hidden state h_T is used as the input to a fully connected layer for classification:

$$\hat{y} = f(W_{ho}^T h_T + b_y) \tag{86}$$

where:

- W_{ho} and b_y are the weight matrix and bias for the output layer.
- f is the activation function (e.g., sigmoid for binary classification).

Then we calculate the loss function by the last output.

$$Loss = \mathcal{L}(\hat{y}, y) \tag{87}$$

4.3.4 Model 4 (Vanilla RNN with 2 layer)

Same setting with model 1, but I stack 2 RNN block. In the other word, this is deep recurrent neural networks.

4.3.5 Model 5 (Gated recurrent unit with 2 layer)

Same setting with model 1, but I stack 2 GRU block. In the other word, this is deep recurrent neural networks.

4.3.6 Model 6 (Long short-term memory with 2 layer)

Same setting with model 1, but I stack 2 LSTM block. In the other word, this is deep recurrent neural networks.

4.3.7 Model 7 (RNN Encoder-Decoder)

For the RNN encoder-decoder architecture, I will explain how to data be processes and I will show the mathmatical transformations for each time step. We first assume the input $X = [x_1, x_2, \cdots, x_T]$ be the input sequence and the T is the sequence length. x is word index in the vocabulary.

Then the embedding layer maps each word index x_t to a high dimensional space.

$$e_t = Embedding(x_t) \tag{88}$$

For each time step, the model will train the parameters h_t by using the current input and the previous hidden state.

$$h_t = tanh(W_{ih} e_t + W_{hh} h_{t-1} + b_h) \tag{89}$$

Where $W_{ih} \in \mathbb{R}^{hidden\ dim \times embedding\ dim}$, $W_{hh} \in \mathbb{R}^{hidden\ dim \times hidden\ dim}$. $b_h \in \mathbb{R}^{hidden\ dim}$.

For the decoder part is do the fully connected layer.

$$z = W_o h_T + b_o \tag{90}$$

And the apply sigmoid activation function to find the highest probability output.

$$\hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}} \tag{91}$$

4.3.8 Model 8 (RNN Encoder-Decoder with attention mechanism)

4.4 Training, Hyperparameter Tuning, and Evaluation

In my experiments, I implemented the training loop using PyTorch. The following details describe the key components of my training code, which are designed to ensure robust model optimization and fair evaluation:

4.4.1 Model Hyperparameters and Setup

The following hyperparameters were chosen based on preliminary experiments and standard practices for sentiment classification tasks:

For model 1 (Vanilla RNN), model 2 (Long short-term memory), model 3 (Gated recurrent unit), model 7 (RNN encoder-decoder), model 8 (RNN encoder-decoder with attention mechanism).

- **Architecture Hyperparameters:**
  - Number of layers: `no_layers = 1`
  - Vocabulary size: `vocab_size` = `len`(`vocab`) + `1` (the extra one is for unknown word)
  - Embedding dimension: `embedding_dim = 100`
  - Hidden dimension: `hidden_dim = 50`
  - Output dimension: `output_dim = 1`
  - Dropout probability: `drop_prob = 0.5`
  - Batch size: `batch_size = 50`
- **Optimization Hyperparameters:**
  - Learning rate: `lr = 0.001`
  - Loss Function: Binary Cross Entropy Loss
  - Optimizer: Adam with predefined learning rate.

For model 4 (Vanilla RNN with 2 layer), model (GRU with 2 layer), model 6 (Long short-term memory with 2 layer)

- **Architecture Hyperparameters:**
  - Number of layers: `no_layers = 2`
  - Vocabulary size: `vocab_size` = `len`(`vocab`) + `1` (the extra one is for unknown word)
  - Embedding dimension: `embedding_dim = 100`
  - Hidden dimension: `hidden_dim = 50`
  - Output dimension: `output_dim = 1`
  - Dropout probability: `drop_prob = 0.5`
  - Batch size: `batch_size = 50`
- **Optimization Hyperparameters:**
  - Learning rate: `lr = 0.001`
  - Loss Function: Binary Cross Entropy Loss
  - Optimizer: Adam with predefined learning rate.

4.4.2 Training Loop Overview

- **Mini-Batch Training with Hidden State Initialization:** At the start of each epoch, the model is set to training mode using `model.train()`. For each mini-batch from the training data:

  1. The model's hidden state is initialized with zero matrix.
  2. Before the forward pass, gradients are zeroed via `model.zero_grad()` and `optimizer.zero_grad()`.
  3. The forward pass is executed, producing outputs and an updated hidden state.
  4. The hidden state is detached from the computation graph using `h.detach()` to avoid backpropagating through all previous batches.
  5. The loss is computed using binary cross entropy.
  6. Backpropagation is performed via `loss.backward()`.
  7. The gradients are clipped using `nn.utils.clip_grad_norm_` with the specified clip value to mitigate exploding gradients.
  8. Finally, the optimizer updates the model parameters using `optimizer.step()`.

- **Cross Validation:** In this project, a stratified k-fold cross validation was applied as part of the training loop to evaluate and compare the performance of various RNN-based model. And I will use the better performance of the best combination which has the highest mean accuracy. And this method was designed to ensure that each model was trained and validated on all data splits, providing a more comprehencive evaluation of model performance.

  First the IMDB dataset was split into 5 folds (k=5) for stratified cross-validation, ensuring that each fold maintained the same proportion of positive and negative sentiment labels. For each fold, there is only one subset was used as the validation set, in the mean time, the rest of the subsets were combined to form as the training set.

  For each fold,the model will train on the trainin set using the predefined batch size and validated on the validation set after each epoch. Once all the folds were completed, the performance metris were averaged to obtain the final evaluation scores for each model. In the other word, the average accuracy were used to compare the models and determine their overall effectiveness.

- **Validation Stage:** After each epoch, the model is switched to evaluation mode using `model.eval()` and evaluated on the validation set. Similar to training, the hidden state is updated parameters and the model will use that parameters to make the prediction. And the accuracy is computed and recorded.

- **Performance Monitoring and Checkpointing:** At the end of each epoch, the mean training and validation losses, as well as accuracy, are calculated. The validation loss is compared against the best loss seen so far. If the current validation loss is lower than the lowerest one, then the model's state dictionary is saved as the best model in .pth format.

4.4.3 Pseudocode Description

The following Python pseudocode summarizes the training procedure implemented in my experiment:

Listing 1: Training Loop Pseudocode

```
clip = 5
epochs = 5
valid_loss_min = np.inf
epoch_tr_loss, epoch_vl_loss = [], []
epoch_tr_acc, epoch_vl_acc = [], []

for epoch in range(epochs):
    train_losses = []
    train_acc = 0.0
    model.train()
    # Initialize the hidden state for the first mini-batch
    h = model.init_hidden(batch_size)
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        model.zero_grad()
        optimizer.zero_grad()

        output, h = model(inputs, h)
        h = h.detach()

        loss = criterion(output, labels.float())
        loss.backward()
        train_losses.append(loss.item())

        accuracy = acc(output, labels)
        train_acc += accuracy

        nn.utils.clip_grad_norm_(model.parameters(), clip)
        optimizer.step()

    val_h = model.init_hidden(batch_size)
    val_losses = []
    val_acc = 0.0
    model.eval()
    for inputs, labels in valid_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        output, val_h = model(inputs, val_h)
        val_loss = criterion(output.squeeze(), labels.float())
        val_losses.append(val_loss.item())

        accuracy = acc(output, labels)
        val_acc += accuracy
```

```
epoch_train_loss = np.mean(train_losses)
epoch_val_loss = np.mean(val_losses)
epoch_train_acc = train_acc / len(train_loader.dataset)
epoch_val_acc = val_acc / len(valid_loader.dataset)

epoch_tr_loss.append(epoch_train_loss)
epoch_vl_loss.append(epoch_val_loss)
epoch_tr_acc.append(epoch_train_acc)
epoch_vl_acc.append(epoch_val_acc)

print(f'Epoch {epoch+1}')
print(f'train_loss : {epoch_train_loss} val_loss : {epoch_val_loss}')
print(f'train_accuracy : {epoch_train_acc*100} val_accuracy : {epoch_

if epoch_val_loss <= valid_loss_min:
    torch.save(model.state_dict(), './working/state_dict.pt')
    print('Validation loss decreased ({:.6f} --> {:.6f}). Saving model
          .format(valid_loss_min, epoch_val_loss))
    valid_loss_min = epoch_val_loss
print(25 * '==')
```

5 Results and Analysis

5.1 Results

In this section, I report the classification accuracy of three model on the testing set. And I will show the performance of RNN, GRU and LSTM on IMDB dataset respectively.

5.1.1 Model 1 (Vanilla RNN)

For the figure 2, RNN with 1 layer performance on IMDB dataset. Here is the deeply interpretation. For the left hand side is the accuracy. We can see that the training accuracy increasing steadily from 50% to roughly 65%. And the validation accuracy shows a more erratic behavior. It starts from 50% and increases to a peak around epoch 3, drops slightly at epoch 4 and the rises to 65% by the end of the training.

For the training loss, it decreases steadily over 5 epochs from 0.7 to 0.6, showing that the model is learning. On the contrary, the validation loss decreases initially and stabilizes around epoch 4 and then rises sharply in later epochs.

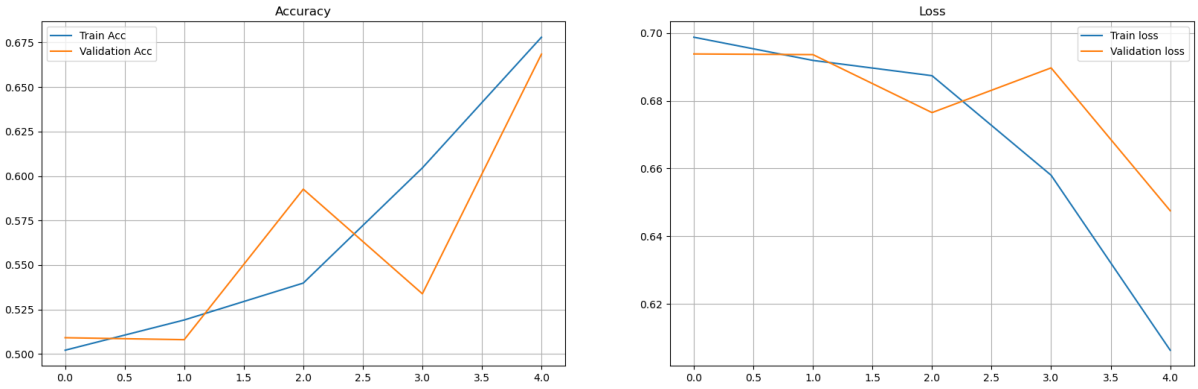


Figure 2: RNN with 1 layer performance on IMDB dataset

5.1.2 Model 4 (Vanilla RNN with 2 layer)

The training accuracy improves over the eopchs, increasing steadily from 50% to 75%. For the validation accuracy also improves, although at a slower pace than training accuracy. Starts at around 50% and reaching approximately 67% by epoch 5.

Then for the tarining loss, it decreases steadily from 0.7 to 0.54, showing that the model is learning effectively and optimizing the training objective. Next for the validation loss, decreases initially, but plateaus and slightly increases after epoch 4.

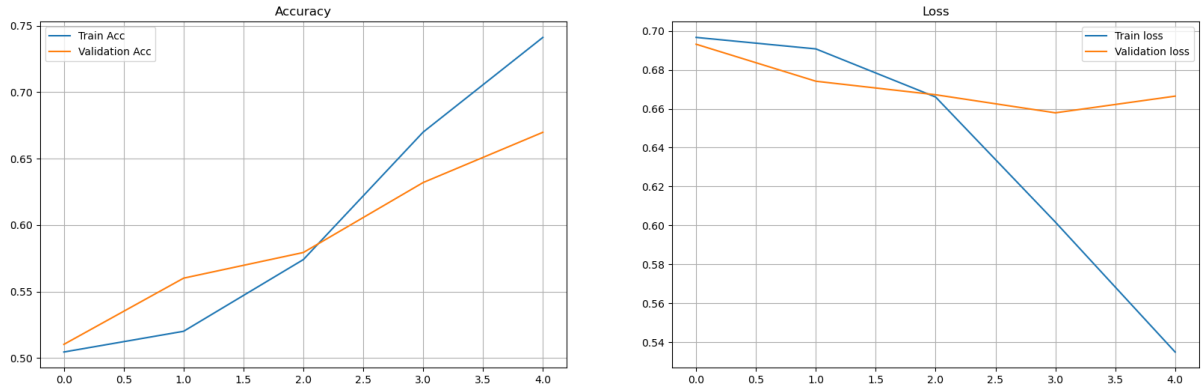


Figure 3: RNN with 2 layer performance on IMDB dataset

5.1.3 Model 3 (Gated recurrent unit)

For the figure 4, GRU with 1 layer, we can see that the training accuracy improves steadily and reaching around 95%, but the validation accuracy remain the same value after epoch 2. Which mean that the model did not learning any new information from the training dataset.

Then for the training loss, the performance is relatively good and drop from 0.7 to 0.15. But for the validation loss, it drop from 0.55 to 0.4 and then rises to 0.45.

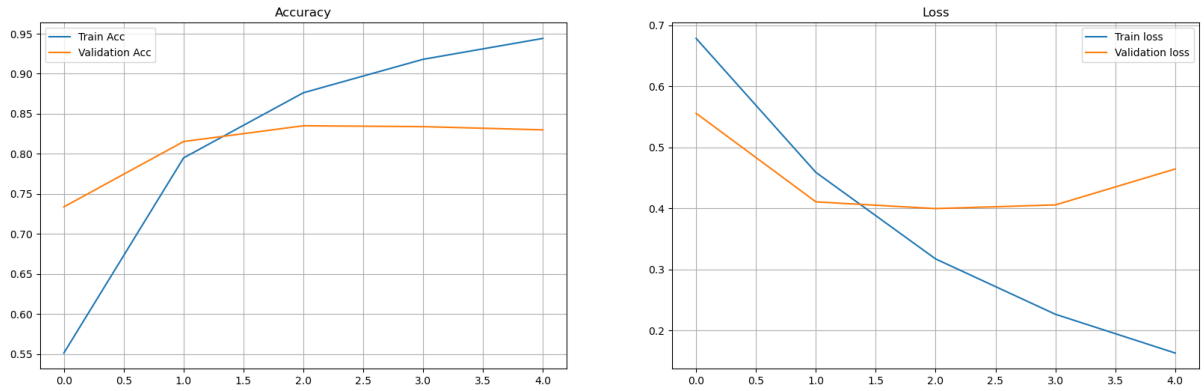


Figure 4: GRU with 1 layer performance on IMDB dataset

5.1.4 Model 5 (Gated recurrent unit with 2 layer)

And next is the performance of GRU with 1 layer, we can see that the training accuracy improves significantly, starting around 60% and reaching over 95% by the end of training. This indicates that the GRU model is fitting the training data very well. On the contrast, the validation accuracy starts at around 80% and improves slightly in the first 2 epochs but then remain on the same value around 85% for the rest of the training. The widening gap between training and validation accuracy after the 2^{nd} epoch suggests overfitting.

Then for the training loss and the validation loss for the GRU model. The training

loss decreases steadily and starting at over 0.6 and dropping to around 0.1 by the 5^{th} epoch. The gap between training and validation accuracy after the second epoch suggests overfitting.

For the loss, the training loss decreases steadily, starting at over 0.6 and dropping to around 0.1 by the 5^{th} epoch. For the validation loss decreases initially and reaching the lowest and then the loss raise over 0.5. Which indicate that overfitting.

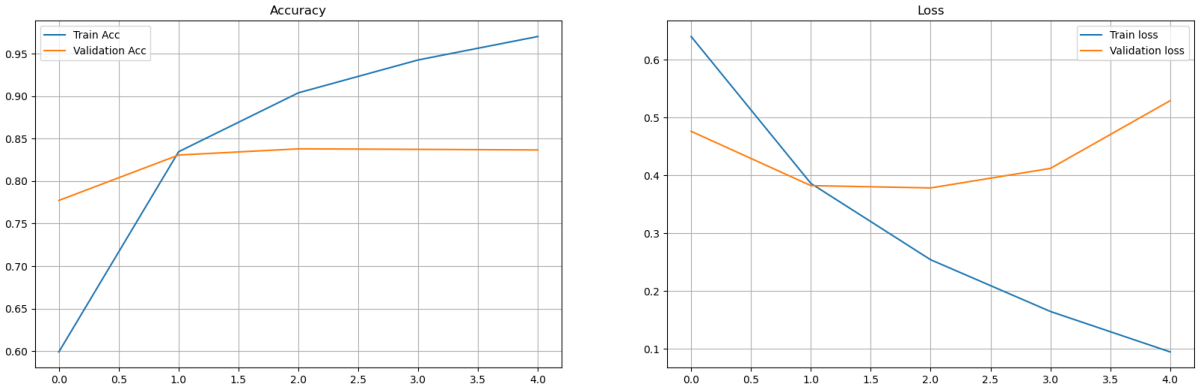


Figure 5: GRU with 2 layer performance on IMDB dataset

5.1.5 Model 2 (Long short-term memory)

And then for the training accuracy of LSTM with 1 layer, the train and validation accuracy increase steadily, for the training accuracy increase from 55% to 90%, for the validation loss is increase from 63% to 83%. For the training loss also decrease steadily, from 0.7 to 0.3. And the validation loss decrease from 0.7 to 0.43.

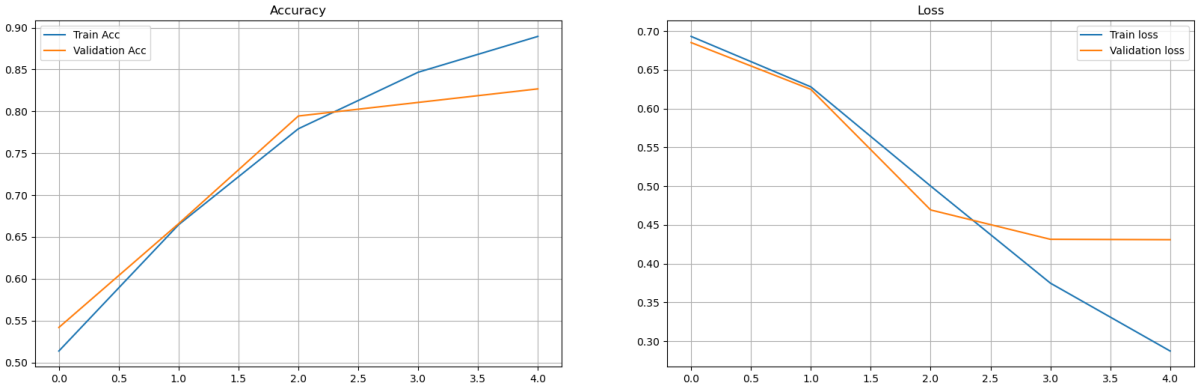


Figure 6: LSTM with 1 layer performance on IMDB dataset

5.1.6 Model 6 (Long short-term memory with 2 layer)

Next is the performance of LSTM with 1 layer, the training accuracy improves steadily over the epochs and starting at around 55% and reaching about 90% at the end of the training. And for the validation accuracy starting at approximately 65% and reaching

about 82.5%. And the loss of the training dataset decreases steadily from about 0.7 to around 0.2. And the validation loss show that the model did learn something from the data.

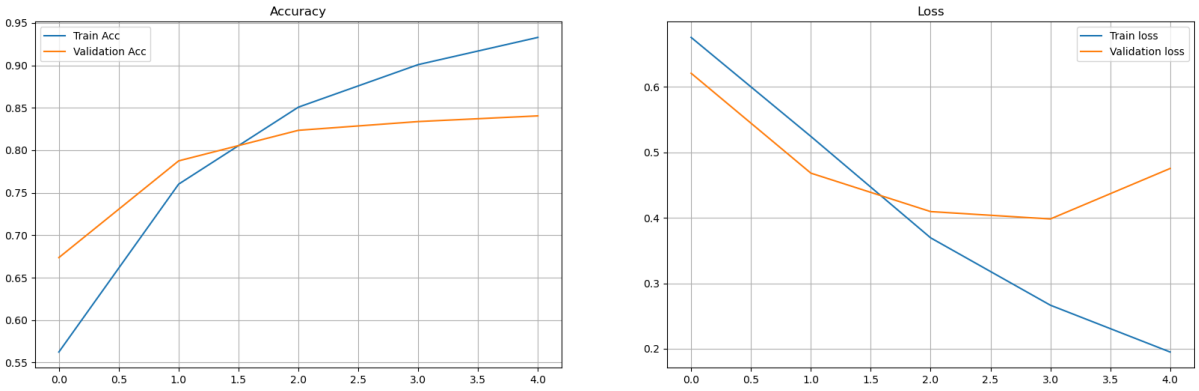


Figure 7: LSTM with 2 layer performance on IMDB dataset

6 Analysis

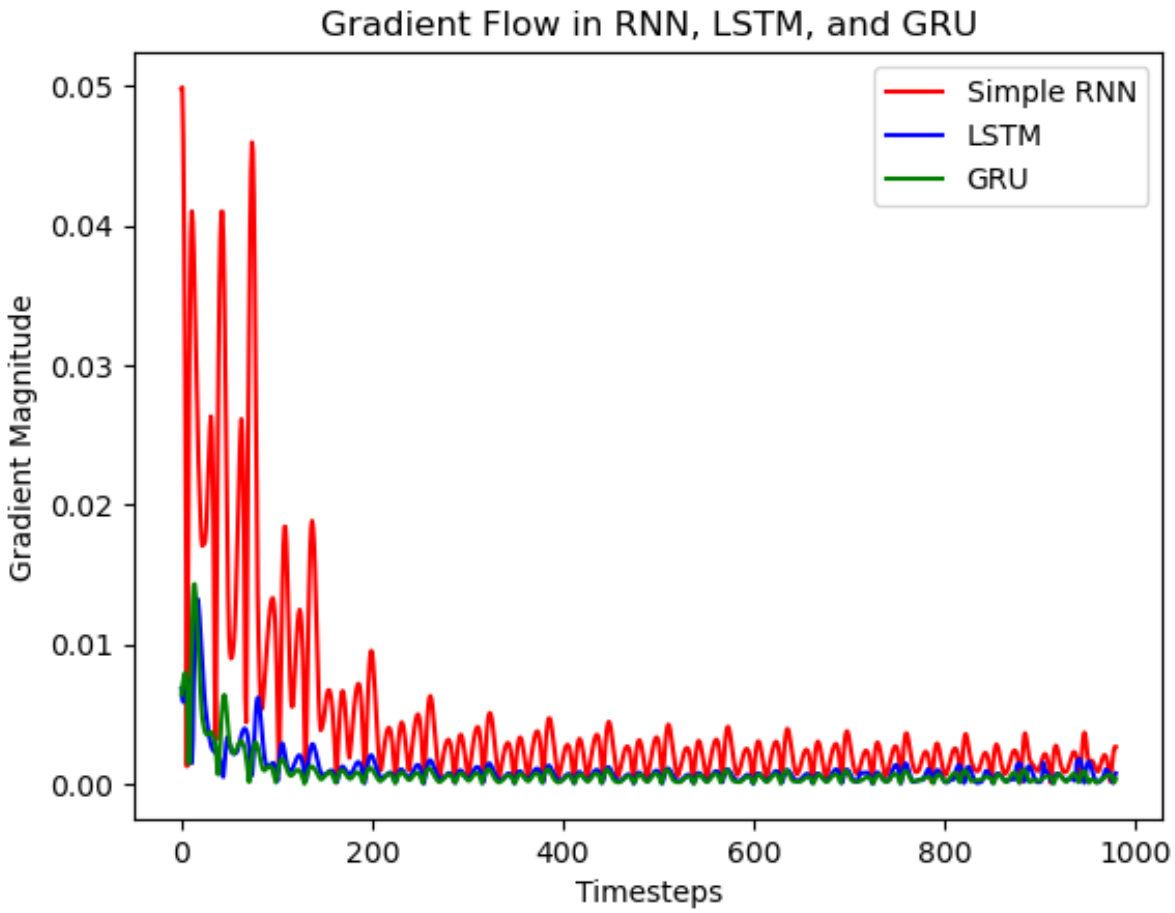
6.1 Learning Behavior

All models show the effective learning during the initial epochs, roughtly 55%, as both training and validation accuracy improve, and loss decrease. But the common problem is overfitting occure in 3 models. The possible reasons is dataset. Since every sentence in dataset was truncated. I just use the first 100 token to training the model. So the existing tokens might not be fully represent the sentence. The second reason is the embedding layers, I used the built-in function in Pytorch to create the embedding layer by randomise the matrix.

Among the 3 models, LSTM and GRU have the better performance compare with vanilla RNN. The reason of GRU and LSTM perform better than RNN is the vanilla RNN struggles with long term dependencies because of the vanishing gradient problem when processing long sequences. In this experience is 100 time step. And this limits its ability to learn the meaningful patterns from data. On the contrast, GRU and LSTM use gate mechanisms to retain the relevant information and discard irrelevant information, allowing them to capture the long term dependencies more effectively.

And for the overfitting problem, we can see that overfitting problem occur in three models. For the vanilla RNN, since this is the basic and simple architecture to handle sequential data, it lacks mechanisms to retain or forget information selectively. As a result, it relying on the memorizing the training data rather than learning the meaningful patterns. This causes it to overfit. And for the GRU and LSTM, although gating mechanisms provide a more sophisticated architecture to learning from data, they aggressively optimizes the training loss, which can result in overfitting when the model continues to learn unnecessary information from the dataset.

The above result and analyse shown that GRU and LSTM perform better than vanilla RNN, which show that the graident vanishing and graident exploding problem were improved by the gating mechanism. And there is so many way will affect the experiment result for example, the embedding layer, tokenization.



6.1.1 Vanilla RNN (Red line)

The gradients in the Vanilla RNN is unstable, as shown by the oscillations and spikes in the early timesteps. Over time, the gradients decay rapidly and go to 0, indicating that the presence of the vanishing gradient problem. This problem occurs because the Vanilla RNNs do not have the gating mechanisms in which this architecture is not perform well in long time dependencies. In other words, as the sequence length increases, repeated matrix multiplications lead to the gradients to shrink exponentially, making it difficult for the model to update the weighting matrix effectively during the BPTT.

6.1.2 LSTM (Blue line)

The gradients of the LSTM are much more stable compared to the simple vanilla RNN. The gradient magnitudes remain consistent across timesteps and do not decay to zero. This stability is achieved through the use of gating machanisms that regulate the flow of information. These gates allow the LSTM to keep the relevant information form the input data while remove some irrelevant information from the data. As a result, LSTM is highly effective at learning long term dependencies in sequences.

6.1.3 GRU (Green line)

The gradients in the GRU are also stable, similar to the LSTM, but the performance is more smoother. GRU achieves this stability using its gate mechanism, which control the degree of information through the network. Compared to the LSTM, the GRU has a simpler architecture, which makes it more computationally efficient while still addressing the vanishing gradient problem.

7 Discussion and Conclusion

7.1 Discussion

During the experimental phase, several insights and challenges emerged, which are discussed below:

- **Unexpected Results:** It was observed that the loss of a two-layer LSTM model was higher compared to that of a one-layer LSTM. This was unexpected because additional layers are generally presumed to capture more complex patterns and improve performance. One possible explanation for this anomaly is the inherent randomness during model initialization and training. Since the performance difference can vary from run to run, it is possible that in some exceptions the two-layer model does not outperform the one-layer configuration.
- **Limitations:** Due to limited computing resources, the experiments were constrained to training the models from scratch with relatively simpler architectures. This restricted the exploration of more complex or state-of-the-art models, such as Transformer-based architectures, which might offer further performance improvements but require substantially more computational power.

7.2 Conclusion

The primary objective of this report was to employ various RNN-based models and compare their performance on a sentiment classification task. The main findings of the project are:

- **Comparative Analysis:** Different RNN-based models, including vanilla RNN, LSTM, and GRU, were implemented and evaluated. The experiments show that LSTM and GRU models effectively address the long-term dependency problem inherent in sequential data. Their internal gating mechanisms allow them to control and preserve information over longer sequences.
- **Efficacy of Gated Architectures:** The results indicate that the gated architectures (LSTM and GRU) significantly outperform the vanilla RNN in terms of handling vanishing and exploding gradient issues, thereby offering more stable and reliable performance.

In conclusion, while some unexpected results were observed—such as the higher loss in the two-layer LSTM model—the overall findings strongly advocate for the use of gated RNN architectures in tasks requiring modeling of long-term dependencies. Despite limitations due to computational resources, this report provides a valuable comparative study and lays the groundwork for future research on more complex models like Transformers in natural language processing tasks.
In conclusion, while the project revealed certain limitations, it nevertheless contributes valuable insights into the training and evaluation of RNN-based models for sentiment classification, thereby setting the stage for further research in the domain of efficient and scalable NLP methodologies.

References

Bahdanau, D., Cho, K., & Bengio, Y. (2016). Neural machine translation by jointly learning to align and translate. https://arxiv.org/abs/1409.0473

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

Luong, M.-T., Pham, H., & Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. https://arxiv.org/abs/1508.04025

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

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. https://arxiv.org/abs/1706.03762

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].