
Department of Mathematical Sciences

# Final Year Project on Deep Learning with its Applications

Author: Yizheng Lu (ID: 201769324)

Project Supervisor: Dr. Youness Boutaib

May 10, 2025

**Abstract**

This project provides a systematic study of neural networks and deep learning, emphasizing both theoretical foundations and practical implementations. It begins with an analysis of feed-forward neural networks, including their structure, backpropagation, and optimization via gradient descent. The study then extends to recurrent models such as RNNs, LSTMs, GRUs, and Echo State Networks (ESNs), addressing challenges like vanishing gradients and proposing solutions such as gating mechanisms and reservoir computing.

Applications span image classification (MNIST), speech recognition (Japanese Vowels), and financial time series forecasting (S&P500). Empirical results demonstrate the strengths of different architectures across tasks, with LSTMs and GRUs excelling in sequential modeling, while feed-forward networks outperform on static inputs. The work offers insights into the comparative performance of deep learning models and their suitability for various data structures.

# Contents

# 1 Feed-forward Neural Network

## 1.1 Supervised Learning

In machine learning, **supervised learning** is a paradigm where a model is trained using **input objects** $X$ (e.g. a vector of predictor variables) and **desired output values** $Y$ (also known as a supervisory signal), which are often human-made labels. Supervised learning is generally used for **regression** and **classification**. An important model of supervised learning is called **Feed-forward Neural Network**.

## 1.2 Definition and Structure of Feed-forward Neural Network

**Definition 1** (Feed-forward Neural Network)**.** A feed-forward neural network is a map $f : \mathbb{R}^{d_1} \to \mathbb{R}^{d_{L+1}}$ in the following form

$$f = f_L \circ f_{L-1} \circ \cdots \circ f_1$$

where $d_1$ is the number of input neurons, $d_{L+1}$ is the number of output neurons, and $L$ is the number of layers. The feed-forward neural network consists of layers of neurons, including an input layer, several hidden layers, and an output layer.

Specifically, for a single training example, given an input vector $x = X_0 \in \mathbb{R}^{d_1}$, we can have

$$x = X_1 = \sigma_1(W_1 X_0 + B_1) \in \mathbb{R}^{d_2}$$

where $W_1 \in \mathbb{R}^{d_2 \times d_1}$ and $B_1 \in \mathbb{R}^{d_2}$. With the similar procedure, we have

$$X_l = \sigma_l(W_l X_{l-1} + B_l) \in \mathbb{R}^{d_{l+1}}$$

and finally gives the output as

$$\hat{y} = X_L = \sigma_L(W_L X_{L-1} + B_L) \in \mathbb{R}^{d_{L+1}}$$

The process of finding $\hat{y}$ for a single training data $(x, y)$ is also called **forward propagation**. Assume there are $m$ training data $(x_1, y_1), \cdots, (x_m, y_m)$, we can define the loss function which gives the gap between predicted value with the actual value.

**Definition 2** (Loss function)**.** The loss function for a feed-forward neural network is defined as

$$l(\hat{y}, y) = \frac{1}{2m} \sum_{i=1}^{m} \|\hat{y}_i - y_i\|$$

**Remark 1.** The choice of loss function is not unique. We use the definition above or other definitions (e.g. cross-entropy loss function) in this paper.

Note that the loss function depends on many parameters $(W_i, \sigma_i, B_i, i \in \{1, 2, \cdots, L\})$. To ensure the best accuracy, our objective is to minimize the loss function $l(\hat{y}, y)$ according to each parameter. We can achieve our objective by using **back propagation** and the **gradient descent** algorithm.

## 1.3 Backpropagation

### 1.3.1 Review of Derivative and Chain Rule

Consider a function $f : \mathbb{R}^n \to \mathbb{R}^d$, the total derivative of the function at $x$ is given by its Jacobian matrix:

$$\begin{aligned} Df(x) &= \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(x) & \frac{\partial f_1}{\partial x_2}(x) & \cdots & \frac{\partial f_1}{\partial x_n}(x) \\ \frac{\partial f_2}{\partial x_1}(x) & \frac{\partial f_2}{\partial x_2}(x) & \cdots & \frac{\partial f_2}{\partial x_n}(x) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_d}{\partial x_1}(x) & \frac{\partial f_d}{\partial x_2}(x) & \cdots & \frac{\partial f_d}{\partial x_n}(x) \end{pmatrix} \\ &= \mathrm{Mat}(df(x)) \end{aligned}$$

Here, $df(x)$ is a linear transformation from $\mathbb{R}^n$ to $\mathbb{R}^d$, i.e. $df(x) \in \mathcal{L}(\mathbb{R}^n, \mathbb{R})$. Consider another map $g : \mathbb{R}^d \to \mathbb{R}^k$, and consider their composition $g \circ f : \mathbb{R}^n \to \mathbb{R}^k$ . The chain rule gives the derivative of $g \circ f$ at $x$ as

$$D(g \circ f)(x) = D(g(f(x)) \cdot D(f(x))$$

Now, we give several examples for the application of derivatives and the chain rule. Consider the linear map $f : \mathbb{R}^n \to \mathbb{R}^d$ where $x \mapsto Ax$, here $A \in \mathbb{R}^{d \times n}$. The derivative of $f(x)$ is given by $Df(x) = A$.

*Proof.* Since $f(x) = Ax$, note that $\frac{\partial f_i}{\partial x_j} = A_{ij}$. The Jacobian Matrix for this linear map is given by

$$Df(x) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_d}{\partial x_1} & \cdots & \frac{\partial f_d}{\partial x_n} \end{pmatrix} = \begin{pmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{d1} & \cdots & A_{dn} \end{pmatrix} = A$$

Hence, $Df(x) = A$. □

Consider $g : \mathbb{R}^d \to \mathbb{R}^d$ where $x \mapsto \sigma(x) = (\sigma(x_1), \cdots, \sigma(x_d))$ for a function $\sigma(x)$. The derivative of $g(x)$ is given by

$$Dg(x) = \begin{pmatrix} \sigma'(x_1) & 0 & \cdots & 0 \\ 0 & \sigma'(x_2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma'(x_d) \end{pmatrix} = \mathbf{diag}(\sigma'(x))$$

which is a diagonal matrix.

*Proof.* Note that $\frac{\partial f_i}{\partial x_i}(x_i) = \sigma'(x_i)$ and $\frac{\partial f_i}{\partial x_j}(x_j) = 0$ for $i \neq j$. Hence, the Jacobian matrix is a diagonal matrix as is shown above.

□

Consider the map $f : \mathbb{R}^n \to \mathbb{R}^d$ where $x \mapsto Ax$ and the map $g : \mathbb{R}^d \to \mathbb{R}^d$ where $z \mapsto \sigma(x) = (\sigma(x_1), \cdots, \sigma(x_d))$ for a function $\sigma(x)$. The derivative of $g \circ f$ is :

$$D(g \circ f) = \mathbf{diag}(\sigma'(Ax)) \cdot A$$

*Proof.* Simply apply the chain rule to the two examples above. □

**Remark 2.** The examples above would be used in the derivation of the formula for back propagation.

### 1.3.2 The Mathematical Derivation for Backpropagation

Firstly, we would like to simplify our calculation by doing the following. Consider the first step of the forward propagation as $X_1 = \sigma_1(W_1X_0 + B_1)$, where $X_0 \in \mathbb{R}^{d_1}$, $W_1 \in \mathbb{R}^{d_2 \times d_1}$ and $B_1 \in \mathbb{R}^{d_2}$. To simplify our calculation, we would combine the $B_1$ term and make the expression into a more compact linear expression as following.

$$X_1 = \sigma_1(W_1X_0 + B_1) = \sigma(\hat{W}_1\hat{X}_0) = \sigma\left(\begin{pmatrix} B_1 & W_1 \end{pmatrix}\begin{pmatrix} 1 \\ X_0 \end{pmatrix}\right)$$

Here, $\hat{X}_0 \in \mathbb{R}^{d_1+1}$ and $\hat{W}_1 \in \mathbb{R}^{d_2 \times (d_1+1)}$. The similar simplification is applied in each layer, and we have

$$X_l = \sigma_l(\hat{W}_L\hat{X}_{l-1}) \in \mathbb{R}^{d_l+1}, l \in \{2, 3, \cdots, l-1\}$$

in each hidden layer, and

$$\hat{y} = X_L = \sigma_L(\hat{W}_L\hat{X}_{L-1}) \in \mathbb{R}^{d_L+1}$$

in the output layer.

To minimize the loss function according to each parameter, we would like to know how does the loss depend on each parameter. We choose $W_{i-1}, i \in \{2, 3, \cdots, L+1\}$ first. First, denote $W = W_{i-1}$ and we define the loss function as a function of $W$

$$g_i(W) := l(y, f_{W_L,\sigma_L} \circ f_{W_{L-1},\sigma_{L-1}} \circ \cdots \circ f_{W_i,\sigma_i} \circ f_{W,\sigma_{i-1}} \circ f_{W_{i-2},\sigma_{i-2}} \cdots \circ f_{w_1,\sigma_1})(X_0)$$

Our goal is to find the derivative of $g_i(W)$ with respect to $W$. We would partition the $g_i(W)$ into two parts. Note that

$$X_{i-2} = (f_{W_{i-2},\sigma_{i-2}} \circ \cdots \circ f_{w_1,\sigma_1})(X_0)$$

and for $1 \leq i \leq L$, we define the function $l_i$ as

$$l_i : u \in \mathbb{R}^{d_i} \mapsto l(y, f_{W_L,\sigma_L} \circ f_{W_{L-1},\sigma_{L-1}} \circ \cdots \circ f_{W_i,\sigma_i})(u) \in \mathbb{R}$$

$$l_{L+1} : u \in \mathbb{R}^{d_{L+1}} \mapsto l(y, u)$$

Thus, the function $g_i(W)$ can be writen in the following form

$$\begin{aligned} g_i(W) &= l(y, \underbrace{f_{W_L,\sigma_L} \circ f_{W_{L-1},\sigma_{L-1}} \circ \cdots \circ f_{W_i,\sigma_i}}_{l_i} \circ f_{W,\sigma_{i-1}} \circ \underbrace{f_{W_{i-2},\sigma_{i-2}} \cdots \circ f_{w_1,\sigma_1})(X_0)}_{\text{gives } X_{i-2}} \\ &= l_i(y, \sigma_{i-1}(WX_{i-2})) \\ &= l_i(\sigma_{i-1}(WX_{i-2})) \end{aligned}$$

Here, in the last formula, we omit the actual output value $y$ for simplicity. Now, we have written $g_i$ as a simple expression with respect to $W$.

Consider $W$ a combination of the transpose of column vectors:

$$W = \begin{pmatrix} W_1^T \\ W_2^T \\ \cdots \\ \underbrace{W_{d_i}^T}_{\in \mathbb{R}_{d_{i-1}}} \end{pmatrix} \in \mathbb{R}^{d_i \times d_{i-1}}$$

Next, we want to transform the matrix $W \in \mathbb{R}^{d_i \times d_{i-1}}$ into a vector $w$

$$w = \begin{pmatrix} W_1 \\ W_2 \\ \cdots \\ W_{d_i} \end{pmatrix} \in \mathbb{R}^{d_i \cdot d_{i-1}}$$

where $W_1, \cdots W_{d_i} \in \mathbb{R}^{d_{i-1}}$. To make the matrix multiplication $WX_{i-2}$ valid, we need to reconstruct $X_{i-2}$ as follows

$$\begin{aligned} WX_{i-2} &= \begin{pmatrix} W_1^T \\ W_2^T \\ \cdots \\ W_{d_i}^T \end{pmatrix} \cdot X_{i-2} \\ &= \begin{pmatrix} X_{i-2}^T & 0 & \cdots & 0 \\ 0 & X_{i-2}^T & \cdots & 0 \\ \vdots & \vdots & \ddots & \\ 0 & 0 & 0 & X_{i-2}^T \end{pmatrix} \cdot \begin{pmatrix} W_1 \\ W_2 \\ \cdots \\ W_{d_i} \end{pmatrix} \\ &= \widetilde{X}_{i-2} w \in \mathbb{R}^{d_i} \end{aligned}$$

where the $0 \in \mathbb{R}^{d_{i-1}}$ are zero vectors, $\widetilde{X}_{i-2} \in \mathbb{R}^{d_i \times (d_i \cdot d_{i-1})}$ and $w \in \mathbb{R}^{d_i \cdot d_{i-1}}$.

Recall the original function with respect to a matrix is $g_i(W) = l_i(\sigma_{i-1}(WX_{i-2}))$, we have constructed a new function with respect to a vector $\widetilde{g}_i : \mathbb{R}^{d_i \cdot d_{i-1}} \to \mathbb{R}$ where $\widetilde{g}_i(w) = l_i(\sigma_{i-1}(\widetilde{X}_{i-2}w))$.

Consider the derivative of $\widetilde{g}_i(w)$ and apply the chain rule, we have

$$\begin{aligned} D\left(\widetilde{g}_i(w)\right) &= D\left(l_i(\sigma_{i-1}(\widetilde{X}_{i-2}w))\right) \\ &= D\left(l_i(\sigma_{i-1}(\widetilde{X}_{i-2}w))\right) \cdot D\left(\sigma_{i-1}(\widetilde{X}_{i-2}w)\right) \\ &= D\left(l_i(X_{i-1})\right) \cdot \mathbf{diag}(\sigma'_{i-1}(\widetilde{X}_{i-2}w)) \cdot \widetilde{X}_{i-2} \end{aligned}$$

since $\sigma_{i-1}(\widetilde{X}_{i-2}w) = \sigma_{i-1}(WX_{i-2}) = X_{i-1}$. We denote $\delta_i = Dl_i(X_{i-1})$ and $WX_{i-2} = \widetilde{X}_{i-2}w = Z_{i-1}$, namely

$$D\widetilde{g}_i(w) = \delta_i \cdot \mathbf{diag}(\sigma'_{i-1}(Z_{i-1})) \cdot \widetilde{X}_{i-2}$$

Recall

$$\begin{aligned} l_i(u) &= l\left(y, f_{W_L,\sigma_L} \circ \cdots \circ f_{W_i,\sigma_i}(u)\right) \\ &= l\left(y, f_{W_L,\sigma_L} \circ \cdots \circ f_{W_{i+1},\sigma_{i+1}}\left(f_{W_i,\sigma_i}(u)\right)\right) \\ &= l_{i+1}(\sigma_i(W_i u)) \end{aligned}$$

Hence,

$$Dl_i(u) = Dl_{i+1}(\sigma_i(W_i u))$$

Then,

$$\begin{aligned} \delta_i &= Dl_i(X_{i-1}) \\ &= Dl_{i+1}(\sigma_i(W_i X_{i-1})) \\ &= Dl_{i+1}(X_i) \cdot \mathbf{diag}(\sigma'_i(Z_i)) \cdot W_i \\ &= \delta_{i+1} \cdot \mathbf{diag}(\sigma'_i(Z_i)) \cdot W_i \end{aligned}$$

Note that $\delta_{L+1} = Dl_{L+1}(X_L) = Dl_y(X_L)$. Namely, we have a formula for the $D(g_i(W))$, or in our new notation, $D\widetilde{g}_i(w)$ using back propagation. We summarize the steps as follows:

- Step1: Given $x = X_0$, compute $X_1, Z_1, \cdots, X_L = \hat{y}, Z_L$. This step is called forward propagation.
- Step2: Compute $\delta_{L+1} = \frac{\partial l(y, X_L)}{\partial X_L}$, then use the iteration formula $\delta_i = \delta_{i+1}\mathbf{diag}(\sigma_i'(Z_i))W_i$. This step is called backward propagation and we get $(\delta_1, \delta_2, \cdots, \delta_L, \delta_{L+1})$.
- Step3: Use the formula $D\tilde{g}_i(w) = \delta_i\mathbf{diag}(\sigma_{i-1}'(S_{i-1}))\widetilde{X}_{i-2}$ to calculate the derivative of $\tilde{g}_i$ with respect to $w$, then hence $W$.

The derivative of other parameters can be calculated similarly. Now, we know how the loss function depend on each parameter. We would use the gradient descent technique to minimize the loss function.

## 1.4 Gradient Descent

The gradient descent algorithm is used to minimize a differentiable and convex function. It is based on the observation that a function $g$ decreases fastest if one goes from a point $a$ in the direction of the negative gradient. For a parameter $w$, given an initial point $w_0$, we can have the following iteration equation:

$$w_{t+1} = w_t - \alpha(t)\frac{\partial g}{\partial w}(w_t), t \in \{0, 1, \cdots, T-1\}$$

where $T$ is the number of iteration and $\alpha(t)$ is the learning rate. For simplicity, we choose $\alpha(t) = \alpha \in \mathbb{R}$. However, the **Adam Optimizer (Adaptive Moment Estimation)** allows us to adapt the learning rate $\alpha(t)$ for each parameter. The figure below shows the process of gradient descent.

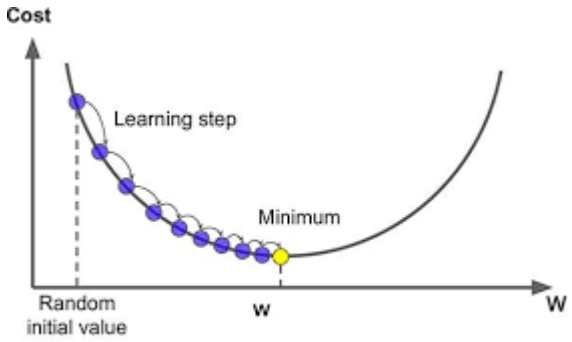



Figure 1: Illustration of Gradient Descent

The following theorem shows that after each iteration, the value of the function decreases and eventually be close to the minimal. For simplicity, we show and prove the theorem for uni-variate function.

**Theorem 1.** Let $g : \mathbb{R} \to \mathbb{R}$ be a continuously differentiable function and let $x_0 \in \mathbb{R}$. Assume that $f'(x_0) \neq 0$, there exist $\alpha_{x_0} > 0$ such that for any $\alpha$ satisfying $0 < \alpha < \alpha_{x_0}$, we have

$$f(x_0 - \alpha f'(x_0)) < f(x_0)$$

*Proof.* To fix the ideas, we assume that $f'(x_0) > 0$. Since $f'$ is continuous, then there exists $\beta_{x_0} > 0$ such that

$$\forall x \in [x_0 - \beta_{x_0}, x_0 + \beta_{x_0}] : f'(x) > 0.$$

We define

$$\alpha_{x_0} = \frac{\beta_{x_0}}{f'(x_0)}.$$

Let $0 < \alpha < \alpha_{x_0}$. Note then that

$$x_0 - \beta_{x_0} < x_0 - \alpha f'(x_0) < x_0 < x_0 + \beta_{x_0}.$$

By the mean value theorem, there exists $c \in (x_0 - \alpha f'(x_0), x_0)$ such that

$$f(x_0 - \alpha f'(x_0)) - f(x_0) = (x_0 - \alpha f'(x_0) - x_0)f'(c) = -\alpha f'(x_0)f'(c).$$

Since $c \in [x_0 - \beta_{x_0}, x_0 + \beta_{x_0}]$, then $f'(c) > 0$. Hence

$$f(x_0 - \alpha f'(x_0)) - f(x_0) < 0.$$

□

Although the proof above shows the existence of such $\alpha_{x_0}$, the following alternative proof gives a specific expression for a possible $\alpha_{x_0}$.

*Proof.* We apply the Taylor expansion of $f$ at $x_0$ up to the second order

$$f(x_0 - h) = f(x_0) - hf'(x_0) + \frac{h^2}{2}f''(\xi),$$

where $\xi$ lies between $x_0 - h$ and $x_0$. Substituting $h = \alpha f'(x_0)$ into the expansion and compute the difference, we have

$$f(x_0 - \alpha f'(x_0)) - f(x_0) = -\alpha f'(x_0)^2 + \frac{\alpha^2}{2}f'(x_0)^2 f''(\xi).$$

We want this expression to be negative, that is

$$-\alpha f'(x_0)^2 + \frac{\alpha^2}{2}f'(x_0)^2 f''(\xi) < 0. \implies \alpha f'(x_0)^2 \left(-1 + \frac{\alpha}{2}f''(\xi)\right) < 0.$$

Since $f'(x_0)^2 > 0$, the inequality simplifies to:

$$-1 + \frac{\alpha}{2}f''(\xi) < 0.$$

Solving for $\alpha$:

$$\frac{\alpha}{2}|f''(\xi)| < 1 \implies \alpha < \frac{2}{|f''(\xi)|}.$$

Let $M = \sup_\xi |f''(\xi)|$ in a neighborhood around $x_0$. Then choose:

$$\alpha_{x_0} = \frac{2}{M}.$$

For any $0 < \alpha < \alpha_{x_0}$, the inequality holds:

$$f(x_0 - \alpha f'(x_0)) < f(x_0).$$

□

**Remark 3.** The theorem is valid for multi-variable function and the proof is similar.

With this theorem, we have shown the gradient descent algorithm is an iteration process that helps to find the minimal of a function. The back propagation gives a formula for the derivative of the loss function with respect to a parameter. Together with the gradient descent process, this can be help to find the parameters that minimize the loss function.

The graph below shows the process of constructing and training a feed-forward neural network.

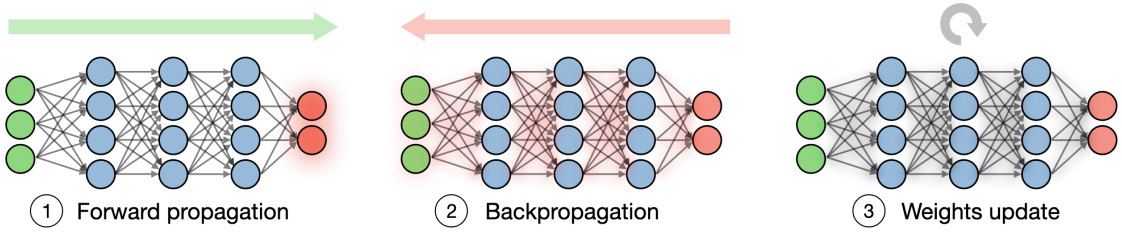



Figure 2: Construction and Training of a Feed-forward Neural Network

## 1.5 Example: MNIST

MNIST (Modified National Institute of Standards and Technology) is one of the most famous datasets in machine learning and computer vision. It consists of 70,000 grayscale images of handwritten digits ($0-9$), each of size $28 \times 28$ pixels. Among them, 60,000 images are used for training, and 10,000 are reserved for testing. The dataset is widely used for benchmarking classification algorithms and is often considered the "Hello World" of deep learning.

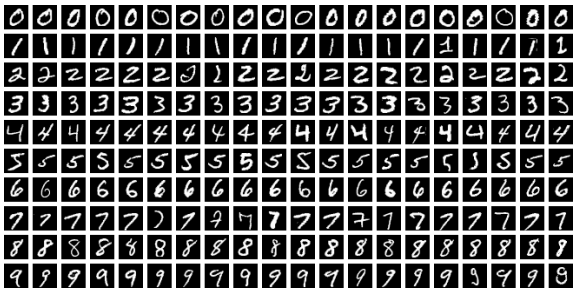


Figure 3: Sample Image from MNIST Sample Data

We would use a feed-forward neural network (FNNs) to train this model. For our FNNs, we use 784 neurons (one for each pixel in the 28×28 image) in the input layer and apply ReLU activation function in the hidden layer. Then, the desired output layer has 10 neurons (one for each digit, 0-9) with a softmax activation function to classify the digits.

We would use the cross-entropy loss function as a loss function, which is defined as

$$\text{Loss} = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

where $y_i$ is the true label, $\hat{y}_i$ is the predicted probability for class $i$ and $C$ is the number of classes (for MNIST, $C = 10$).

In this case, we choose Adam Optimizer to train the model since it is a simple model and the data volume is not too large. We want to find the best learning rate for the Adam Optimizer. Our goal is to select a learning rate such that the loss decreases quickly, but not too much shock or cause instability in training. We plot a line chart (Figure 3) comparing the loss with different learning rates. We find the learning rate as 0.001519, which is the point where the loss function decreases most rapidly. With this learning rate, we iterate our training 20 epochs and note that the loss converges quickly. The final loss is approximately 0.00505 in the 20th epoch and we reach a prediction accuracy of 97.85% in the test set.

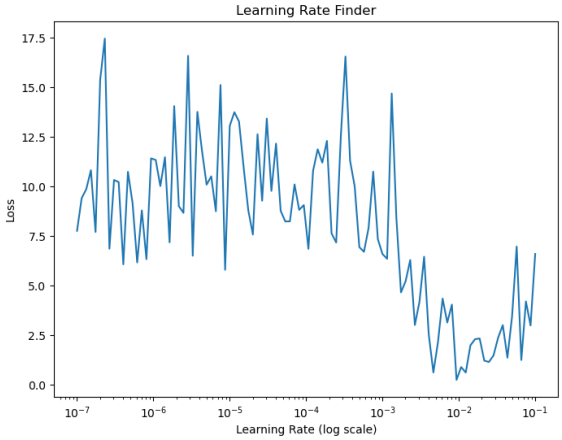



Figure 4: Learning Rate Finder

We randomly choose 20 samples from the test set and put the predicted result together with their true label in the figure below. We note that the overall recognition accuracy is good, with only one mistake for the recognition of label 5.

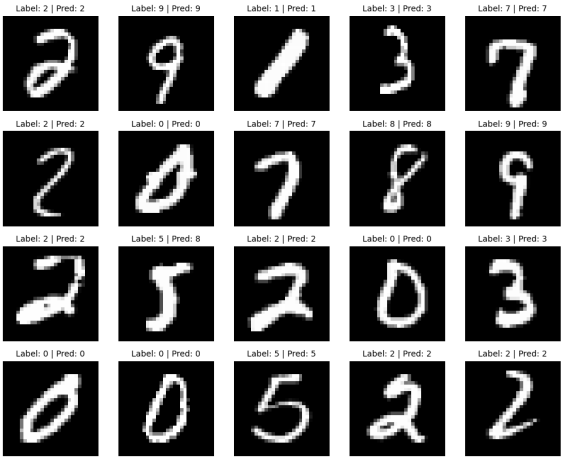



Figure 5: 20 Samples from the Test Set

# 2 Recurrent Neural Network

## 2.1 Introduction

**Recurrent neural networks (RNNs)** are a class of artificial neural networks designed for processing **sequential data**, such as text and audio. Unlike feedforward neural networks, RNNs utilize recurrent connections, where the output of a neuron at one time step is fed back as input to the network at the next time step. This enables RNNs to capture temporal dependencies and patterns within sequences.

Unlike feedforward neural networks, which only propagate data forward through the network, RNNs have connections that allow data to flow in both directions, enabling the network to process and store information about past inputs. At each time step, the RNN processes the current input along with the previous hidden state to produce an output and an updated hidden state, which is then passed to the next time step. Like the normal FNNs, the weight of RNNs are trained through a type of backpropagation, called backpropagation through time (BPTT).

While FNNs are usually used for classification and regression, RNNs are often used for forecasting. However, RNNs often suffer from short-term memory. Consider a example that we want to know what should be filled in in the blank of the sentence below.

**I am from China and I live in Liverpool now. So I can speak fluent _____.**

Obviously, the blank should be Chinese. But it depends on the information at the beginning of the sentence. However, due to the short-term memory of RNNs, i.e. the network may not remember the information that appears a lot time ago, the network may not accuately predict the desired word for the blank. To combat the short-term memory, we may use **LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit)** to make the network be able to learn from long-term memory.

We would start with the simplest architecture of RNNs, which is called the simple recurrent network (SRN), and then see any variations that could be applied on it.

## 2.2 Architecture of Simple Recurrent Network (SRN)

Here we define the most simple and common architecture of recurrent neural network. Recurrent neural network allows previous (in the sense of time step) outputs to be used as inputs while having hidden states. Consider at each time step $t$, the input $x^{\langle t\rangle} \in \mathbb{R}^p$ and the previous hidden state $a^{\langle t-1\rangle} \in \mathbb{R}^N$. Then, the input $x^{\langle t\rangle} \in \mathbb{R}^p$ is parameterized by a matrix $W_{ax} \in \mathbb{R}^{N\times p}$ and $a^{\langle t-1\rangle}$ is parameterized by a matrix $W_{aa} \in \mathbb{R}^{N\times N}$. Denote

$$z^{\langle t\rangle} = W_{aa}a^{\langle t-1\rangle} + W_{ax}x^{\langle t\rangle} + b_a \in \mathbb{R}^N$$

And set the activation function $g_1 : \mathbb{R}^N \to \mathbb{R}^N$, we have

$$a^{\langle t\rangle} = g_1(z^{\langle t\rangle}) = g_1(W_{aa}a^{\langle t-1\rangle} + W_{ax}x^{\langle t\rangle} + b_a) \in \mathbb{R}^N$$

Then, we parameterize $a^{\langle t\rangle}$ using a matrix $W_{ya} \in \mathbb{R}^{d\times N}$ and yield

$$o^{\langle t\rangle} = W_{ya}a^{\langle t\rangle} + b_y \in \mathbb{R}^d$$

Apply a activation function $g_2 : \mathbb{R}^d \to \mathbb{R}^d$ on it

$$y^{\langle t\rangle} = g_2(o^{\langle t\rangle}) = g_2(W_{ya}a^{\langle t\rangle} + b_y) \in \mathbb{R}^d$$

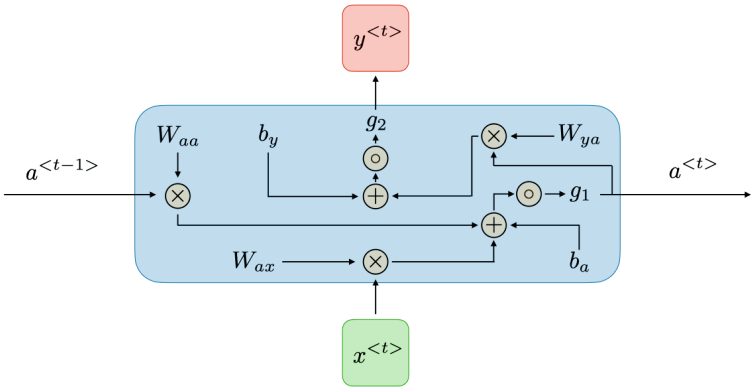



Figure 6: Architecture of Recurrent Neural Network

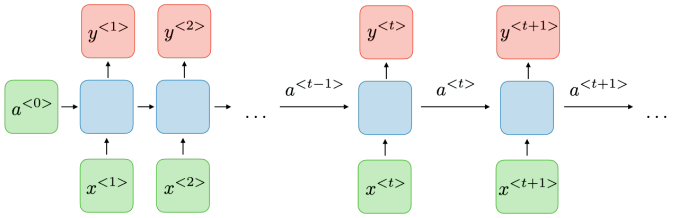



Figure 7: Overall Workflow of Recurrent Neural Network

Note that, $g_1$ and $g_2$ are non-linear activation function, which would be often selected as the ReLU or tanh function.

## 2.3 Backpropagation Through Time (BPTT)

Similar to the stand backpropagation that has been introduced in section 1.3, we would introduce the backpropagation that would be used in FNNs, which is the so-called Backpropagation through time (BPTT). However, unlike standard backpropagation, we would like to consider the temporal dimension for the RNNs. Finally, We would like to find the partial derivative of the loss function with repesct to all the parameters.

First, we assume the time dimension of the RNNs is $\{1, 2, \cdots, T\}$. We construct a loss function $L : \mathbb{R}^T \to \mathbb{R}$, where $(\hat{y}^{\langle 1\rangle}, \cdots, \hat{y}^{\langle T\rangle}) \mapsto L(\hat{y}^{\langle 1\rangle}, \cdots, \hat{y}^{\langle T\rangle})$. Also, we define the loss function $l_t : \mathbb{R}^d \to \mathbb{R}$ and $\phi_t : \mathbb{R}^N \to \mathbb{R}$ at time step $t$ as

$$l_t(o^{\langle t\rangle}) = l_t(W_{ya}a^{\langle t\rangle} + b_y) = \phi_t(a^{\langle t\rangle}) = L(\hat{y}^{\langle 1\rangle}, \cdots, g_2(o^{\langle t\rangle}), \cdots, \hat{y}^{\langle T\rangle})$$

Then, we would define two intermediate value $\alpha$ and $\beta$. Let $\beta^{\langle t\rangle} := D(l_t(o^{\langle t\rangle})) = D\left(l_t(W_{ya}a^{\langle t\rangle} + b_y)\right)$, then by chain rule, we have

$$\beta^{\langle t\rangle} = D_{\hat{y}^{\langle t\rangle}}(L) \cdot \mathbf{diag}(g_2'(o^{\langle t\rangle}))$$

Then, define $\alpha^{\langle t\rangle} := D\phi_t(a^{\langle t\rangle})$, since $a^{\langle t\rangle}$ would go to two directions, $a^{\langle t+1\rangle}$ and the ouput $\hat{y}^{\langle t\rangle}$, also by chain rule, we have

$$\alpha^{\langle t\rangle} = \alpha^{\langle t+1\rangle} \cdot \mathbf{diag}(g_1'(z^{\langle t\rangle})) \cdot W_{aa} + \beta^{\langle t\rangle} \cdot W_{ya}$$

Then, consider the last step $T$ , since $\alpha^{\langle T\rangle}$ will not flow to $\alpha^{\langle T+1\rangle}$, we have

$$\alpha^{\langle T\rangle} = \beta^{\langle T\rangle} \cdot W_{ya}$$

Like the standard backpropagation, we would iterate backward similarly. Then, we define the loss function with respect to all the parameters as $J : (W_{aa}, W_{ax}, b_a, W_{ya}, b_y) \mapsto L(\hat{y}^{\langle 1\rangle}, \cdots, \hat{y}^{\langle T\rangle})$. Then, the derivative of $J$ with resepct to all the parameters is:

$$
\begin{aligned}
D_{b_y}(J) &= \sum_{t=1}^{T} \beta^{\langle t\rangle} \\
D_{b_a}(J) &= \sum_{t=1}^{T} \alpha^{\langle t\rangle} \cdot \mathbf{diag}(g_1'(z^{\langle t\rangle})) \\
D_{W_{ya}}(J) &= \sum_{t=1}^{T} \beta^{\langle t\rangle} \cdot A_{ya}^{\langle t\rangle} \\
D_{W_{aa}}(J) &= \sum_{t=1}^{T} \alpha^{\langle t\rangle} \cdot \mathbf{diag}(g_1'(z^{\langle t\rangle})) \cdot A_{aa}^{\langle t-1\rangle} \\
D_{W_{ax}}(J) &= \sum_{t=1}^{T} \alpha^{\langle t\rangle} \cdot \mathbf{diag}(g_1'(z^{\langle t\rangle})) \cdot X_{ax}^{\langle t\rangle}
\end{aligned}
$$

where $A_{ya}^{\langle t\rangle}$ , $A_{aa}^{\langle t\rangle}$ and $X_{ax}^{\langle t\rangle}$ are three very wide matrix given by:

$$
A_{ya}^{\langle t\rangle} = \begin{pmatrix}
(a^{\langle t\rangle})^T & 0 & 0 & \cdots & 0 \\
0 & (a^{\langle t\rangle})^T & 0 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \cdots & (a^{\langle t\rangle})^T & 0 \\
0 & 0 & \cdots & 0 & (a^{\langle t\rangle})^T
\end{pmatrix} \in \mathbb{R}^{d \times Nd}
$$

$$
A_{aa}^{\langle t\rangle} = \begin{pmatrix}
(a^{\langle t\rangle})^T & 0 & 0 & \cdots & 0 \\
0 & (a^{\langle t\rangle})^T & 0 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \cdots & (a^{\langle t\rangle})^T & 0 \\
0 & 0 & \cdots & 0 & (a^{\langle t\rangle})^T
\end{pmatrix} \in \mathbb{R}^{N \times N^2}
$$

$$
X_{ax}^{\langle t\rangle} = \begin{pmatrix}
(x^{\langle t\rangle})^T & 0 & 0 & \cdots & 0 \\
0 & (x^{\langle t\rangle})^T & 0 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \cdots & (x^{\langle t\rangle})^T & 0 \\
0 & 0 & \cdots & 0 & (x^{\langle t\rangle})^T
\end{pmatrix} \in \mathbb{R}^{N \times Np}
$$

The process of BPTT is similar to the standard backpropagation in section 1.3.

## 2.4 Vanishing Gradient and Exploding Gradient

Note from the previous section that the gradient will depends on the derivative of the activation function $g_1$ and $g_2$. If the temporal dimension of the RNNs is large, the length of the chain rule would be large. Thus, for a long series of multiplication, if the derivative of the activation function is smaller than 1, the gradient would converge to 0, which is the so-called vanishing gradient problem. However, if the derivative of the activation function is larger than 1, the gradient would converge to $\infty$, which is the so-called exploding gradient problem.

Many activation functions would have a derivative that is smaller than 1. For example, consider the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$, where the derivative is given by $\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$, where $\max_{x \in (0,1)} \sigma'(x) = 0.25$, and when $x \to \infty$, $\sigma'(x) \to 0$. This may cause the gradient vanishing problem. Similarly, if $\sigma(x) = \tanh(x)$, we have $\max_{x \in (-1,1)} \sigma'(x) = 1$ but will decrease rapidly. Also, as $x \to \infty$, $\sigma'(x) \to 0$. This will also cause vanishing gradient problem.

Consider the ReLU activation function $\sigma(x)$, where the derivation is given by

$$\sigma'(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

We note that when $x > 0$, the derivative is 1, which will not cause the vanishing gradient problem. However, there are two problems regarding the ReLU function. Firstly, since there is no restriction for the derivative when $x > 0$, the output for a deep network may be vert large. This may cause the exploding gradient problem. Secondly, since $\sigma'(x) = 0$ when $x \leq 0$, it may cause the problem of dead neuron, which means the activation function will not activate the input and update the network through BPTT since the gradient is zero.

To avoid the problem of dead neuron, we may use the Leaky ReLU $\sigma_{\text{Leaky}}(x)$ as the input function. Usually, $\alpha$ is a small value like 0.01.

$$\sigma_{\text{Leaky}}(x) = \begin{cases} x, & x > 0 \\ \alpha x, & x \leq 0 \end{cases}, \quad \alpha \in (0, 1)$$

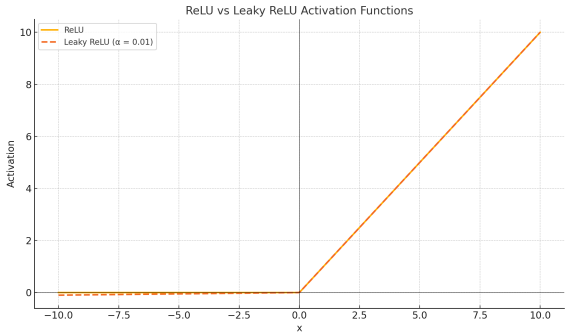



Figure 8: ReLU and Leaky ReLU

Choosing the Leaky ReLU activation function will effectively reduce the risks of dead neuron, but the problem of exploding gradient still exists. The common technique to combat exploding gradient is gradient clipping. We manually set a threshold value $\tau$ for the gradient. When the $L_2$-norm of gradient $\nabla$ exceeds the threshold $\tau$, we scale down the gradient $\nabla$ proportionally but do not change its direction. Mathematically, we have

$$\nabla_{\text{clipped}} = \begin{cases} \nabla, & \text{if } \|\nabla\|_2 \leq \tau \\ \frac{\tau}{\|\nabla\|_2} \cdot g, & \text{if } \|\nabla\|_2 > \tau \end{cases}$$

Vanishing gradient becomes a problem when the gap between information needed is very large, i.e. when the model requires a long-time memory. There are several ways that would directly or indirectly solve this problem. Directly, one would use LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit) which construct memory cell and gates to solve this problem. Alternatively,

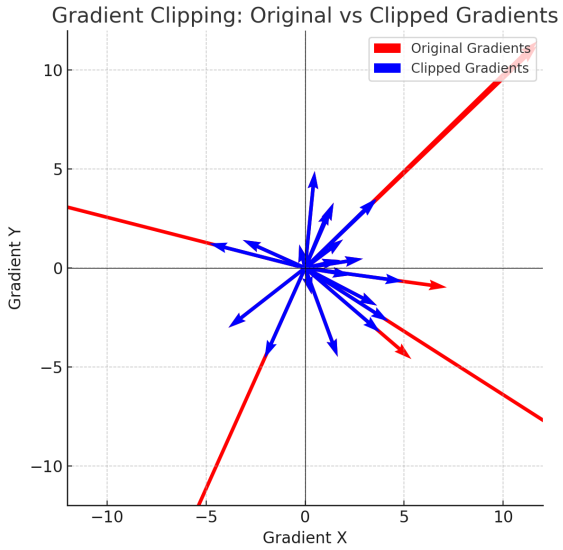



Figure 9: Gradient Clipping for $\tau = 5$

one could use reservoir computing (RC) method to avoid backpropagation in the recurrent part, and thus indirectly avoid the vanishing gradient problem.

## 2.5 Reservoir Computing - Echo State Network

### 2.5.1 Introduction to Reservoir Computing

Reservoir computing is a framework for computation derived from recurrent neural network theory that maps input signals into higher dimensional computational spaces through the dynamics of a fixed, non-linear system called a reservoir. Training is performed only at the output stage, as the reservoir dynamics are fixed. Thus, no backpropagation through time is performed and thus, the problem of vanishing gradient is avoided. An echo state network (ESN) is a type of reservoir computer.

### 2.5.2 Basic Framework of Echo State Network

We will give the basic framework of a ESN. At each time step $t \in \{1, \cdots, T\}$, we have the input $x^{\langle t \rangle} \in \mathbb{R}^p$ and the previous hidden state $a^{\langle t-1 \rangle} \in \mathbb{R}^N$. Then, the input $x^{\langle t \rangle} \in \mathbb{R}^p$ is parameterized by a matrix $W_{\text{in}} \in \mathbb{R}^{N \times p}$ and $a^{\langle t-1 \rangle}$ is parameterized by a matrix $W_{\text{hidden}} \in \mathbb{R}^{N \times N}$. Here, $W_{\text{hidden}}$ and $W_{\text{in}}$ is fixed after initialization. For a nonlinear activation function $g_1 : \mathbb{R}^N \to \mathbb{R}^N$, we have

$$a^{\langle t \rangle} = g_1 \left( W_{\text{hidden}} a^{\langle t-1 \rangle} + W_{\text{in}} x^{\langle t \rangle} \right) \in \mathbb{R}^N$$

Also, for $W_{\text{out}} \in \mathbb{R}^{d \times N}$, we have

$$y^{\langle t \rangle} = W_{\text{out}} \cdot a^{\langle t \rangle} \in \mathbb{R}^d$$

The only parameter that will be trained is $W_{\text{out}}$. Because of this, ESNs can yield much faster results and sometimes even desirable, in particular when dealing with long-term time series predictions.

### 2.5.3 Echo State Property

Dive deeply in to reservoir computing, we can view the ESNs as discrete time nonlinear dynamic systems driven by inputs. We can note that

$$a^{\langle t \rangle} = g_1\left(W_{\text{hidden}} a^{\langle t-1 \rangle} + W_{\text{in}} x^{\langle t \rangle}\right) = \mathcal{F}(a^{\langle t-1 \rangle}, x^{\langle t \rangle})$$

for a dynamic system $\mathcal{F}$, where $x^{\langle t \rangle}$ is the input. The most frequently asked question for a dynamic system is: if we start from a different initial state $a^{\langle 0 \rangle}$ and given the same input sequence $x^{\langle t \rangle}$, will the dynamic system eventually converge to the same trajectory? i.e. If the long-term behavior of the system only depends on the input, regardless of the starting point? This leads to the so-called Echo State Property (ESP). Before stating and proving ESP, we will give several definitions and theorems.

**Definition 3** (Spectral Radius)**.** The spectral radius of a square matrix $A$ is the maximum of the absolute values of its eigenvalues. That is, given $A \in \mathbb{C}^{n\times n}$, the spectral radius is $\rho(A) := \max\{|\lambda_1|, \cdots, |\lambda_n|\}$.

**Theorem 2.** For a square matrix $A \in \mathbb{C}^{n\times n}$, if $\rho(A) < 1$, then $\lim_{t\to\infty} A^t = 0$.

**Definition 4** (Lipschitz Continuous)**.** Given two metric space $(X, d_X)$ and $(Y, d_Y)$, a function $X : X \to Y$ is called Lipschitz continuous if there exists a real constant $L > 0$, such that, for all $x_1, x_2 \in X$, $d_Y(f(x_1), f(x_2)) \leq L \cdot d_X(x_1, x_2)$. Such constant $L$ is called the Lipschitz constant.

**Remark 4.** For $L \in [0, 1)$, such function $f$ is called a contraction.

**Theorem 3** (Banach Fixed Point Theorem)**.** Let $(X, d)$ be a non-empty complete metric space, and let $f : X \to X$ be a contraction mapping, then $f$ has a unique fixed point $x^* \in X$, i.e., $f(x^*) = x^*$. Moreover, for any $x_0 \in X$, the sequence defined by $x_{n+1} = f(x_n)$ converges to $x^*$.

**Definition 5** (Operator Norm)**.** Given two normed vector space $V$ and $W$, and a linear map $A : V \to W$, then the operator norm of $A$ is defined as

$$\|A\|_{\text{op}} := \inf\{c \geq 0 : \|Av\| \ \leq c\|v\|\}$$

for every $v \in V$.

**Theorem 4** (Echo State Property)**.** For a ESN dynamic system $\mathcal{F}$, if a sufficient condition (will be given below) is satisfied, given the same bounded input sequence $\{x^{\langle t \rangle}\}_{t=1}^{T}$, for any two different starting states $a^{\langle 0 \rangle}$ and $\tilde{a}^{\langle 0 \rangle}$, we have

$$\lim_{t\to\infty} \|a^{\langle t \rangle} - \tilde{a}^{\langle t \rangle}\| = 0$$

For (approximately) linear activation functions, a sufficient condition is that the spectral radius of $W_{\text{hidden}}$ is less than 1. For nonlinear activations such as tanh, ESP can be ensured by requiring that the product of the activation's Lipschitz constant $L$ and the operator norm $\|W_{\text{hidden}}\|$ is strictly less than 1.

**Remark 5.** Echo State Property ensures that the reservoir state asymptotically depends only on the input history, not on the initial state. That is, the reservoir state only reflect the echo of the input, and will not be disrupted by the initial states.

We will now prove the Echo State Property for a linear activation function and nonlinear activation functions. We will show the case for linear activation function first.

*Proof.* We assume that the spectral radius $\rho(W_{\text{hidden}}) < 1$. Consider a simple linear activation function $g_1(x) = x$. Then $a^{\langle t \rangle} = W_{\text{hidden}} a^{\langle t-1 \rangle} + W_{\text{in}} x^{\langle t \rangle}$. Define $\delta^{\langle t \rangle} := a^{\langle t \rangle} - \tilde{a}^{\langle t \rangle}$. Thus, $\delta^{\langle t \rangle} = W_{\text{hidden}} \cdot \delta^{\langle t-1 \rangle}$. Then, by simple iteration, we have $\delta^{\langle t \rangle} = W_{\text{hidden}}^t \cdot \delta^{\langle 0 \rangle}$. If $\rho(W_{\text{hidden}}) < 1$, then $\lim_{t \to \infty} W_{\text{hidden}}^t = 1$. Thus,

$$\lim_{t \to \infty} \|\delta^{\langle t \rangle}\| = \lim_{t \to \infty} \|a^{\langle t \rangle} - \tilde{a}^{\langle t \rangle}\| = 0$$

□

We have concluded the proof for linear activation function. Now, we will consider the proof for non-linear activation functions.

*Proof.* In this case, we will assume that the activation function $g_1$ is Lipschitz continuous and the Lipschitz constant $L < \infty$ and the product of the activation's Lipschitz constant $L$ and the operator norm $\|W_{\text{hidden}}\|$ is strictly less than 1. Consider $\mathcal{F}(a) = g_1(W_{\text{hidden}} a + W_{\text{in}} x)$. We want the system $\mathcal{F}$ be a contraction. Consider

$$\begin{aligned} \|\mathcal{F}(a) - \mathcal{F}(b)\| &= \|g_1(W_{\text{hidden}} a + W_{\text{in}} x) - g_1(W_{\text{hidden}} b + W_{\text{in}} x)\| \\ &= \|g_1(u) - g_1(v)\| \end{aligned}$$

where $c = W_{\text{in}} x$ and $u = W_{\text{hidden}} a + c$ and $v = W_{\text{hidden}} b + c$. Thus, $u - v = W_{\text{hidden}}(a - b)$. Since $g_1$ is Lipschitz continuous, we have

$$\|g_1(u) - g_1(v)\| \leq L \cdot \|u - v\| = L \cdot W_{\text{hidden}}(a - b)$$

Consider the operator norm of $W_{\text{hidden}}(a - b)$, we have

$$\|W_{\text{hidden}}(a - b)\| \leq \|W_{\text{hidden}}\| \cdot \|a - b\|$$

where $\|W_{\text{hidden}}\|$ is the operator norm for $W_{\text{hidden}}$. Putting them together, we have

$$\|\mathcal{F}(a) - \mathcal{F}(b)\| \leq L \cdot \|W_{\text{hidden}}\| \cdot \|a - b\|$$

As we assumed, $L \cdot \|W_{\text{hidden}}\| < 1$, then $\mathcal{F}$ is a contraction. Thus, by the Banach Fixed Point Theorem, we know that for any initial states $a^{\langle 0 \rangle}$, given the contraction iteration $\mathcal{F}$, we have

$$\lim_{t \to \infty} a^{\langle t \rangle} = a^*$$

where $a^*$ is the unique fixed point for the contraction $\mathcal{F}$. Namely, for any two initial states $a^{\langle 0 \rangle}$ and $\tilde{a}^{\langle 0 \rangle}$

$$\lim_{t \to \infty} \|a^{\langle t \rangle} - \tilde{a}^{\langle t \rangle}\| = \lim_{t \to \infty} \|a^* - a^*\| = 0$$

Namely, no matter what the initial state is, $a^{\langle t \rangle}$ would converge to the unique trajectory $a^*$.

□

### 2.5.4 Choice of $W_{\text{hidden}}$

Note from above, the sufficient condition of ESP for linear and general non-linear activation function is different. However, for general non-linear activation, the condition $L \cdot \|W_{\text{hidden}}\| < 1$ is somehow too complicated in practice, especially for training. Therefore, we regard the condition $\rho(W_{\text{hidden}}) < 1$ as a general condition for ESP to hold. For the reservoir computer to run efficiently, the choice of $W_{\text{hidden}}$ is crucial. A common choice of $W_{\text{hidden}} \in \mathbb{R}^{N \times N}$ would be letting

each element in the matrix $(W_{\text{hidden}})_{i,j}$ be i.i.d. $\frac{1}{\sqrt{N}}\mathcal{N}(0,1)$ random variable where $N$ is usually a large number, say 1000. To see why this choice would be good, we would introduce an important theorem.

**Theorem 5** (Circular Law, Girko–Bai–Tao–Vu)**.** [1] [2] [3] Let $W_N \in \mathbb{C}^{N\times N}$ be a random matrix whose entries $\{w_{ij}\}$ are independent and identically distributed (i.i.d.) complex random variables with

$$\mathbb{E}[w_{ij}] = 0, \quad \mathbb{E}[|w_{ij}|^2] = \frac{1}{N}.$$

Define the empirical spectral distribution of $W_N$ as

$$\mu_{W_N} := \frac{1}{N}\sum_{i=1}^{N} \delta_{\lambda_i},$$

where $\lambda_1, \ldots, \lambda_N$ are the eigenvalues of $W_N$, and $\delta_z$ is the Dirac measure at $z \in \mathbb{C}$. Then, as $N \to \infty$, the empirical spectral distribution $\mu_{W_N}$ converges weakly, almost surely, to the uniform distribution $\mu_{\text{circ}}$ on the unit disk:

$$\mu_{W_N} \xrightarrow{a.s.} \mu_{\text{circ}},$$

where $\mu_{\text{circ}}$ is the uniform probability measure on the set $\{z \in \mathbb{C} : |z| \le 1\}$. Moreover, the spectral radius of $W_N$ converges almost surely to 1:

$$\rho(W_N) := \max_{1\le i\le N} |\lambda_i| \xrightarrow{a.s.} 1.$$

Due to the complexity of the proof, we skip it in this thesis. Now, we have noticed that one advantage of the choice $(W_{\text{hidden}})_{i,j}$ be i.i.d. $\frac{1}{\sqrt{N}}\mathcal{N}(0,1)$ is that the spectral radius of $W_{\text{hidden}}$ converges to 1 almost surely. We claim that a choice of $W_{\text{hidden}}$ which ensure $\rho(W_{\text{hidden}}) \approx 1$ would be appropriate. We discuss it in two circumstances:

- If $\rho(W_{\text{hidden}})$ is too small, say $\rho(W_{\text{hidden}}) < 0.5$. Then, the system would decay rapidly over time, i.e. $\|a^{\langle t+1\rangle}\| < \|a^{\langle t\rangle}\|$. Thus, information from earlier time steps is quickly "forgotten" and the reservoir behaves like a narrow sliding window, capturing only the most recent input influences. Therefore, temporal dependencies are poorly preserved, which leads to underperformance in tasks such as speech recognition and financial time series forecasting, or any problem requiring long memory.
- If $\rho(W_{\text{hidden}})$ is too large, say $\rho(W_{\text{hidden}}) > 1.2$. Then, the reservoir dynamics become unstable since $\|a^{\langle t+1\rangle}\| > \|a^{\langle t\rangle}\|$ may lead to potential divergence. Specifically, small perturbations in input or state can grow exponentially, leading to exploding activations. And the reservoir may enter a chaotic regime, where similar inputs produce radically different internal dynamics. Also, output becomes highly unpredictable and it is difficult to train using only the output weights $W_{\text{out}}$.

Therefore, our choice for $W_{\text{hidden}}$ would be stable, and any choice where $\rho(W_{\text{hidden}}) \approx 1$ could be considered and discussed.

## 2.6 Long Short Term Memory (LSTM) Network

### 2.6.1 Introduction to LSTM

LSTM was developed as a specialized variant of RNNs, aimed at overcoming the problem of vanishing gradient and the difficulty of long-term dependency. LSTM addresses this problem by introducing a cell structure that includes forget gates, input gates, cell states, and output gates. The function of each of them are:

- Forget gate: Decides what information to discard from the cell state.
- Update gate: Decides what new information to store.
- Candidate Gate: Generate new candidate memories.
- Output gate: Decides what to output from the current cell.

LSTM has a very general application in time series forecasting like stock price prediction, natural language processing, speech recognition, music generation, etc.

### 2.6.2 Basic Framework of LSTM

In this section, we give the basic mathematical framework of LSTM. At each time step $t$, let $a^{\langle t-1\rangle} \in \mathbb{R}^N$ be the previous hidden state, $c^{\langle t\rangle}$ be the cell state at time $t$ and $x^{\langle t\rangle} \in \mathbb{R}^p$ be the input. Let $\Gamma_f$, $\Gamma_u$, $\Gamma_r$, $\Gamma_o$ denote the output of the forget gate, update gate, candidate gate and output gate respectively. The framework of each gates in LSTM is given by:

**Forget Gate:** In this gate, we decide what information to discard from the cell state. For parameterized matrix $W_f = [W_f^{(a)}, W_f^{(x)}] \in \mathbb{R}^{N\times(N+p)}$ and bias term $b_f \in \mathbb{R}^N$, we have

$$\begin{aligned}\Gamma_f &= \sigma(W_f \cdot [a^{\langle t-1\rangle}, x^{\langle t\rangle}] + b_f) \\ &= \sigma(W_f^{(a)} a^{\langle t-1\rangle} + W_f^{(x)} x^{\langle t\rangle} + b_f) \in \mathbb{R}^N\end{aligned}$$

where $\sigma(x)$ can usually take the sigmoid function. We will keep use the block matrix multiplication notation for simplicity.

**Update Gate:** In this gate, we decide what new information to store. For parameterized matrix $W_u \in \mathbb{R}^{N\times(N+p)}$ and bias term $b_u \in \mathbb{R}^N$, we have

$$\Gamma_u = \sigma(W_u \cdot [a^{\langle t-1\rangle}, x^{\langle t\rangle}] + b_u) \in \mathbb{R}^N$$

**Candidate Gate:** In this gate, we generate new candidate memories. For parameterized matrix $W_r \in \mathbb{R}^{N\times(N+p)}$, a non-linear function $g_1$ (usually be the tanh function) and bias term $b_r \in \mathbb{R}^N$, we have the candidate memory cell as

$$\Gamma_u = \tilde{c}^{\langle t\rangle} = g_1(W_r \cdot [a^{\langle t-1\rangle}, x^{\langle t\rangle}] + b_r) \in \mathbb{R}^N$$

Then, we update new cell state by

$$c^{\langle t\rangle} = \Gamma_f \odot c^{\langle t-1\rangle} + \Gamma_u \odot \tilde{c}^{\langle t\rangle} \in \mathbb{R}^N$$

where $\odot$ denotes the Hadamard product, i.e. the point-wise multiplication between two vectors.

**Output Gate:** In this gate, we decide what to output from the current cell. For parameterized matrix $W_o \in \mathbb{R}^{N\times(N+p)}$ and bias term $b_o \in \mathbb{R}^N$, we have

$$\Gamma_o = \sigma(W_o \cdot [a^{\langle t-1\rangle}, x^{\langle t\rangle}] + b_o) \in \mathbb{R}^N$$

and the new hidden state $a^{\langle t\rangle}$ is

$$a^{\langle t\rangle} = \Gamma_o \odot g_2(c^{\langle t\rangle})$$

where $g_2$ is another nonlinear function (also usually choose tanh function). A clear illustration of LSTM is shown in Figure 10.

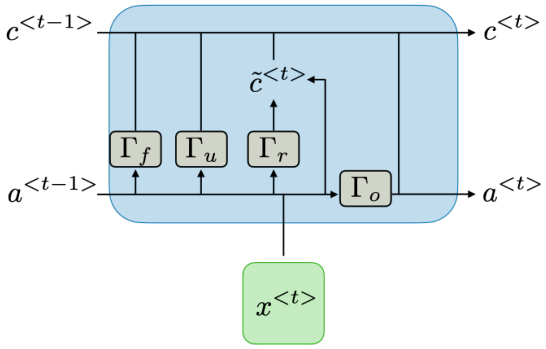



Figure 10: Framework of LSTM

### 2.6.3 Backpropagation Through Time of LSTM

Like the normal RNNs, backpropagation through time (BPTT) is used to compute the gradient with respect to every parameters. More intermediate values are needed because of the LSTM structure. Suppose the loss function $L : \mathbb{R}^T \to \mathbb{R}$ is defined as $(a^{\langle 1\rangle}, \ldots, a^{\langle T\rangle}) \mapsto L((a^{\langle 1\rangle}, \ldots, a^{\langle T\rangle}))$. To compute the gradients of $L$ with respect to the parameters $W_*, b_*$ for each gate $* \in \{\Gamma_f, \Gamma_u, \Gamma_o, \tilde{c}\}$.

Before computing the gate-specific derivatives, we first define an intermediate variable $\delta^{\langle t\rangle}$, which is the gradient of the loss function $L$ with respect to the hidden state $a^{\langle t\rangle}$. In the last time step $t = T$, we have: $\delta_h^{\langle T\rangle} := D_{a^{\langle T\rangle}}(L)$. Despite the last time step, $\delta^{\langle t\rangle}$ contains the recurrent dependency and the dependencies from the other layers, which is defined by

$$\delta_a^{\langle t\rangle} = \epsilon^{\langle t\rangle} + \delta_{\tilde{c}}^{\langle t+1\rangle} W_{\tilde{c}}^{(a)} + \delta_u^{\langle t+1\rangle} W_u^{(a)} + \delta_f^{\langle t+1\rangle} W_f^{(a)} + \delta_o^{\langle t+1\rangle} W_o^{(a)} \quad \text{for } t < T$$

where $\epsilon^{\langle t\rangle}$ is the recurrent dependence at time step $t$. The intermediate variable $\delta^{\langle t\rangle}$ at each gates are

$$\begin{aligned}
\delta_{\Gamma_o}^{\langle t\rangle} &:= D_{a^{\langle t\rangle}}(L) \odot g_2(c^{\langle t\rangle}) \odot \sigma'\left(\Gamma_o^{\langle t\rangle}\right) \\
\delta_c^{\langle t\rangle} &:= D_{a^{\langle t\rangle}}(L) \odot \Gamma_o^{\langle t\rangle} \odot g_2'\left(c^{\langle t\rangle}\right) + \delta_c^{\langle t+1\rangle} \odot \Gamma_f^{\langle t+1\rangle} \\
\delta_{\Gamma_f}^{\langle t\rangle} &:= \delta_c^{\langle t\rangle} \odot c^{\langle t-1\rangle} \odot \sigma'\left(\Gamma_f^{\langle t\rangle}\right) \\
\delta_{\Gamma_u}^{\langle t\rangle} &:= \delta_c^{\langle t\rangle} \odot \tilde{c}^{\langle t\rangle} \odot \sigma'\left(\Gamma_u^{\langle t\rangle}\right) \\
\delta_{\tilde{c}}^{\langle t\rangle} &:= \delta_c^{\langle t\rangle} \odot \Gamma_u^{\langle t\rangle} \odot g_1'\left(\tilde{c}^{\langle t\rangle}\right)
\end{aligned}$$

Let the concatenated input vector be

$$z^{\langle t\rangle} := \begin{pmatrix} a^{\langle t-1\rangle} \\ x^{\langle t\rangle} \end{pmatrix} \in \mathbb{R}^{N+p}$$

And then define the wide matrix as

$$Z^{\langle t\rangle} := \begin{pmatrix} (z^{\langle t\rangle})^T & 0 & \cdots & 0 \\ 0 & (z^{\langle t\rangle})^T & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & (z^{\langle t\rangle})^T \end{pmatrix} \in \mathbb{R}^{N \times (N+p)N}$$

This allows us to express the gradients of the loss function with respect to each gate's parameters compactly. For $* \in \{\Gamma_f, \Gamma_u, \Gamma_o, \tilde{c}\}$, we have:

$$D_{b_*}(\mathcal{L}) = \sum_{t=1}^{T} \delta_*^{\langle t \rangle}$$
$$D_{W_*}(\mathcal{L}) = \sum_{t=1}^{T} \delta_*^{\langle t \rangle} \cdot Z^{\langle t \rangle}$$

The results above is based on Houdt's [5] recent research. The process runs similarly to the standard backpropagation and BPTT. The recursion proceeds backward from $t = T$ to $t = 1$, and once all $\delta_*^{\langle t \rangle}$ are computed, we can evaluate the gradient of the loss function with respect to all the parameters in LSTM.

### 2.6.4 Introduction of Gated Recurrent Unit (GRU) and Comparison with LSTM

GRU is another method to combat the problem of vanishing gradient, enabling the model to capture long-term dependencies in sequential data. Similar to LSTM, GRU uses gates to control the long-term dependencies. While they share a similar motivation and overall structure, there are notable differences in their internal architectures and gating mechanisms. We can generally view GRU as a simplified version of LSTM.

LSTM introduces an explicit memory cell, denoted $c^{\langle t \rangle}$, that is designed to store long-term information. It uses three separate gates to control the flow of information: the forget gate, the input gate, and the output gate. In contrast, GRU simplifies the gating mechanism by merging the memory and hidden state into a single vector, $a^{\langle t \rangle}$, eliminating the need for a separate cell state. It employs only two gates: an update gate and a reset gate. The update gate plays a role similar to a combination of forget and input gates in LSTM and it decides how much of the past information should be kept and how much should be replaced by the new candidate activation. The reset gate determines how to combine the new input with the previous hidden state when generating the candidate activation $\tilde{a}^{\langle t \rangle}$. This simpler structure makes GRU computationally more efficient, with fewer parameters, and generally faster to train.

Despite the structural differences, both LSTM and GRU are capable of modeling long-term dependencies, and the choice between them often depends on the specific task and dataset. In general, LSTM tends to perform slightly better in tasks requiring precise control over long-term memory, while GRU often performs comparably with reduced computational complexity.

## 2.7 Example: MNIST Revisit

In this section, we will revisit the MNIST using different deep learning structure, including FNNs, RNNs, LSTM, GRU and ESN. Similarly to section 1.5, we will use 60000 images to train the model and use 10000 image for testing. For the RNNs structures, we would consider the 784 pixel in each image as a long sequential data. Specifically, we will denotes the pixel in row $i$ and column $j$ in the image as $p_{i,j}$. From the top left pixel $p_{1,1}$, the sequence would be $\{p_{1,1}, \cdots, p_{1,28}, p_{2,28}, \cdots, p_{2,1}, p_{3,1}, \cdots, p_{28,28}\}$, i.e. the pixels are put into a sequence like a snake. For all the five models, we will train two epochs and output the loss for each epoch. We will summarize the key features of each model in the Table 1.

Overall, we note that the loss for the epoch 2 are smaller than the loss for the epoch 1, which means the learning is effective. Comparing the FNNs and RNNs, we notice that the loss and accuracy for the FNNs are smaller than those for the RNNs. FNNs can efficiently extract patterns in this

| Model | Epoch 1 Loss | Epoch 2 Loss | Accuracy |
|---|---|---|---|
| FNNs | 365.9649 | 181.1698 | 95.43% |
| RNNs | 861.5661 | 436.1662 | 88.90% |
| LSTM | 537.2346 | 176.1201 | 95.15% |
| GRU | 639.7835 | 197.5640 | 95.64% |
| ESN | 1351.7092 | 1001.2443 | 71.41% |

Table 1: Loss and accuracy of different neural network models under two Epochs

kind of data without the complexity of memory or recurrence. However, MNIST is not sequential by nature — treating image rows or pixels as a sequence can be unnatural and ineffective. Also, RNNs also suffer from vanishing gradients as the sequence has 784 pixels, which is very long. This will make training less stable and convergence slower.

As LSTM and GRU would resolve the problem of vanishing gradient effectively, we note that the loss for these two models decreases very fast, and the overall accuracies are close to the traditional FNNs. Also, since MNIST is a very simple dataset, the simpler structure GRU would outperform LSTM as its simpler structure matches MNIST better and would make the training more effectively. However, the ESN performs the worst. This is because ESN is not designed for static data. It works well for time series prediction, speech recognition, and tasks involving temporal patterns. But MNIST is composed of static images and feeding images as sequences to an ESN is forced and unnatural.

## 2.8 Example: Japanese Vowels

### 2.8.1 Introduction to Japanese Vowels

In this example, we would use different recurrent neural network structures to train a dataset called Japanese Vowels[4]. The dataset is available on `https://archive.ics.uci.edu/dataset/128/japanese+vowels`. In this dataset, nine male speakers (labeled 0 to 8) uttered two Japanese vowels /ae/ successively. For each utterance, 12-degree linear prediction analysis was applied to obtain a discrete-time series with 12 LPC cepstrum coefficients. This means that one utterance by a speaker forms a time series whose length is in the range 7-29 and each point of a time series is of 12 features (12 coefficients). The number of the time series is 640 in total. We used one set of 270 time series for training and the other set of 370 time series for testing. The goal is to identify the utterance on the test set belongs to whom of the nine male speaker, i.e. it is a nine-category classification problem. We would use four different structures for this example. They are standard RNNs, LSTM, GRU and ESN.

### 2.8.2 Data Pre-processing

We implement a crucial data pre-processing before starting to train the model. The original Japanese Vowels dataset consists of four files: `ae.train`, `ae.test`, `size_ae.train`, and `size_ae.test`. Each line in `ae.train` or `ae.test` represents one frame of speech, encoded as 12-dimensional LPC cepstrum coefficients. Each speaker uttered multiple sequences of the vowels /ae/, and each sequence is stored as a block of 7–29 lines, separated by a blank line.

To convert the data into a format suitable for sequence modeling in PyTorch, we followed the steps below:

- **Segmentation:** Each file (`ae.train` and `ae.test`) was parsed by splitting on blank lines. Each resulting block was treated as one utterance, and converted into a variable-length matrix of shape $(T_i, 12)$, where $T_i$ is the number of frames in the $i$-th utterance.
- **Label assignment:** For training data, it is known that there are 9 speakers, each with exactly 30 utterances, in order. Thus, we assigned labels as: $y_i = \left\lfloor \frac{i}{30} \right\rfloor, \quad i = 0, 1, \ldots, 269$. For testing data, the number of utterances per speaker is given as: $[31, 35, 88, 44, 29, 24, 40, 50, 29]$. Labels were assigned sequentially according to this speaker distribution.
- **Transform to `.npy` file:** Each utterance was stored as a NumPy array with variable time length.
  - `X_train.npy`: List of 270 arrays $(T_i, 12)$
  - `y_train.npy`: Array of shape $(270,)$ with speaker labels
  - `X_test.npy`: List of 370 arrays $(T_i, 12)$
  - `y_test.npy`: Array of shape $(370,)$ with speaker labels

  where the `.npy` file would be easily implemented in model training.

### 2.8.3 Model Training and Analysis

We construct our model in PyTorch in a classic way. Note that the RNNs, LSTM and GRU has built-in modules in PyTorch. The performance of these three models are crucial. Also, We implemented a custom-built ESN for comparison. To make the ESN perform as good as possible, we set each element in the hidden matrix as $(W_{\text{hidden}})_{i,j}$ be i.i.d. $\frac{1}{\sqrt{N}}\mathcal{N}(0, 1)$. Also, we tested the spectral radius from 0.7 to 1.3 and the leaking rate from 0.1 to 0.5 with a step size of 0.05. Finally, we determined the optimal spectral radius to be 0.95 and the optimal leaking rate to be 0.15. We train each model with 50 epochs and plot the training loss and the test accuracy for each model in Figure 11.

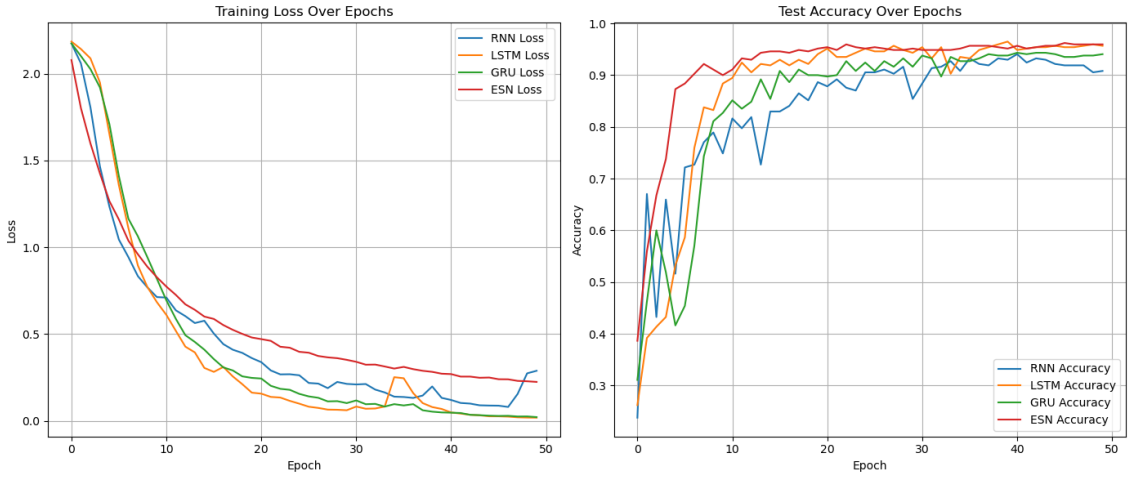



Figure 11: Training Loss and Test Accuracy of Four Models

Overall, the performances of LSTM and ESN are closed, both reaching a test accuracy of 95%, while GRU and standard RNNs reach a reletively low accuracy but still above 90% after 50 epochs.

Note that LSTM outperform the other three models. It is explainable since we have a long sequence of time series, ESN, LSTM and GRU could have better performance when handling the sequence. However, we note that the loss shows a slight decrease and test accuracy shows a sharp increase since problem of vanishing gradient may affect the performance of standard RNNs.

Also, as Japanese Vowel is a classification problem, we plot the confusion matrices of the four models in Figure 12, and we can specifically tell which part of the test set goes wrong. The confusion matrices compare the true label with the predicted label. If a model performs well, the values should be on the diagonal and any values that are not on the diagonal would be regarded as errors.

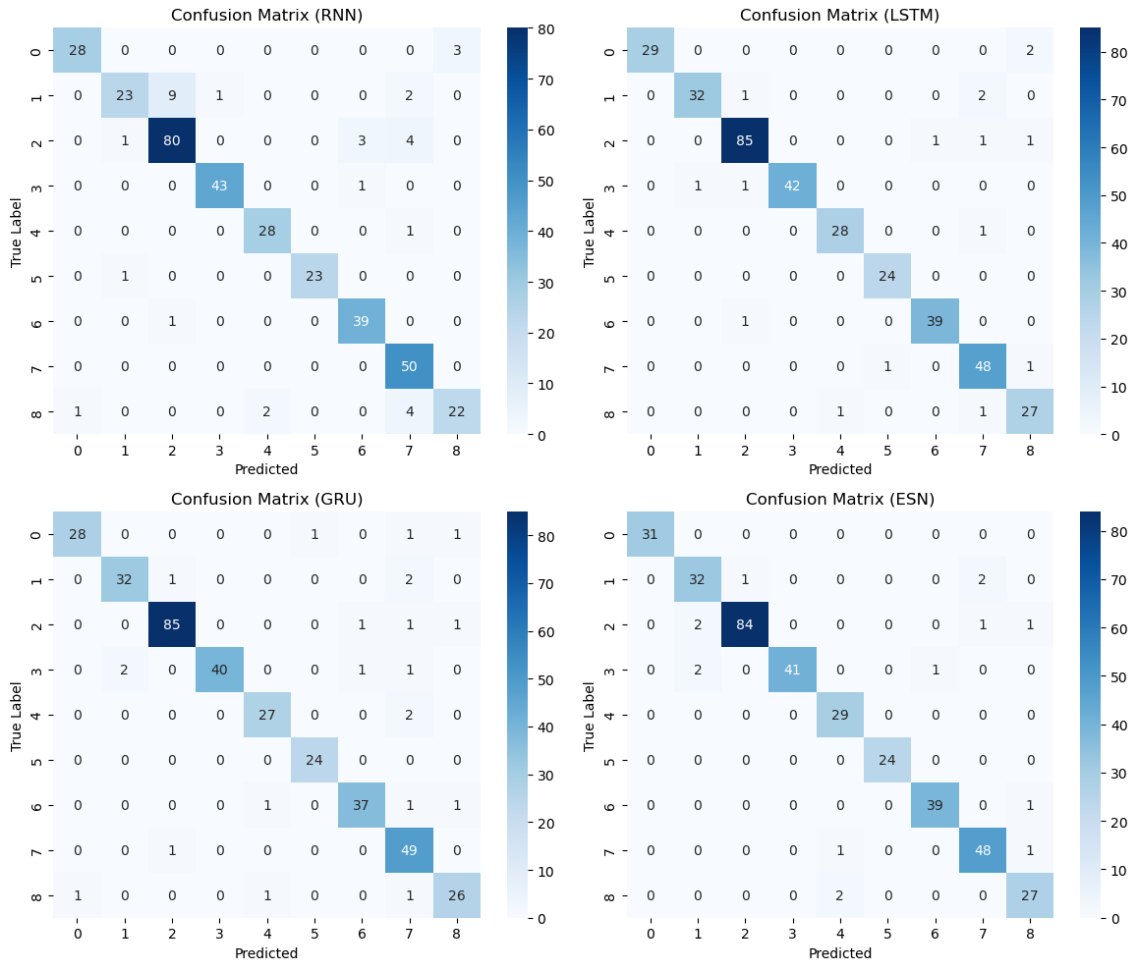



Figure 12: Confusion Matrices of Four Models

From the confusion matrices, we can also notice that the performances of four models are closed. Specifically, the models, especially standard RNNs, would frequently make errors on the person with label 1 and classify him as the person of label 2. Also, they have a better performance that ESN. This is probably because the classification between label 1 and label 2 needs a long time dependency, while standard RNNs may have a worse performance when handling it. Other interpretation from the confusion matrices could be made by readers as well.

## 2.9 Example: S&P500 Index Price Prediction

### 2.9.1 Introduction

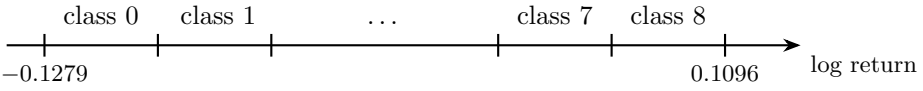
In this section, we would like to predict the distribution of the future close price of S&P500 Index. We would use three different neural network architectures, i.e, RNNs, LSTM and ESN for the prediction. The complete dataset is from 2006-01-01 to 2025-04-10 and we select 13 different finance features from Yahoo Finance and Federal Reserve Economic Data (FRED). They are concluded in the table below.

Table 2: List of potential features for the model

| **Data** | **Source** | **Frequency** | **Abbreviation** |
|---|---|---|---|
| **Fundamental** | | | |
| Open price | Yahoo Finance | Daily | Open |
| Close price | Yahoo Finance | Daily | Close |
| Highest Price | Yahoo Finance | Daily | High |
| Lowest Price | Yahoo Finance | Daily | Low |
| Volume | Yahoo Finance | Daily | Volume |
| **Macroeconomic** | | | |
| Cboe volatility index | Yahoo Finance | Daily | VIX |
| Interest rate | FRED | Daily | EFFR |
| Civilian unemployment rate | FRED | Monthly | UNRATE |
| Consumer sentiment index | FRED | Monthly | UMCSENT |
| US dollar index | Yahoo Finance | Daily | USDX |
| **Technical indicator** | | | |
| Moving average convergence divergence | Yahoo Finance | Daily | MACD |
| Average true range | Yahoo Finance | Daily | ATR |
| Relative strength index | Yahoo Finance | Daily | RSI |

The whole data set is a $4848 \times 13$ matrix where 4848 is the time span in days from 2006-01-01 to 2025-04-10 and 13 is the number of selected features.

Considering the log return of the S&P500 index $\log(\frac{S_{t+1}}{S_t})$ where $S_t$ is the close price of the index at day $t$. Given the log return of the previous dates, we would like to predict the distribution of the log return of the index in the future. Specifically, we would like to construct several bins where the log returns would fall in. Based on the history data, the lowest log return of the index is $-0.12788942$ and the highest log return is 0.10957197. An illustration of the bins of the log return is shown below.




The whole dataset is divided into three parts, which are 65% training set, 15% validation set and 20% test set. After determining the best hyperparameter set on validation set, we merge validation set and training set and start the training officially (80% training set and 20% test set). Specifically, there are 3556 pieces of data in the training set and 889 pieces of data in the test set. Before we train the model, the problem of the partition the bins is a crucial. We will discuss the problem in to next section.

### 2.9.2 Discussion of the Partition of the Bins

There are two ways to partition the bins. They are equal-length partition method and equal-frequency partition method. Specifically, equal-length partition (EL) requires the length of each bin to be equal, while equal-frequency (EF) partition method requires the sample number in each bin to be equal. We list the range of each bin and the number of sample in each bin in the table below.

Table 3: Range and sample number for EL and EF Partition Methods

| **Class** | **EL range & sample number** | | **EF range & sample number** | |
|---|---|---|---|---|
| 0 | $[-0.1279, -0.1013)$ | 1 | $[-0.1279, -0.0113)$ | 539 |
| 1 | $[-0.1013, -0.0749)$ | 6 | $[-0.0113, -0.0052)$ | 538 |
| 2 | $[-0.0749, -0.0486)$ | 16 | $[-0.0052, -0.0021)$ | 539 |
| 3 | $[-0.0486, -0.0222)$ | 153 | $[-0.0021, -0.0002)$ | 538 |
| 4 | $[-0.0222, 0.0041)$ | 3148 | $[-0.0002, 0.0016)$ | 539 |
| 5 | $[0.0041, 0.0305)$ | 1469 | $[0.0016, 0.0037)$ | 538 |
| 6 | $[0.0305, 0.0569)$ | 41 | $[0.0037, 0.0066)$ | 539 |
| 7 | $[0.0569, 0.0832)$ | 8 | $[0.0066, 0.0114)$ | 538 |
| 8 | $[0.0832, 0.1096]$ | 5 | $[0.0114, 0.1096]$ | 539 |

We have tried to train the deep learning model using both of these methods. We note that for the EL method, a huge amount of data concentrates at class 4 and class 5, while less than 5% of the data stays in the other seven classes. The number of sample in each class shows an approximate normal shape. Because of this, the accuracy of the RNNs, ESN and LSTM would converges to 50% since the model would tend to classify the data in the test set in either class 4 or class 5, which shows a relatively poor performance.

However, note that the sample number of the EF methods is approximately the same for each class, and thus, there is enough information for the model to learn in each class. The overall performance for the EF methods is much better than the EL method while some architectures show relatively poor performance (would be discussed later). Therefore, we select the EF partition method for the training.

### 2.9.3 Model Training and Results

In this section, we use the EF partition method for model training. Based on our assumption that the number of total classes is 9, the output of the models would be a vector $\mathbf{y} = (y_1, \cdots, y_9) \in \mathbb{R}^9$, which is called the logits of the model. Then, we will use the softmax function to transform it to a probability distribution $\mathbf{q} = (q_1, \cdots, q_9)$, where

$$q_i = \frac{\exp(y_i)}{\sum_{j=0}^{9} \exp(y_j)}$$

Here $i \in \{0, \cdots, 8\}$. Then, we will calculate the loss between the predicted distribution $\mathbf{q}$ with the true distribution $\mathbf{p}$, where $\mathbf{p} = (0, \cdots, 0, \underbrace{1}_{k}, 0, \cdots, 0)$, if the true label falls in the $k$-th class. We will use the cross-entropy loss function $L$, which is defined as $L = -\sum_{i=0}^{8} p_i \log(q_i)$, to calculate the loss.

Firstly, we will use the RNNs structure for training. After trying several potential hyperparameter, we obtain the optimal hyperparameter. The learning rate is set as 0.001, the batch size is 32 and

the number of hidden layers is 64. For the LSTM architecture, the batch size and learning rate are the same as the RNNs model, but the number of hidden layers is set as 128. For the ESN architecture, we set each element in the hidden matrix as $(W_{\text{hidden}})_{i,j}$ be i.i.d. $\frac{1}{\sqrt{N}}\mathcal{N}(0,1)$ where $N = 1000$. Also, we set the optimal spectral radius to be 0.95 and the optimal leaking rate to be 0.15. Also, the reservoir size is set as 256. The three architectures are trained for 150 epochs, and we use the loss and prediction accuracy (the proportion of predicted class that falls into the true class) to evaluate the performance of each model. The result is shown in the table below.

Table 4: Summary of the performance of three models

| **Model** | **Hyperparameter** | **Loss** | **Accuracy** |
|---|---|---|---|
| RNNs | Learning_rate= 0.001, Batch_size= 32, Hidden_layer = 64 | 27.6766 | 84.96% |
| LSTM | Learning_rate= 0.001, Batch_size= 32, Hidden_layer = 128 | 18.5478 | 88.05% |
| ESN | $(W_{\text{hidden}})_{i,j} \sim \frac{1}{\sqrt{N}}\mathcal{N}(0,1)$, Radius = 0.95, Leaking_rate = 0.15 | 114.5757 | $\approx 22\%$ |

From the table above, we note that both RNNs and LSTM models achieved strong performance in classifying log return intervals. Notably, the LSTM slightly outperformed the RNNs in terms of classification accuracy. This suggests that LSTM is more effective in capturing the relevant patterns in financial time series data, which often exhibit non-linear dynamics and temporal dependencies. While the performance gap between the two models is not large, the LSTM showed more consistent results across multiple runs, indicating higher robustness. Given the noisy and volatile nature of financial data, such consistency is particularly valuable. However, ESN performs worse in this classification task. We would have a detailed discussion of the reasons based on several papers and our observations.

### 2.9.4 Discussion about the Poor Performance of ESN

Compared to RNNs and LSTM, ESN performs significantly worse in this classification task. One possible reason is that under EF partition, the class boundaries are very close, especially for the several classes near 0, which is the center of the distribution. This requires the model to distinguish highly similar input patterns. Unlike RNNs and LSTMs, which adapt their internal representations during training via backpropagation through time, the ESN keeps its reservoir weights fixed and only trains a simple linear readout layer. This limits its ability to adjust to subtle class boundaries and to learn nuanced distinctions between classes. As a result, the ESN tends to collapse predictions toward the more central classes, failing to effectively utilize the full class range.

Recent studies have explored multi-class classification of financial time series using various deep learning architectures. Notably, Zhang et al. [6] divided stock returns into 10 deciles and applied a CNN+LSTM model to capture both local and temporal patterns in S&P500 daily data. Their use of deep architectures allowed the model to learn nonlinear return structures effectively. In contrast, our ESN model, with only a linear readout layer, struggled to distinguish subtle return differences in a 9-class setting.

Similarly, Bai et al.[7] employed a Transformer + MLP (multilayer perceptron) architecture for 5-class and 10-class log return classification on CSI300 index data. They leveraged attention mechanisms and frequency-balanced labels, which are especially suited for class separation. This again contrasts with the ESN, whose randomly-initialized reservoir is not tailored to extract such detailed class distinctions.

Furthermore, Bukhari et al. [8] compared ESN and LSTM in a 3-class cryptocurrency trend

prediction task. While ESN achieved a reasonable accuracy of 61%, it was still outperformed by LSTM (68%), indicating that gated recurrent models may be more capable of capturing nonlinear dependencies in financial time series. Also, a smaller number of classes (here 3 classes) may be more suitable for models like ESN to deal with.

Moreover, we have tested the performance of ESN based on the EL partition, where the distance between each bins are equal and will not be very small or large. We find that the accuracy of ESN quickly converges to 50%, while the convergence of LSTM and RNNs is much slower. Therefore, ESN will have a better performance if the classification task is not very accurate, i.e. the gap between each class is not very small. Also, ESN will perform well if the features of different classes are very distinctively different, like we did in the Japanese Vowel example in section 2.8. Also, based on Zhengyi Guo's [9] thesis, ESN will have a good performance when doing the point prediction, instead of doing the distribution prediction.

In summary, ESN demonstrates significant limitations in handling high-noise multi-class classification tasks such as 9-class log return prediction while it is efficient and fast to train. This is largely due to its fixed reservoir and limited readout capacity. Compared to deep models like LSTM, GRU, or Transformer, ESN lacks the adaptability needed to distinguish subtle differences between classes—especially near the distribution center. Our experiments also confirm that ESN tends to perform better when the class boundaries are well-separated (e.g. under EL partition) or when fewer, more distinct classes are used (e.g. 3-class trend prediction). Additionally, ESN is more suitable for point prediction tasks rather than distribution modeling, and they perform better when input patterns differ significantly, as demonstrated in the Japanese Vowel classification task.

Lastly, the cross-entropy function will penalize the wrong class prediction, even though the error is quite small, especially for the EF partition method, where the gap between each class may be very small. Therefore, a revised version of loss function (instead of simple cross-entropy function) may be sought when doing the prediction.

# 3 Source Code

All source code is available at my GitHub: `https://github.com/AndyLu-web03/Final_Year_Project_Liverpool`

Or you can scan the QR code to access the source code

# References


[1] Z. D. Bai, *Circular Law*, *Ann. Probab.*, vol. 25, no. 1, pp. 494–529, 1997.

[2] T. Tao and V. Vu, *Random matrices: Universality of ESDs and the circular law*, *Ann. Probab.*, vol. 38, no. 5, pp. 2023–2065, 2010.

[3] V. L. Girko, *The circular law*, *Theory Probab. Appl.*, vol. 29, pp. 694–706, 1985.

[4] M. Kudo, J. Toyama, and M. Shimbo, Japanese Vowels [Dataset], UCI Machine Learning Repository, 1999. `https://doi.org/10.24432/C5NS47`.

[5] G. Van Houdt, C. Mosquera, and G. Nápoles, *A review on the long short-term memory model*, *Artif. Intell. Rev.*, vol. 53, no. 8, pp. 5929–5955, Dec. 2020.

[6] Y. Zhang, J. Li, and W. Chen, *Forecasting Stock Returns Using Decile Classification with Deep Learning*, *arXiv preprint arXiv:2006.10825*, 2020.

[7] J. Bai, K. Liu, and Z. Wang, *A Transformer-Based Model for Multi-Class Log Return Classification*, *Proc. IEEE Intl. Conf. on Data Mining*, 2020.

[8] S. Bukhari, A. Mahmood, and R. Khan, *Comparing LSTM and Echo State Networks for Cryptocurrency Trend Prediction*, *Proc. Intl. Conf. on Artificial Intelligence*, 2021.

[9] Z. Guo, *Project on Deep Learning* (Undergraduate thesis). Department of Mathematical Science, University of Liverpool, 2024.