Machine Learning

Yuzhe Zhang

March 2025

0.1 Introduction

In this document, I aim to demonstrate the mathematical principles underlying machine learning algorithms. The document is to record the study path of my Ph.D. in University of Otago.

Chapter 1

Feedforward Network

1.1 Introduction

In this chapter, we derive the formula for the feedforward network.

The feedforward network is the foundation of neural networks, which are considered the backbone of deep learning. The core idea behind many machine learning algorithms can be traced back to linear regression:

$$\hat{y} = ax + b$$

In linear regression, the goal is to find the parameter a that minimizes an error function. There are many ways to define this error function, such as Mean Squared Error (MSE), Mean Absolute Error (MAE), and others, which we will describe in detail later. The most commonly used one is MSE:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{1.1}$$

This equation calculates the average squared distance between the true values and the predicted values. In linear regression, the optimal parameter a minimizes the MSE. This process is referred to as training the model. Once the model is trained, it can make predictions using new input x and the learned parameter a.

In a feedforward network (FFN), the goal is similar to that of linear regression: use a dataset to train a model by finding parameters that minimize the error function. After training, the FFN can make predictions on unseen data.

To construct and train an FFN, several elements and steps are required. First, we consider the input data (which are assumed to have predictive value), the weights (which act similarly to the coefficient a in linear regression), and the bias b, which allows the network to approximate functions that do not pass through the origin:

$$a = Wx + b \tag{1.2}$$

Here, W is the weight matrix and b is the bias vector. At this point, the FFN resembles a linear regression model. The key difference is that FFNs contain multiple layers, each with neurons and non-linear activation functions.

Let us consider a simple two-layer FFN that predicts the next day's S&P 500 index using the past two days' S&P 500 index and gold prices. The input data would look like:

Day	**S&P 500**	**Gold Price**
Day 1	3000	800
Day 2	3100	790

Table 1.1: Simple simulated data.

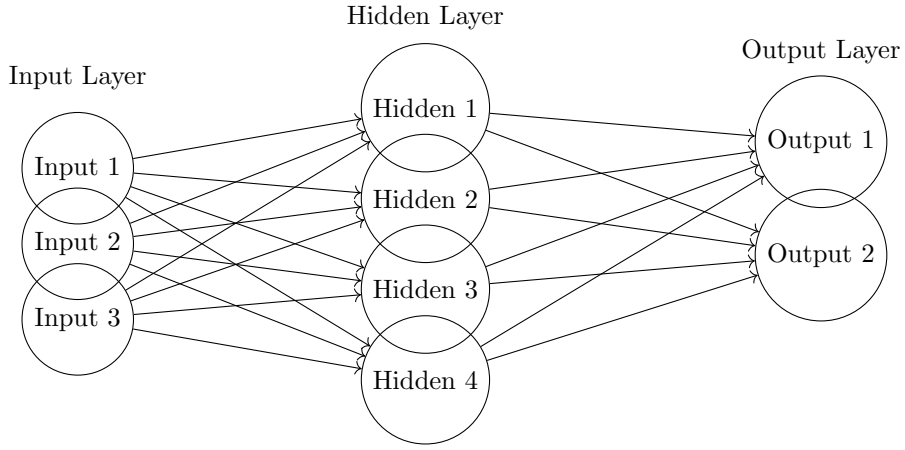
1. The input training data x is flattened into a single vector since FFNs do not incorporate time structure. In this case, the input vector is in $\mathbb{R}^4$.

2. The algorithm initializes weights for each element in the input vector. The weight matrix W has 5 columns (4 for the input features and 1 for the bias), and a number of rows equal to the number of neurons in the next layer.

The FFN consists of multiple layers, neurons (perceptrons), and activation functions. Unlike linear regression, FFNs can model non-linear relationships. This is achieved by applying a non-linear activation function to each neuron's output:

1. In the first layer, compute the activation function output: $\sigma(Wx + b)$. If there are 3 neurons in the first layer, then $W \in \mathbb{R}^{3\times5}$.

2. The output of the first layer becomes the input to the next layer. This process is repeated until the final prediction is generated. The non-linearity of the activation function allows the network to model complex relationships.



This process is called a feedforward network. After the forward pass, it produces a predicted output $\hat{y}$. Similar to linear regression, we aim to minimize the loss function. For example, using mean squared error (MSE), the loss is defined as:

$$L = \frac{1}{2}(\hat{y} - y)^2$$

The factor $\frac{1}{2}$ is included to simplify the derivative during backpropagation. Specifically, when computing the gradient of the loss with respect to the weights W, we get:

$$\frac{\partial L}{\partial W} = (\hat{y} - y) \cdot \frac{\partial \hat{y}}{\partial W}$$

Minimizing the prediction error via gradient descent allows the model to learn the optimal weights.

The weights are updated according to the rule:

$$W \leftarrow W - \eta \frac{\partial L}{\partial W}$$

where η is the learning rate, which is a predetermined hyperparameter in feedforward neural networks (FFNs).

Activation Function	**Expression**	**Output Range**	**Vanishing Gradient?**	**Usage**
Sigmoid	$\frac{1}{1+e^{-x}}$	$(0, 1)$	Yes	Binary classification, early networks
Tanh	$\tanh(x)$	$(-1, 1)$	Yes	Early hidden layers
ReLU	$\max(0, x)$	$[0, \infty)$	No (partially)	Most common hidden layers
Leaky ReLU	$\begin{cases} x & x \geq 0 \\ \alpha x & x < 0 \end{cases}$	$(-\infty, \infty)$	No	Improved ReLU
ELU	$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$	$(-\infty, \infty)$	No	Smoother than ReLU
Swish	$x \cdot \sigma(x)$	$(-\infty, \infty)$	No	Newer networks (e.g. EfficientNet)
Softmax	$\frac{e^{x_i}}{\sum_j e^{x_j}}$	$(0, 1)$, sum=1	N/A	Output layer for classification

Table 1.2: Common Activation Functions and Their Properties

Chapter 2

Recurrent Neural Network

From the previous chapter, we can identify a key limitation of feedforward neural network (FNN)-like algorithms: they treat the input as a static vector and neglect the sequential nature of the data. In other words, these algorithms ignore the temporal dependencies inherent in the input. However, in the financial domain, time plays a pivotal role, as financial data is inherently generated over time. Therefore, to accurately model and predict financial behavior, algorithms that incorporate temporal dynamics are essential.

To address this issue, recurrent-type algorithms are introduced. Unlike feedforward models that process the input as a static vector, recurrent neural networks (RNNs) treat the data as a temporal sequence. For instance, when the input consists of stock closing prices, recurrent models process the data sequentially, receiving one observation at a time, such as one trading day after another.

Given a sequence of inputs $\{x_1, x_2, \ldots, x_T\}$, the recurrent neural network computes a sequence of hidden states $\{h_1, h_2, \ldots, h_T\}$ and outputs $\{y_1, y_2, \ldots, y_T\}$ through the following recurrence relations:

$$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h), \quad t = 1, 2, \ldots, T \tag{2.1}$$

$$y_t = W_{hy}h_t + b_y \tag{2.2}$$

Where:

- $x_t \in \mathbb{R}^n$: input vector at time t
- $h_t \in \mathbb{R}^m$: hidden state vector at time t
- $y_t \in \mathbb{R}^k$: output vector at time t
- $W_{xh} \in \mathbb{R}^{m \times n}$: input-to-hidden weight matrix
- $W_{hh} \in \mathbb{R}^{m \times m}$: hidden-to-hidden weight matrix
- $W_{hy} \in \mathbb{R}^{k \times m}$: hidden-to-output weight matrix

- $b_h \in \mathbb{R}^m$, $b_y \in \mathbb{R}^k$: bias vectors
- $\tanh(\cdot)$: hyperbolic tangent activation function

If the task is a sequence classification (e.g., predicting the label based on the whole sequence), the final prediction can be computed as:

$$\hat{y} = \text{softmax}(W_o h_T + b_o) \tag{2.3}$$

Assuming we use cross-entropy loss for a classification task, the loss function is:

$$\mathcal{L} = -\sum_{i=1}^{K} y_i \log \hat{y}_i \tag{2.4}$$

Backpropagation Through Time

To update the network parameters, we compute the gradients of the loss with respect to each parameter. Specifically, the gradient with respect to the recurrent weight matrix W_{hh} is computed as:

$$\frac{\partial \mathcal{L}}{\partial W_{hh}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial h_t} \cdot \frac{\partial h_t}{\partial W_{hh}} \tag{2.5}$$

Using the chain rule, the gradient of the loss with respect to the hidden state h_t depends not only on the current output but also on all future time steps:

$$\frac{\partial \mathcal{L}}{\partial h_t} = \sum_{k=t}^{T} \left(\frac{\partial \mathcal{L}}{\partial h_k} \cdot \prod_{j=t+1}^{k} \frac{\partial h_j}{\partial h_{j-1}} \right) \tag{2.6}$$

Given the recurrence relation $h_t = \tanh(W_{xh} x_t + W_{hh} h_{t-1} + b_h)$, we have:

$$\frac{\partial h_t}{\partial h_{t-1}} = \text{diag}(1 - h_t^2) \cdot W_{hh} \tag{2.7}$$

Substituting this into the gradient expression, the full derivative involves repeated multiplication of Jacobian matrices:

$$\frac{\partial \mathcal{L}}{\partial h_t} = \sum_{k=t}^{T} \left(\frac{\partial \mathcal{L}}{\partial h_k} \cdot \prod_{j=t+1}^{k} \text{diag}(1 - h_j^2) \cdot W_{hh} \right) \tag{2.8}$$

Vanishing Gradient Problem

If the spectral norm of W_{hh} is less than 1, and $\tanh'$ lies within $(0, 1)$, then the repeated product:

$$\prod_{j=t+1}^{k} \text{diag}(1 - h_j^2) \cdot W_{hh} \tag{2.9}$$

tends to shrink exponentially as $k - t$ increases. This leads to:

$$\left\| \frac{\partial \mathcal{L}}{\partial h_t} \right\| \to 0 \quad \text{as } t \to 1 \tag{2.10}$$

This phenomenon is known as the **vanishing gradient problem**, where the gradients of earlier time steps become negligible, making it difficult for the model to learn long-term dependencies.