Neural Networks

Ruwen Qin
Stony Brook University

October 20, 2025

Data used in this module:

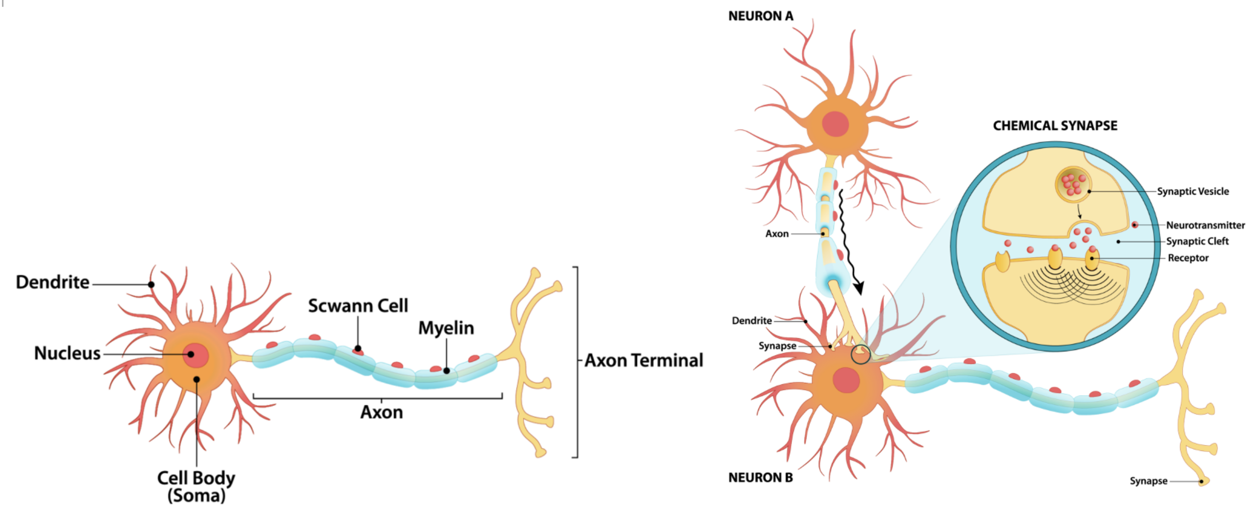
- NNData

Python notebook used in this module

- Neural_Networks.ipynb

1 Introduction

Neural networks are a foundation of modern artificial intelligence. Neural networks can learn from data to approximate nonlinear relationships and perform tasks such as classification, regression, among others. This learning module introduces the fundamental concepts of neural networks, including their architecture and learning process, forming the basis for understanding more advanced deep learning models. This lecture note is mainly adopted from the references [1, 4].



(a) Neuron Anatomy (a) Neuron Communication

Figure 1: Biology Inspiration of Artificial Neural Networks. (Source: https://mhanational.org/resources/neurons-how-the-brain-communicates/

2 Biology Inspiration of Artificial Neural Networks

Artificial neural networks are loosely inspired by the structure and functioning of the human brain, which consists of billions of interconnected nerve cells, or neurons. As Figure 1 illustrates, each biological neuron receives electrical signals from other neurons through dendrites(input terminals), integrates these inputs in its cell body (CPU), and transmits an output signal through its axon (output wire) when the combined input exceeds a certain threshold. Axon terminals (output terminals) enable connections to other neurons or its destination (e.g., an actuator such as a muscle). The connections between neurons, known as synapses, play a crucial role in learning.

This biological mechanism forms the conceptual foundation for artificial neural networks, where artificial "neurons" process input signals, apply activation functions, and adjust connection weights during training to mimic the brain's adaptive learning behavior. Meantime, research in deep learning today draws inspiration from a much wider source, coming in equal or greater measure from mathematics, linguistics, psychology, statistics, computer science, and many other fields.

3 Perceptron

McCulloch and Pitts [2] introduced the first mathematical model of a neuron. Later, Rosenblatt [3] developed perceptron, an early algorithm for classification. A *perceptron* integrates the inputs to produce an output response, as outlined in Figure 2. Mathematically, the perceptron computes a weighted sum of input features and applies an activation function to model the non-linearity of the output signal. Specifically, given an input vector $\mathbf{x} = [x_1, x_2, \ldots, x_N]^{\mathrm{T}}$ and corresponding weights $\mathbf{w} = [w_1, w_2, \ldots, w_N]$ and bias b, the perceptron output is expressed as:

$$\hat{y} = g(\mathbf{w}\mathbf{x} + b), \tag{1}$$

where $g(\cdot)$ is an *activation function*. This formulation serves as a building block for more complex neural network models.

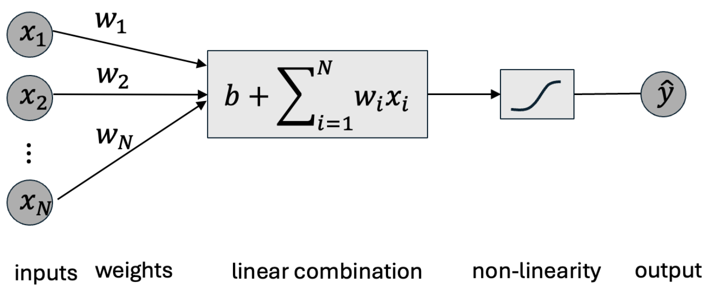


Figure 2: Perceptron

4 Multilayer Perceptron

Multiplayer perception (MLP) extends the perceptron concept by introducing multiple layers of interconnected artificial neurons, mimicking the way biological neurons communicate and process information through layered signaling pathways.

4.1 Network Structure

Figure 3 illustrates an MLP with two hidden layers. The input layer is the input vector with N_I neurons, $\mathbf{x} = [x_1, \ldots, x_{N_I}]^{\mathrm{T}} \in \mathbb{R}^{N_I}$. The first hidden layer $\mathbf{h}^{(1)} = [h_1^{(1)}, \ldots, h_{N_1}^{(1)}]^{\mathrm{T}} \in \mathbb{R}^{N_1}$ has N_1 neurons, constructed as

$$\mathbf{h}^{(1)} = \sigma_1(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}), \tag{2}$$

where $\mathbf{W}^{(1)} \in \mathbb{R}^{N_1 \times N_N}$ and $\mathbf{b}^{(1)} \in \mathbb{R}^{N_1}$ are respectively the weights and biases connecting the input layer and the first hidden layer. σ_1 is the activation function.

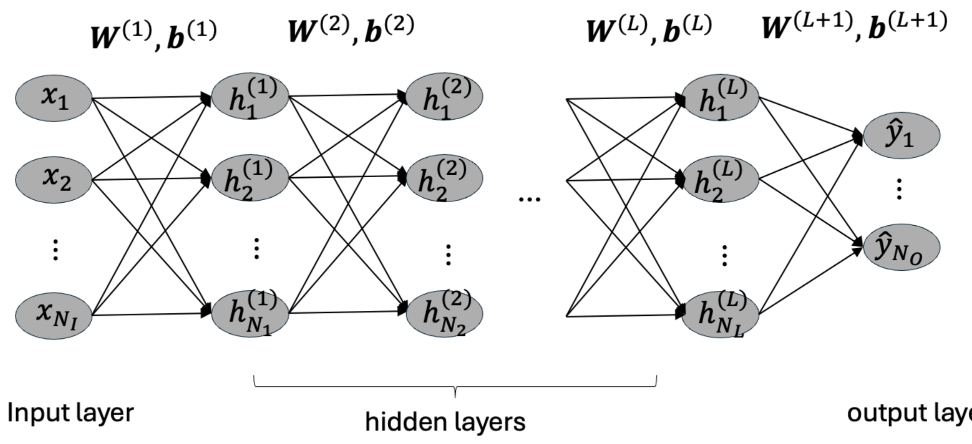


Figure 3: An MLP with two hidden layers

The second hidden layer with N_2 neurons is constructed further:

$$\mathbf{h}^{(2)} = \sigma_2(\mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}), \tag{3}$$

where $\mathbf{W}^{(2)} \in \mathbb{R}^{N_2 \times N_1}$ and $\mathbf{b}^{(2)} \in \mathbb{R}^{N_2}$ are respectively the weights and biases connecting the input layer and the second hidden layer. σ_2 is an activation function. More hidden layers could be added sequentially, making the neural network deeper.

The output layer, consists of N_O outputs, is obtained from the final hidden layer:

$$\hat{\mathbf{y}} = \mathbf{W}^{(L+1)}\mathbf{h}^{(L)} + \mathbf{b}^{(L+1)}, \tag{4}$$

where L is the number of hidden layers, $\mathbf{h}^{(L)} \in \mathbb{R}^{N_L}$ is the last hidden layer, and and $\mathbf{W}^{(L+1)} \in \mathbb{R}^{N_O \times N_{L+1}}$ and $\mathbf{b}^{(L+1)} \in \mathbb{R}^{N_O}$ are respectively the weights and biases connecting the last hidden layer to the output layers.

4.2 Activation Functions

Activation functions decide whether a neuron should be activated or not by calculating the weighted sum and further adding bias to it. They are differentiable operators for transforming input signals to outputs, while most of them add nonlinearity.

4.2.1 ReLU Function

Rectified linear unit (ReLU) is probably the most popular choice, due to both simplicity of implementation and its good performance on a variety of predictive tasks. ReLU provides a very simple nonlinear transformation. Given an input x, the function is defined as the maximum of that element and 0:

$$\text{ReLU}(x) = \max(x, 0), \tag{5}$$

as illustrated in Figure 4.

The derivative of ReLU function is given as:

$$\frac{d}{dx}\text{ReLU}(x) = \begin{cases} 1, & \text{if } x > 0, \\ 0, & \text{if } x \leq 0 \end{cases} \tag{6}$$

The ReLU function is not differentiable when the input takes value precisely equal to 0. In these cases, we default to the left-hand-side derivative and say that the derivative is 0 when the input is 0.

The reason for using ReLU is that its derivatives are particularly well behaved. Either they vanish or they just let the argument through. This makes optimization better behaved and it mitigated the well-known problem of vanishing gradients.

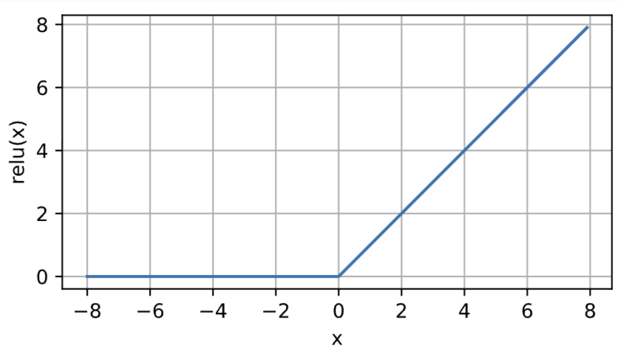


Figure 4: ReLU Function (Source: [4])

4.2.2 Sigmoid Function

The sigmoid function maps inputs whose values lie in the domain $\mathbb{R}$, to outputs that lie on the interval (0, 1) For that reason, the sigmoid is often called a squashing function: it squashes any input in the range (-∞, ∞) to some value in the range (0, 1):

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}, \tag{7}$$

as shown in

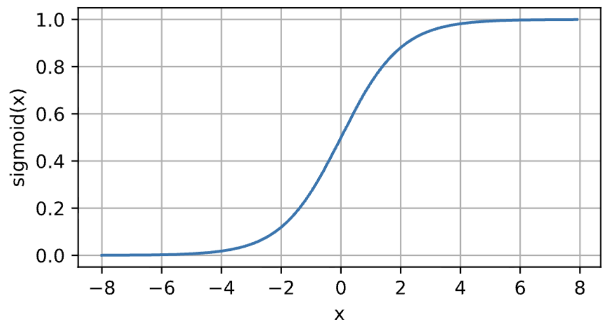


Figure 5: Sigmoid Function (Source: [4])

When attention shifted from thresholding activation (fire or not fire) to gradient-based learning, the sigmoid function was a natural choice because it is a smooth, differentiable approximation to a thresholding unit. Sigmoids are still widely used as activation functions on the output units when we want to interpret the outputs as probabilities for binary classification problems.

The derivative of ReLU function is given below:

$$\frac{d}{dx}\text{sigmoid}(x) = \frac{\exp(-x)}{(1 + \exp(-x))^2} = \text{sigmoid}(x)\left(1 - \text{sigmoid}(x)\right). \tag{8}$$

When the input is 0, the derivative of the sigmoid function reaches a maximum of 0.25. As the input diverges from 0 in either direction, the derivative approaches 0. Therefore, sigmoid poses challenges for optimization since its gradient vanishes for large positive and negative arguments. This can lead to plateaus that are difficult to escape from. Nonetheless sigmoids are important.

4.2.3 Tanh Function

The tanh (hyperbolic tangent) function also maps its inputs in the domain $\mathbb{R}$ to outputs on the interval (0, 1):

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}. \tag{9}$$

The derivative of the tanh function is:

$$\frac{d}{dx} \tanh(x) = 1 - \tanh^2(x). \tag{10}$$

The derivative of the tanh function approaches a maximum of 1. The derivative of the tanh function approaches 0 when the input diverges from 0 to either direction.

Figure 6 shows that, as input nears 0, the tanh function approaches a linear transformation. Although the shape of the function is similar to that of the sigmoid function, the tanh function exhibits point symmetry about the origin of the coordinate system.

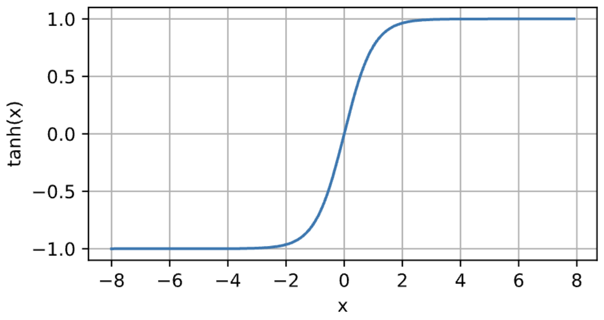


Figure 6: tanh Function (Source: [4])

4.3 Forward Propagation

Forward propagation refers to the calculation and storage of intermediate variables (including outputs) for a neural network in order from the input layer to the output layer.

Let's define the intermediate variables $\mathbf{z}^{(l)}$:

$$\mathbf{z}^{(l)} = \begin{cases} \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}, & \text{when } l = 1 \\ \mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}, & \text{when } l = 2, \ldots, L+1 \end{cases} \tag{11}$$

Then, applying the activation function, σ_l, to the intermediate variable leads to the hidden layer $\mathbf{h}^{(l)}$:

$$\mathbf{h}^{(l)} = \sigma_l(\mathbf{z}^{(l)}). \tag{12}$$

The output layer is:

$$\hat{\mathbf{y}} = \mathbf{z}^{(L+1)}. \tag{13}$$

We can then calculate the loss term l for a single training data point:

$$L = l(\hat{\mathbf{y}}, \mathbf{y}) \tag{14}$$

where $\mathbf{y}$ is the ground truth for the data point.

We can also introduce regularization for learnable parameters

$$s = \frac{\lambda}{2} \sum_{l=1}^{L+1} \|\mathbf{W}^{(l)}\|_F^2, \tag{15}$$

where λ is a hyperparameter for adjusting the strength of the regularization.

Finally, the model's regularized loss on a given training data point is:

$$\mathcal{J} = L + s. \tag{16}$$

4.4 Backpropagation

Backpropagation refers to the method of calculating the gradient of neural network parameters. It traverses the network in reverse order, from the output to the input layer, according to the chain rule from calculus.

In this MLP model, learnable parameters that can be optimized are the weights $\mathbf{W}^{(l)}$ and biases $\mathbf{b}^{(l)}$ for each layer l. The algorithm calculates the gradient with respect to those parameters and stores the intermediate variables required:

$$\begin{aligned}\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(l)}} &= \frac{\partial \mathcal{J}}{\partial L} \cdot \frac{\partial L}{\partial \mathbf{W}^{(l)}} + \frac{\partial \mathcal{J}}{\partial s} \cdot \frac{\partial s}{\partial \mathbf{W}^{(l)}} \\ &= \frac{\partial L}{\partial \mathbf{W}^{(l)}} + \lambda \mathbf{W}^{(l)} \\ &= \frac{\partial L}{\partial \mathbf{z}^{(l)}} \cdot \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{W}^{(l)}} + \lambda \mathbf{W}^{(l)} \\ &= \frac{\partial L}{\partial \mathbf{z}^{(l)}} \cdot \mathbf{h}^{(l-1)} + \lambda \mathbf{W}^{(l)}\end{aligned} \tag{17}$$

$$\begin{aligned}\frac{\partial \mathcal{J}}{\partial \mathbf{b}^{(l)}} &= \frac{\partial \mathcal{J}}{\partial L} \cdot \frac{\partial L}{\partial \mathbf{b}^{(l)}} + \frac{\partial \mathcal{J}}{\partial s} \cdot \frac{\partial s}{\partial \mathbf{b}^{(l)}} \\ &= \frac{\partial L}{\partial \mathbf{b}^{(l)}} \\ &= \frac{\partial L}{\partial \mathbf{z}^{(l)}} \cdot \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{b}^{(l)}} \\ &= \frac{\partial L}{\partial \mathbf{z}^{(l)}}\end{aligned} \tag{18}$$

In both gradient calculations above, the partial derivative $\partial L/\partial \mathbf{z}^{(l)}$, which is named *error term of* layer l, needs to be calculated:

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{z}^{(l)}} &= \frac{\partial L}{\partial \mathbf{h}^{(l)}} \cdot \frac{\partial \mathbf{h}^{(l)}}{\partial \mathbf{z}^{(l)}} \\ &= \frac{\partial L}{\partial \mathbf{z}^{(l+1)}} \cdot \frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{h}^{(l)}} \odot \sigma_l'(\mathbf{z}^{(l)}) \\ &= \frac{\partial L}{\partial \mathbf{z}^{(l+1)}} \cdot \mathbf{W}^{(l+1)} \odot \sigma_l'(\mathbf{z}^{(l)})\end{aligned} \tag{19}$$

were $\odot$ means pairwise product. That is because the activation function σ_l applies elementwise.

If we denote the error term for each layer as:

$$\delta^{(l)} = \frac{\partial L}{\partial \mathbf{z}^{(l)}}, \tag{20}$$

then, the backpropagation process is the following process:

For each layer, working backward from $l = L + 1$ to 1:

1. Calculate Error Term $\delta^{(l)}$:

When $l = L + 1$

$$\delta^{L+1} = \frac{\partial L}{\partial \hat{\mathbf{y}}} \tag{21}$$

When $l = L, L-1, \ldots, 1$:

$$\delta^{(l)} = \delta^{(l+1)} \cdot \mathbf{W}^{(l+1)} \odot \sigma_l'(\mathbf{z}^{(l)}), \tag{22}$$

2. Calculate Gradient:

With respect to the weights:

$$\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(l)}} = \delta^{(l)} \cdot \mathbf{h}^{(l-1)} + \lambda \mathbf{W}^{(l)}, \tag{23}$$

for $l = L+1, \dots, 2$; and

$$\frac{\partial \mathcal{J}}{\partial \mathbf{W}^{(l)}} = \delta^{(l)} \cdot \mathbf{x} + \lambda \mathbf{W}^{(l)}, \tag{24}$$

when $l = 1$.

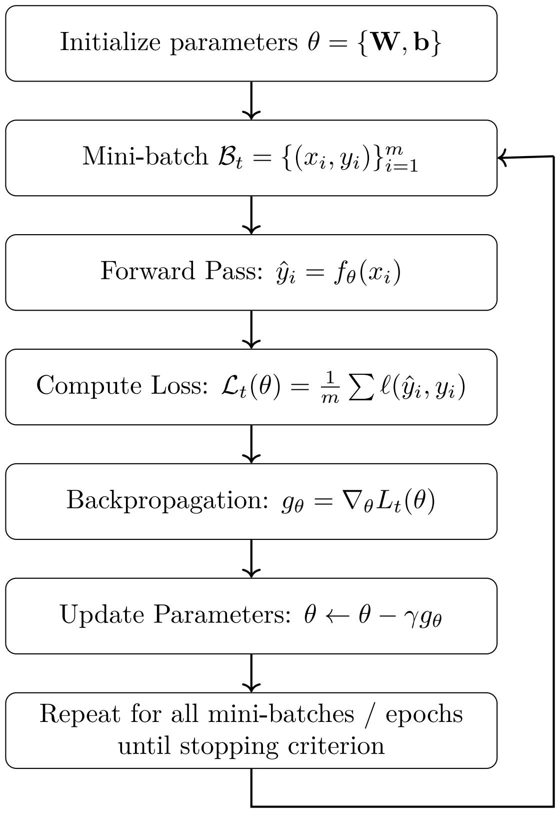
With respect to biases:

$$\frac{\partial \mathcal{J}}{\partial \mathbf{b}^{(l)}} = \delta^{(l)} \tag{25}$$

for any l.

4.5 Model Training

When training neural networks, once model parameters are initialized, we alternate forward propagation with backpropagation, updating model parameters using gradients given by backpropagation. Note that backpropagation reuses the stored intermediate values from forward propagation to avoid duplicate calculations. One of the consequences is that we need to retain the intermediate values until backpropagation is complete. This is also one of the reasons why training requires significantly more memory than plain prediction. Besides, the size of such intermediate values is roughly proportional to the number of network layers and the batch size.



Initialize parameters $\theta = \{\mathbf{W}, \mathbf{b}\}$
Mini-batch $\mathcal{B}_t = \{(x_i, y_i)\}_{i=1}^{m}$
Forward Pass: $\hat{y}_i = f_\theta(x_i)$
Compute Loss: $\mathcal{L}_t(\theta) = \frac{1}{m} \sum \ell(\hat{y}_i, y_i)$
Backpropagation: $g_\theta = \nabla_\theta L_t(\theta)$
Update Parameters: $\theta \leftarrow \theta - \gamma g_\theta$
Repeat for all mini-batches / epochs until stopping criterion

4.6 Overfitting and Regularization

The difference between our fit on the training data and our fit on the test data is called the generalization gap and when this is large, we say that our models overfit to the training data. In extreme cases of overfitting, we might exactly fit the training data, even when the test error remains significant.

Regularization, either by reducing the model complexity or by applying a penalty to constraint the set of values that our parameters might take.

4.6.1 Early Stopping

Early stopping is a classic technique for addressing the overfitting issue. It constrains the number of epochs of training. A common way is to monitor validation error throughout training and to cut off training when the validation error is less than some small amount for some number of epochs. This is sometimes called a patience criterion.

4.6.2 Dropout

Dropout is a regularization technique used in neural networks to prevent overfitting. As Figure 7 shows, during training, a fraction of neurons in each layer is randomly "dropped out" (set to zero) on each iteration, which forces the network to avoid relying on specific neurons. This can be viewed as training a large ensemble of subnetworks. During inference, all neurons are used, and no scaling is required in most cases.

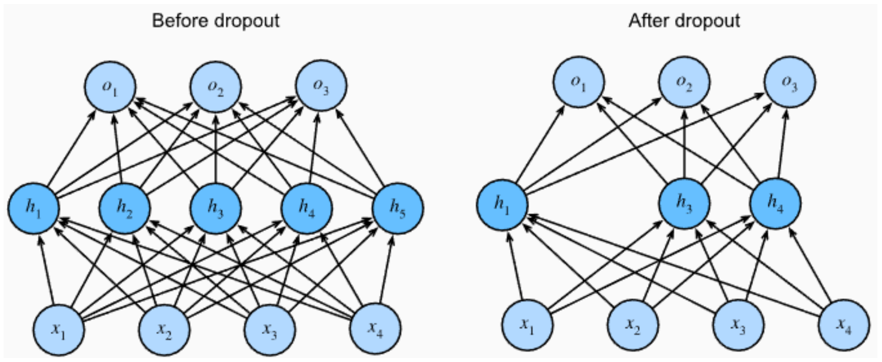


Figure 7: MLP before and after dropout (Source: Fig. 5.6.1 in [4])

Dropout is related to the idea of smoothness and robustness: by injecting noise into the network, the model is encouraged to be less sensitive to small perturbations in inputs or internal activations, improving generalization. It is widely used and a simple yet effective way to make networks more resilient to overfitting.

References

[1] Alexander Amini and Ava Amini. Mit introduction to deep learning. https://introtodeeplearning.com/, 2025. © Alexander Amini and Ava Amini, Massachusetts Institute of Technology.

[2] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[3] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[4] Aston Zhang, Zachary C Lipton, Mu Li, and Alexander J Smola. *Dive into deep learning*. Cambridge University Press, 2023.