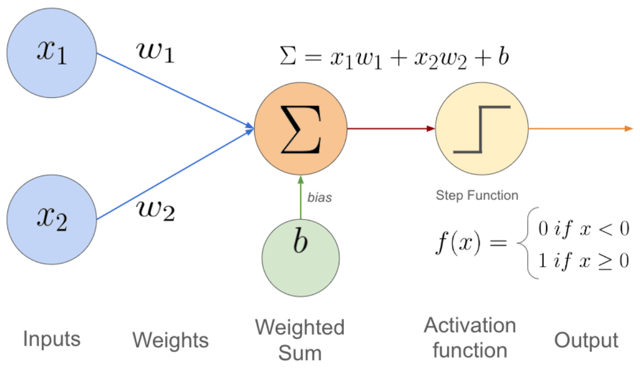
Lecture 04

Neural Network

May 23, 2025

1 Perceptron

The perceptron is the simplest form of neural network. It performs binary classification that maps input features to output values in 1s and 0s.



x_1
w_1
$\Sigma = x_1 w_1 + x_2 w_2 + b$
$\sum$
x_2
w_2
bias
b
Step Function
$f(x) = \begin{cases} 0 \; if \; x < 0 \\ 1 \; if \; x \geq 0 \end{cases}$
Inputs
Weights
Weighted Sum
Activation function
Output

Figure 1: Perceptron AND Gate with step function [10123]

1.1 Perceptron Problem

Perceptrons have limitations, whereby single-layer perceptrons can only learn linearly separable patterns, which means that they can only solve problems where data can be divided by a straight line or plane. An example of the perceptron problem is the **XOR problem**.

2 Feed-forward Neural Network

A feed-forward neural network consists of one or more nodes in hidden layers. Every nodes in a layer is linked to every node in the next layer, and multiple activation functions can be used in the same neural network.

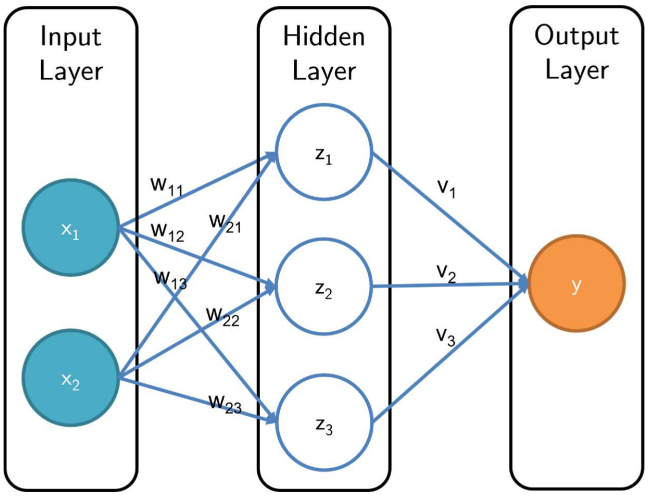
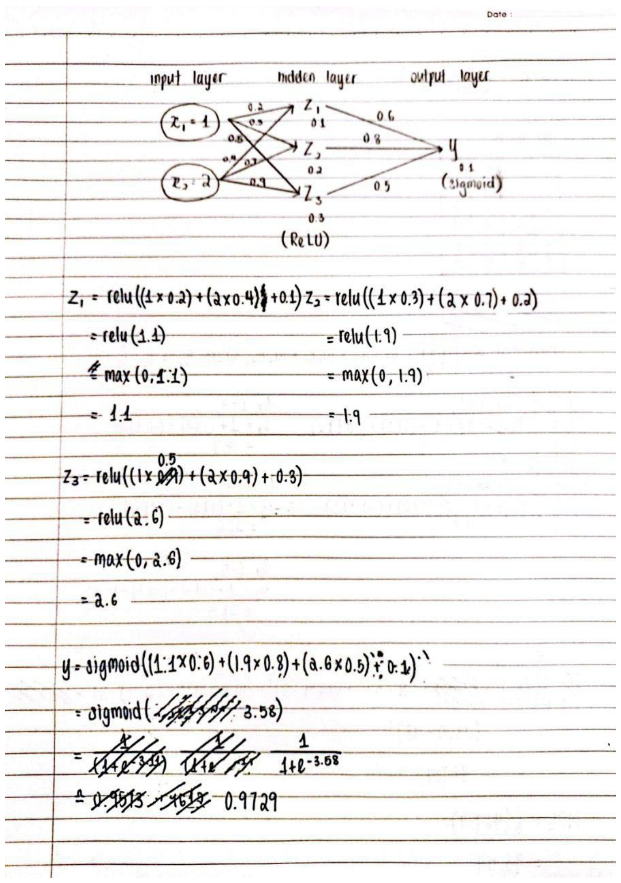


Figure 2: Feed-forward Neural Network Diagram

2.1 Simple Calculation of Feed-forward Neural Network



Date :
input layer
hidden layer
output layer
$x_1 = 1$
$x_2 = 2$
0.2
0.3
0.5
0.4
0.7
0.9
Z_1
0.1
Z_2
0.2
Z_3
0.3
(ReLU)
0.6
0.8
0.5
y
0.1
(sigmoid)

$Z_1 = \text{relu}((1 \times 0.2) + (2 \times 0.4) + 0.1)$ $\quad Z_2 = \text{relu}((1 \times 0.3) + (2 \times 0.7) + 0.2)$

$= \text{relu}(1.1)$ $\quad = \text{relu}(1.9)$

$= \max(0, 1.1)$ $\quad = \max(0, 1.9)$

$= 1.1$ $\quad = 1.9$

$Z_3 = \text{relu}((1 \times \overset{0.5}{\cancel{0.9}}) + (2 \times 0.9) + 0.3)$

$= \text{relu}(2.6)$

$= \max(0, 2.6)$

$= 2.6$

$y = \text{sigmoid}((1.1 \times 0.6) + (1.9 \times 0.8) + (2.6 \times 0.5) + 0.1)$

$= \text{sigmoid}(\cancel{[illegible]}\ 3.58)$

$= \cancel{\frac{1}{(1+e^{-3.44})}}\ \cancel{\frac{1}{(1+e^{-3.5})}}\ \frac{1}{1+e^{-3.58}}$

$\doteq \cancel{0.9513}\ \cancel{.4613}\ 0.9729$

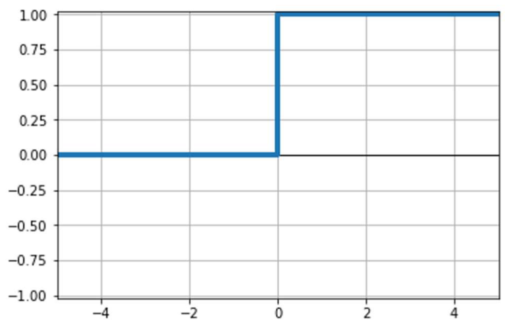
3 Activation Functions

Activation functions introduce non-linearities in neural network models, making them able to detect complex patterns from datasets. All nodes in the same neural network uses the same activation function. Let's take a look at a few of them.

3.1 Threshold Function

The threshold function is the simplest type of activation function. They are used in models that uses **simple logic-based**, where binary decisions are sufficient. It is rarely used in modern deep learning models.

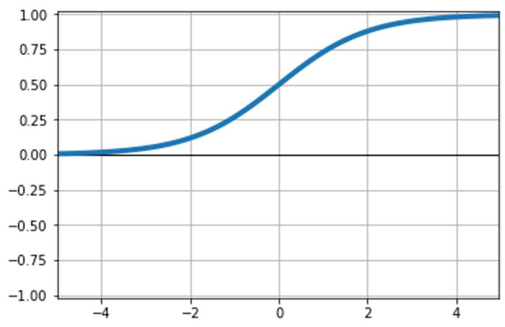
$$threshold(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$



3.2 Sigmoid Function

The sigmoid function is used in **binary classification problems** like logistic regression. It is suitable for neural network models where vanishing gradient is not a concern.

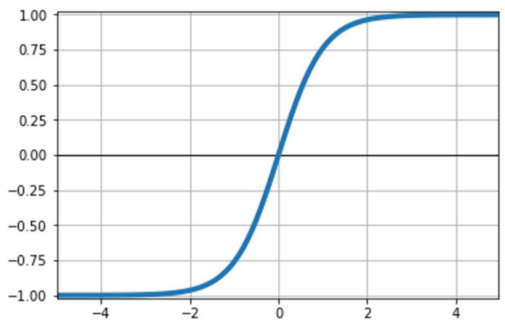
$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$



3.3 Hyperbolic Tangent (tanh) Function

The hyperbolic tangent function is used when there are zero-centered outputs (outputs in the range of -1 to 1) are present in neural network models. However, it suffers from vanishing gradients.

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

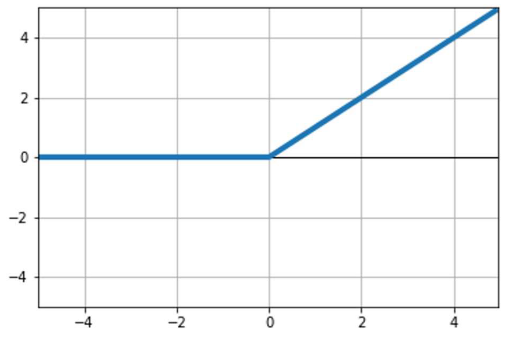


3.4 Rectifier Linear Unit Function

The Rectifier Linear Unit (ReLU) function is the default choice for the hidden layers in modern deep learning models like CNNs, RNNs and transformers. This activation function enables **faster model training** and has **less vanishing gradient**.

The ReLU function encourages **sparse activation**, where only a subset of neurons are only used at a time, which improves model efficiency. However, using this activation function can get neurons stuck at outputting zero if the weights become negative, this is known as the **dying ReLU problem**.

$$relu(x) = \max(0, x)$$



4 Backpropagation in Neural Networks

Backpropagation is the opposite of feed-forward. It involves nodes in the same neural network layer taking the error values of nodes from the subsequent layer to run the gradient descent algorithm and update their weights, that aims to minimize the loss function. The backpropagation algorithm helps neural network models to improve their accuracy and performance.

The loss function used to calculate error values of the nodes depends on the type of task the neural network model is solving. Mean Squared Error will be used to solve **regression tasks** while cross-entropy will be used to solve **classification tasks**.

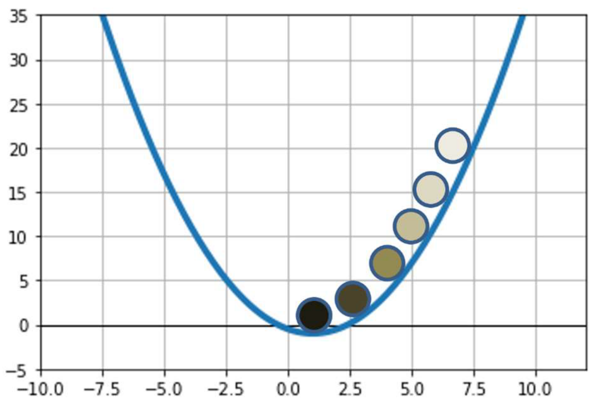
$$\textbf{Mean Square Error: } J = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

$$\textbf{Cross-Entropy: } J = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right)$$

4.1 Gradient Descent

In the backpropagation section, we mentioned gradient descent. It is an optimization algorithm where the set of weight parameters converges to a point where the loss function *J(w)* is minimal.

$$w_j = w_j - \eta\frac{\partial J}{\partial w_j}, \quad \text{where} \quad \frac{\partial J}{\partial w_j} = -\frac{2}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)x_{ij}$$



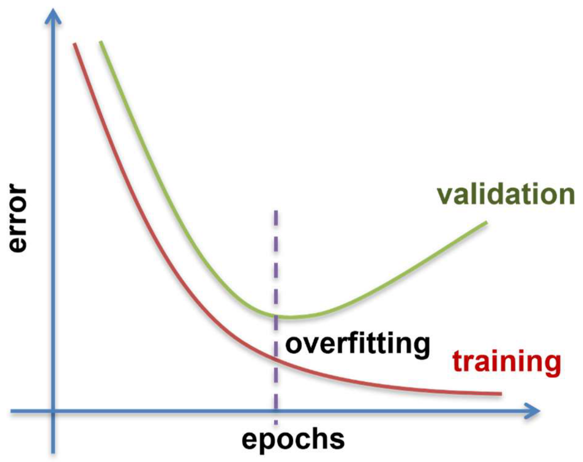
We can introduce learning rate α to the gradient descent algorithm. The learning rate allows us to control the aggressiveness of updates to the weight parameters. If the learning rate is set to too high, the gradient descent will find it hard to converge, or even diverge or oscillate. On the other hand, if the learning rate is set to too low, the gradient descent may take forever to converge, or it will be stuck at a poor local minima.

A well-chosen learning rate ensures that our machine learning model is able to converge smoothly to a minimum of the loss function.

$$w_j = w_j - \eta(\alpha)\frac{\partial J}{\partial w_j}, \quad \text{where } \alpha = \text{learning rate}$$

5 Overfitting

Overfitting occurs in machine learning models when its **training data accuracy is extremely high** while the **prediction accuracy on external datasets is low**. It indicates that the model is prone to noise after it is being trained. This phenomenon is common in deep learning models with high numbers of hidden nodes and layers.



We can overcome model overfitting by **reducing its complexity**. One way to do so is to reduce the complexity of the dataset used to train deep learning models. Principal Component Analysis is used to simplify the dataset while retaining most of the information from the dataset.

Other than that, we can utilize regularization techniques like **dropout**, which randomly deactivates a subset of neurons in a neural network in every training iteration. This technique reduces dependency of neurons on other ones, to learn redundant and more robust features on their own, which makes the entire model more tolerant to variability in inputs.

There are two other types of regularization techniques: **Lasso regularization** and **Ridge regularization**.

In Lasso regularization, the sum of absolute values **(L1 norm)** of the model's weights is added to the loss function.

$$\mathcal{L}_{\text{lasso}} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 + \alpha\sum_{j=1}^{M}|w_j|$$

What the regularization term $\alpha\sum_{j=1}^{M}|w_j|$ does is it encourages the model to **shrink some weights exactly to zero**, which indirectly does feature selection for us as well. Lasso regularization should be used in **sparse models**, where features are not highly correlated.
In Ridge regularization, the sum of squares **(L2 norm)** of the model's weights is added to the loss function instead.

$$\mathcal{L}_{\text{ridge}} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 + \alpha\sum_{j=1}^{M}w_j^2$$

What the regularization term $\alpha\sum_{j=1}^{M}w_j^2$ does is it **penalizes large weights without forcing them to zero**, which **reduces the model complexity, preventing overfitting**. Ridge regularization should be used in models with features that are highly correlated to other features, or to stabilize linear models. Unlike Lasso regularization, all model features remain in the model.

We can prevent overfitting by splitting the provided datasets into training and testing datasets for model evaluation.

6 Cross-validation

Cross-validation is a process in machine learning model development to choose the best performing models by comparing multiple models with different settings of hyperparameters like **number of neurons in hidden layers**, **learning rate**, and **batch size**.

6.1 K-fold Cross-validation

K-fold cross-validation is a commonly used type of cross-validation used in machine learning. To conduct the k-fold cross-validation, we first split our dataset into **k** equally-sized folds. In each of **k** iterations, we use **k-1** folds for training and use the remaining one for validation. We will rotate the fold used for validation in every iteration, every fold is used once for validation.

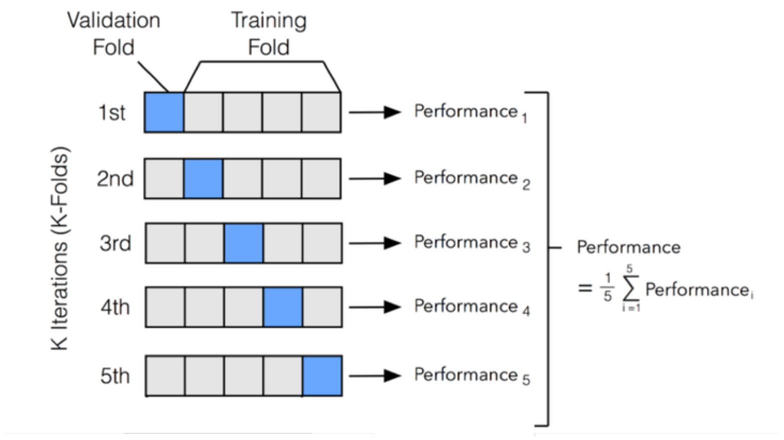


Figure 3: [She20]

Lastly, we will average the performance metrics used (e.g., accuracy, loss, F1-score) across all **k** runs so that we get a reliable estimate of our model performance. K-fold cross-validation allows us to detect the model is overfitting when it performs well on training folds but badly on validation folds.

References

[10123] 101 AI. Perceptron - neural network visualizer. http://101ai.net/nnet/perceptron, 2023. Accessed: 2025-05-19.

[She20] Zitao Shen. Machine learning 101: Cross validation. https://zitaoshen.rbind.io/project/machine_learning/machine-learning-101-cross-vaildation/, 2020. Accessed: 2025-05-23.