CS 412 — Introduction to Machine Learning (UIC)	April 17, 2025
Lecture [24]	
Instructor: Aadirupa Saha	*Scribe(s): (Yukta Salvi)*

[This draft is not fully proofread. Please email any typos/errors to the instructor or directly edit the latex file.]

Overview

In the last lecture, we covered the following main topics:

1. Feed-Forward NN
2. Backpropagation

This lecture focuses on:

1. Backpropagation (contd)
2. Recurrent Neural Network (RNN)
3. Convolutional Neural Network (CNN)

1 Backpropagation (contd)

Neural network learning is a type of supervised learning, where a model is trained on example inputs along with their corresponding correct outputs (labels). Neural networks are extensively used for a wide range of tasks, particularly in classification and regression problems.

One of the most fundamental and commonly used architectures is the Feedforward Neural Network (FNN).

A Feedforward Neural Network (FNN) is an artificial neural network where information moves in a single direction—from the input layer, through one or more hidden layers, and finally to the output layer. The term feedforward reflects the fact that data flows forward without forming any cycles or feedback loops. Additionally, connections only exist between adjacent layers—no connections skip layers, and no layer receives input from a subsequent layer.

1.1 Single Output Feed-Forward Neural Network

The figure shows a single-output Feedforward Neural Network (FFNN). In this architecture, the single output represents the value of the function for a given input. This setup is suitable for binary classification tasks.

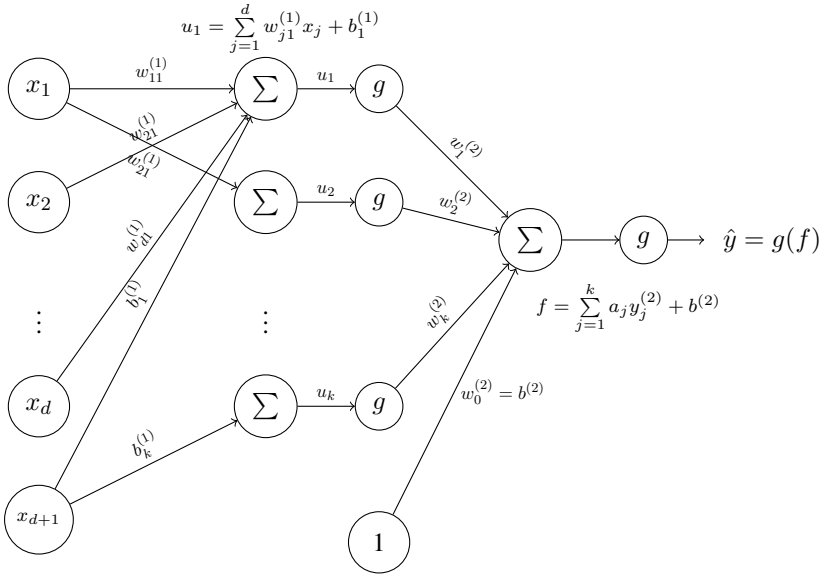


Figure 1: Single Feed-Forward Neural Network

1.2 Multi Output Feed-Forward Neural Network

For more complex tasks such as multi-class or multi-label classification—where the input X could be a subset of data like images, emails, or suitable candidates—a single-output FFNN is insufficient. In such cases, a multi-output FFNN with multiple output nodes is used to perform multi-class or multi-label classification.

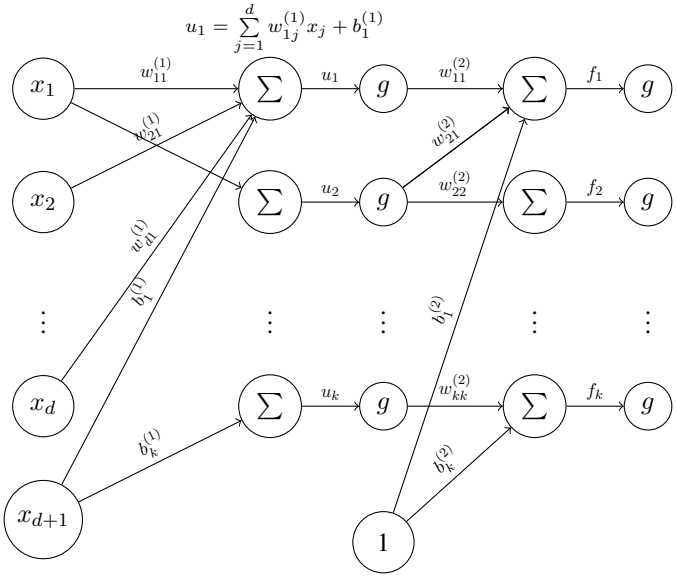


Figure 2: Multi-Output Feed-Forward Neural Network

The output is compared to a predefined threshold, and if it exceeds this threshold, the input is classified as belonging to the target class.

Note: Difference between Multi-class and Multi-label Classification

Multi-class Classification: In this setting, each input example X is mapped to a *single class* out of C possible classes. The function can be defined as:

$$f : X \to \{1, 2, \ldots, C\}$$

For example, handwritten digit recognition—where the task is to identify which digit (0 through 9) is written in a given image—is a multi-class classification problem.

Multi-label Classification: Here, each input can be associated with *multiple labels simultaneously*. The function can be defined as:

$$f : X \to 2^C$$

where 2^C denotes the power set of the class set C. For example, multi-label classification is used in identifying and labeling multiple objects or features within an image, like recognizing both "cat" and "outdoor" in a photograph.

1.3 Backpropagation Algorithm

Backpropagation is a fundamental technique used in deep learning to train artificial neural networks, particularly feedforward networks.

The objective of machine learning involves the optimization of the chosen loss function. With every epoch, the machine "learns" by adapting the weights and biases to minimize the loss.

In order to minimize the cost function, one must determine which weights and biases to adjust. Computing the gradient with respect to the parameters can help us do just that. With each epoch, the machine converges towards the local minimum.

Backpropogation Algorithm:

Algorithm 1.1: Backpropagation Algorithm

1: Initialize weights and biases randomly
2: **for** each training example (x, y) **do**
3: **Forward pass:** Compute activations layer by layer from input to output
4: **Compute loss:** $L = \text{Loss}(\hat{y}, y)$
5: **Backward pass:** Compute gradients of loss with respect to each weight and bias using the chain rule
6: **Update parameters:**
7: $w \leftarrow w - \eta \frac{\partial L}{\partial w}$
8: $b \leftarrow b - \eta \frac{\partial L}{\partial b}$
9: **end for**

Backpropagation is typically combined with optimization algorithms such as gradient descent. The algorithm uses the chain rule to update the weights of the network by calculating the gradient of the loss function with

respect to the weights. This allows it to efficiently propagate error information backward through the network layers. In reverse, this algorithm is better known as Backpropagation. Backpropagation is recursively done through every single layer of the neural network.

1.4 Chain rule of differentiation

i) Initialization

We initialize the parameters of the neural network at time step $t = 0$:

$$W^{(1)} \in \mathbb{R}^{k \times d}, \quad W^{(2)} \in \mathbb{R}^{1 \times k}, \quad b^{(1)} \in \mathbb{R}^{k}, \quad b^{(2)} \in \mathbb{R}$$

These parameters are typically initialized randomly.

ii) Forward Pass

Given an input $x \in \mathbb{R}^d$ and label $y \in \{0, 1\}$:

- Compute the pre-activation for hidden layer: $u = W^{(1)}x + b^{(1)}$
- Apply activation: $a = g(u)$
- Compute the output pre-activation: $f = W^{(2)}a + b^{(2)}$
- Apply output activation: $\hat{y} = g(f)$

iii) Loss Function

We use the binary cross-entropy loss:

$$\mathcal{L}(y, \hat{y}) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

iv) Backward Pass (Gradient Computation)

For each training example $t = 1, \ldots, T$, compute gradients w.r.t. all parameters:

$$\frac{\partial \mathcal{L}}{\partial W^{(2)}}, \quad \frac{\partial \mathcal{L}}{\partial b^{(2)}}, \quad \frac{\partial \mathcal{L}}{\partial W^{(1)}}, \quad \frac{\partial \mathcal{L}}{\partial b^{(1)}}$$

by propagating the gradients backward through the network using the chain rule.

1.5 Gradient Descent Update for $\frac{\partial \mathcal{L}}{\partial W^{(2)}}$

After computing the forward pass and loss, we update the weights of the network using gradient descent.

Weight Update Rule: The gradient descent update rule for the weights $W^{(2)}$ is:

$$W_{ij}^{(2)}(t+1) = W_{ij}^{(2)}(t) - \eta \frac{\partial \mathcal{L}}{\partial W_{ij}^{(2)}}\Big|_t$$

Gradient Computation via Chain Rule: We compute the gradient using the chain rule:

$$\frac{\partial \mathcal{L}}{\partial W^{(2)}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial f} \cdot \frac{\partial f}{\partial W^{(2)}}$$

Intermediate Derivatives:

1. The derivative of the binary cross-entropy loss:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \frac{d}{d\hat{y}} \left(-y \log \hat{y} - (1-y) \log(1-\hat{y})\right) = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}$$

2. The derivative of the activation function $g(f)$, assuming g is the sigmoid function:

$$\frac{\partial \hat{y}}{\partial f} = \frac{d}{df} g(f) = g(f)(1-g(f)) = \hat{y}(1-\hat{y})$$

3. The derivative of the affine transformation:

$$\frac{\partial f}{\partial W^{(2)}} = \frac{\partial}{\partial W^{(2)}} (W^{(2)} a + b) = a$$

Final Expression: Putting everything together:

$$\frac{\partial \mathcal{L}}{\partial W^{(2)}} = \left(-\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}\right) \cdot \hat{y}(1-\hat{y}) \cdot a$$

1.6 Computing $\frac{\partial \mathcal{L}}{\partial b^{(2)}}$

Given the output of the network:

$$f = W^{(2)} a + b^{(2)}, \quad \hat{y} = g(f)$$

$$\mathcal{L}(y, \hat{y}) = -y \log(\hat{y}) - (1-y) \log(1-\hat{y})$$

Using the chain rule:

$$\frac{\partial \mathcal{L}}{\partial b^{(2)}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial f} \cdot \frac{\partial f}{\partial b^{(2)}}$$

We compute:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}, \quad \frac{\partial \hat{y}}{\partial f} = \hat{y}(1-\hat{y}), \quad \frac{\partial f}{\partial b^{(2)}} = 1$$

Putting it all together:

$$\frac{\partial \mathcal{L}}{\partial b^{(2)}} = \left(-\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}\right) \cdot \hat{y}(1-\hat{y}) = \hat{y} - y$$

1.7 Computing $\frac{\partial \mathcal{L}}{\partial W^{(1)}}$

Given a two-layer neural network with:

$$
\begin{aligned}
u &= W^{(1)}x + b^{(1)} \\
a &= g(u) \\
f &= W^{(2)}a + b^{(2)} \\
\hat{y} &= g(f)
\end{aligned}
$$

and the binary cross-entropy loss function:

$$\mathcal{L}(y, \hat{y}) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

We aim to compute:

$$\frac{\partial \mathcal{L}}{\partial W^{(1)}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial f} \cdot \frac{\partial f}{\partial a} \cdot \frac{\partial a}{\partial u} \cdot \frac{\partial u}{\partial W^{(1)}}$$

Step-by-step derivation:

- $\frac{\partial \mathcal{L}}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}$
- $\frac{\partial \hat{y}}{\partial f} = \hat{y}(1 - \hat{y})$ (assuming sigmoid activation)
- $\frac{\partial f}{\partial a} = W^{(2)}$
- $\frac{\partial a}{\partial u} = a(1 - a)$ (element-wise)
- $\frac{\partial u}{\partial W^{(1)}} = x$

Putting it all together:

$$\frac{\partial \mathcal{L}}{\partial W^{(1)}} = \left(\frac{y}{\hat{y}} - \frac{1 - y}{1 - \hat{y}} \right) \cdot \hat{y}(1 - \hat{y}) \cdot W^{(2)} \cdot a(1 - a) \cdot x^T$$

1.8 Computing $\frac{\partial \mathcal{L}}{\partial b^{(1)}}$

Recall the forward computations:

$$
\begin{aligned}
u &= W^{(1)}x + b^{(1)} \\
a &= g(u) \\
f &= W^{(2)}a + b^{(2)} \\
\hat{y} &= g(f)
\end{aligned}
$$

The loss function is the binary cross-entropy loss:

$$\mathcal{L}(y, \hat{y}) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

We compute the gradient with respect to the hidden layer bias $b^{(1)}$:

$$\frac{\partial \mathcal{L}}{\partial b^{(1)}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial f} \cdot \frac{\partial f}{\partial a} \cdot \frac{\partial a}{\partial u} \cdot \frac{\partial u}{\partial b^{(1)}}$$

Step-by-step:

- $\frac{\partial \mathcal{L}}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}$
- $\frac{\partial \hat{y}}{\partial f} = \hat{y}(1 - \hat{y})$ (assuming sigmoid)
- $\frac{\partial f}{\partial a} = W^{(2)}$
- $\frac{\partial a}{\partial u} = a(1 - a)$ (element-wise sigmoid derivative)
- $\frac{\partial u}{\partial b^{(1)}} = 1$

Putting it together:

$$\frac{\partial \mathcal{L}}{\partial b^{(1)}} = \left(\frac{y}{\hat{y}} - \frac{1 - y}{1 - \hat{y}} \right) \cdot \hat{y}(1 - \hat{y}) \cdot W^{(2)} \cdot a(1 - a)$$

2 Recurrent neural Network

2.1 Introduction

Recurrent neural networks (RNNs) are a class of deep learning models fundamentally designed to handle sequential data. Unlike feedforward neural networks, RNNs possess the unique feature of maintaining a memory of previous inputs by using their internal state (memory) to process sequences of inputs. This makes them ideally suited for applications such as natural language processing, speech recognition, and time series forecasting, where context and the order of data points are crucial.

RNNs have been used extensively for text-generation tasks. In sentiment analysis, RNNs have been shown to outperform traditional models by capturing the context and details of sentiment expressed in text. Below shows a multi-class classification task for sentiment analysis.

- Input: A document with **variable length**, represented as a sequence of features of length d.
- Output: One of multiple classes.

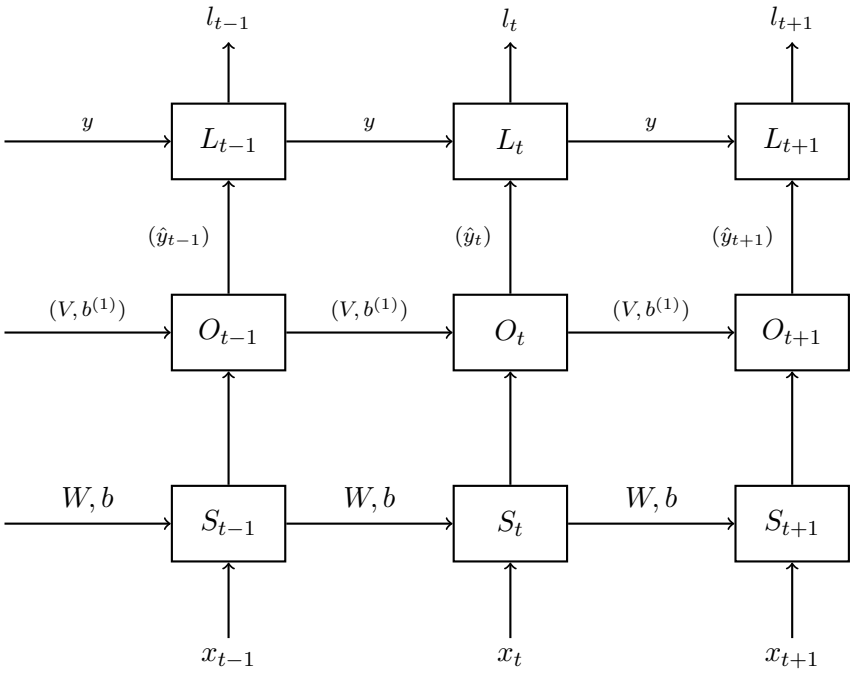
Given a document, the RNN processes the sequence and outputs a sentiment class: joyful, melancholic, or sad.

Comparison

- **FFNN** is better when the input has fixed size and order is not important.
- **RNN** is better when the input sequence is of variable length and order matters.

2.2 Basic Architecture of RNN

An RNN processes sequential input data by maintaining a hidden state that gets updated at each time step. The diagram below represents the recurrent neural network (RNN) unrolled over time steps $t-1$, t, and $t+1$.



- S Layer (State Layer): Takes input x_t and parameters (W,b) to compute hidden states at each time step.
- O Layer (Output Layer): Transforms the hidden state using parameters $(V, b^{(1)})$ to compute intermediate output representations.
- L Layer (Loss Layer): Computes the final output ℓ_t, comparing the predicted $\hat{y}_t$ to the actual label y.

Example

Input sentence: `Alice likes brownies.`
Feed words to the network:

$$x^{(1)} = \text{"Alice"}, \quad x^{(2)} = \text{"likes"}, \quad x^{(3)} = \text{"brownies"}$$

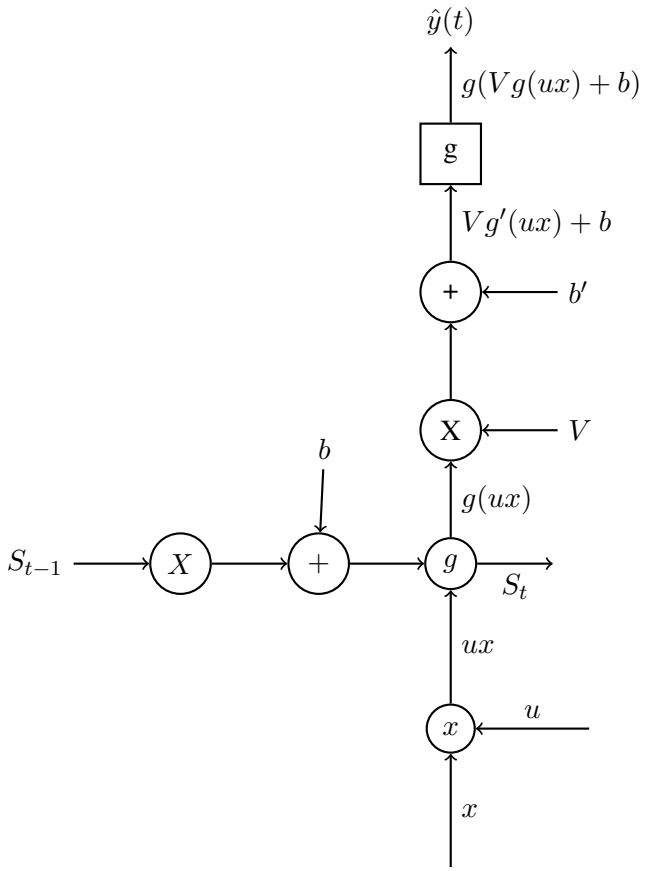
Another sentence: `Alice was excited to see chocolate.`
Each word is fed sequentially into the network as a vector representation (embedding), and the network learns patterns based on word order and context.

2.3 Limitations

RNNs are often perceived as "black-box" models due to their complex internal dynamics, making it challenging to interpret their decisions.

RNNs require large amounts of high-quality, labeled sequential data for effective training. Such data may be scarce, noisy, or incomplete in many real-world scenarios and can lead to overfitting when trained on small datasets.

2.4 Example



The above figure shows how an RNN works at a given instance t.

It takes the input x_t and processes it using shared weights u to compute ux, which is passed through an activation function g to produce the hidden state S_t. This hidden state is combined with the previous state S_{t-1} and bias b to form an intermediate result. The output layer further transforms this through a weighted operation with parameter V and bias b', eventually producing the predicted output $\hat{y}_t = g(Vg(ux) + b)$.

The figure below represents a multi-layered computational graph with outputs propagating forward through intermediate variables $S^1 \to S^2 \to S^3 \to \hat{y}$.

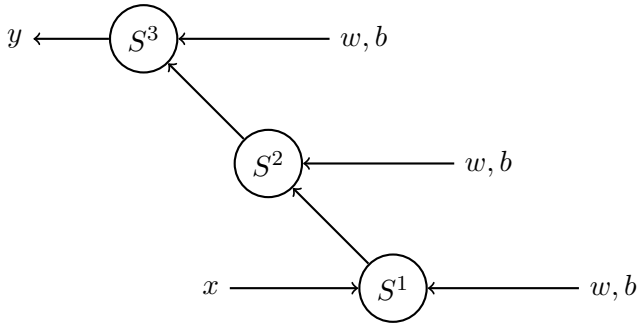


Figure 3: Computational graph showing flow from S^1 to output x, with parameters w used at each stage for T=3.

This diagram shows how a Recurrent Neural Network (RNN) works when it's unrolled over 3 time steps — that is, when you look at how it processes inputs at times t=1,2,3.

The derivative will look like this:

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial S^3} \cdot \frac{\partial S^3}{\partial w} + \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial S^3} \cdot \frac{\partial S^3}{\partial S^2} \cdot \frac{\partial S^2}{\partial w} + \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial S^3} \cdot \frac{\partial S^3}{\partial S^2} \cdot \frac{\partial S^2}{\partial S^1} \cdot \frac{\partial S^1}{\partial w}$$

3 Convolution Neural Network

One of the largest limitations of traditional ANN forms is that they struggle with the computational complexity required to compute image data. Larger, colored image, like one that's 64 by 64 pixels with three color channels (RGB), a single neuron would need to process 12,288 weights (64 × 64 × 3), just in the first layer.

And since the network needs to be bigger to handle these more detailed images, it becomes clear why using simple neural networks (like fully connected ones) is not ideal for large image data. This is one of the reasons why Convolutional Neural Networks (CNNs) are used instead — they are better at handling high-dimensional image data efficiently.

CNNs are primarily used to solve difficult image-driven pattern recognition tasks and with their precise yet simple architecture.

Why Not Just Make the Neural Network Bigger?

Short answer: No — and here's why:

- **It's expensive:** We don't have unlimited computing power or time to train huge networks. Bigger models take longer to train and require more resources.
- **It causes overfitting:** A larger model might memorize the training data too well — including noise and irrelevant details. This means it performs poorly on new or unseen data.
- **Overfitting is a common problem:** It makes the model unable to generalize, which is critical for real-world tasks and test datasets.
- **Simpler is often better:** Smaller networks with fewer parameters are less likely to overfit, and they can actually perform better in practice.

3.1 Convolution Operation on Matrices

Convolution is a fundamental mathematical operation performed by sliding the kernel M_2 over the input matrix M_1 and computing the **element-wise product sum** at each location. It is used in various fields such as signal processing, image processing, and neural networks. It involves two functions, (f) and (g), and produces a third function that represents how the shape of one is modified by the other. The convolution of (f) and (g) is denoted by (f * g). IIt acts as the mathematical filters that help computers find edges of images, dark and light areas, colors, and other details, such as height, width, and depth.

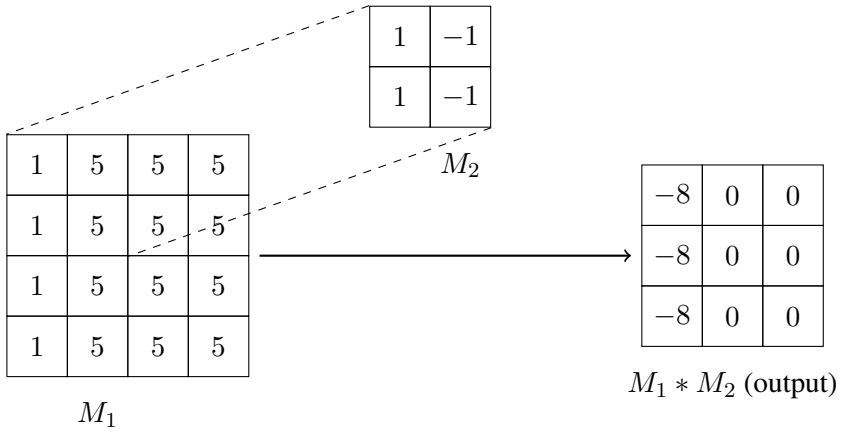
Mathematical Expression

$$(M_1 * M_2)[i, j] = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} M_1[i + m, j + n] \cdot M_2[m, n]$$

Where: - M_1 is the input image matrix - M_2 is the kernel (e.g., edge detector) - (i, j) indexes the position in the output matrix - $k \times k$ is the kernel size

Example: Vertical Edge Detection Kernel

A commonly used vertical edge detection kernel is:



This negative result indicates a sharp change from left to right, which is a vertical edge!

1. Output Size Calculation

We are performing a **valid convolution** with:

- Input matrix M_1 of size 4×4
- Filter/kernel M_2 of size 2×2
- Stride = 1
- No padding (i.e., "valid" convolution)

The formula for output dimensions is:

$$\text{Output Rows} = n - f + 1, \quad \text{Output Cols} = n - f + 1$$

where:

- $n = 4$ (size of input)
- $f = 2$ (size of kernel)

$$\Rightarrow \text{Output size} = (4 - 2 + 1) \times (4 - 2 + 1) = 3 \times 3$$

2. Value Computation and Final Matrix

Kernel:

$$M_2 = \begin{bmatrix} 1 & -1 \\ 1 & -1 \end{bmatrix}$$

At each position in the output matrix, we:

1. Select a 2×2 patch from M_1
2. Perform element-wise multiplication with the kernel M_2
3. Sum the resulting values to get a scalar output

Example 1 – Top-left cell:

$$\text{Patch from } M_1 = \begin{bmatrix} 1 & 5 \\ 1 & 5 \end{bmatrix}, \quad (1)(1) + (5)(-1) + (1)(1) + (5)(-1) = 1 - 5 + 1 - 5 = -8$$

Example 2 – Center cell:

$$\text{Patch from } M_1 = \begin{bmatrix} 5 & 5 \\ 5 & 5 \end{bmatrix}, \quad (5)(1) + (5)(-1) + (5)(1) + (5)(-1) = 5 - 5 + 5 - 5 = 0$$

Final Output Matrix:

$$\begin{bmatrix} -8 & 0 & 0 \\ -8 & 0 & 0 \\ -8 & 0 & 0 \end{bmatrix}$$

Note: To preserve the original input size after convolution, **zero-padding** is commonly applied in practice.

When to Use RNN or CNN?

CNN is better suited for image and video processing.
RNN, on the other hand, performs better on sequential data like text, speech, and time series — since it retains memory of previous inputs, unlike CNNs.

Summary of Applications:

- **CNNs are commonly used for:**
  - Image data
  - Video data
- **RNNs are commonly used for:**
  - Text data
  - Speech data
  - Time series data
  - Generative models (e.g., text generation)

Conclusion: CNNs dominate in computer vision tasks, while RNNs are essential for natural language processing and other sequential tasks.

Next Lecture

The next lecture will cover the following topics:
(i) CNN,
(ii) Dropout regularization,
(iii) Vanishing Exploding Gradients,
(iv) RNN-LSTM.

References:

1. Lecture 13: Recurrent Neural Networks Lecturer: Swaprava Nath source
2. Convolutional Neural Networks cheatsheet source
3. Lecture note on Notes on Multilayer, Feedforward Neural Networks by Prof. Lynne E. Parker from CS494/594: Projects in Machine Learning Spring 2006 source
4. Recurrent Neural Networks: A Comprehensive Review of Architectures, Variants, and Applications by Ibomoiye Domor Mienye, ORCID,Theo G. Swart, ORCID, and George Obaido source
5. An Introduction to Convolutional Neural Networks by Keiron O'Shea1 and Ryan Nash2 source