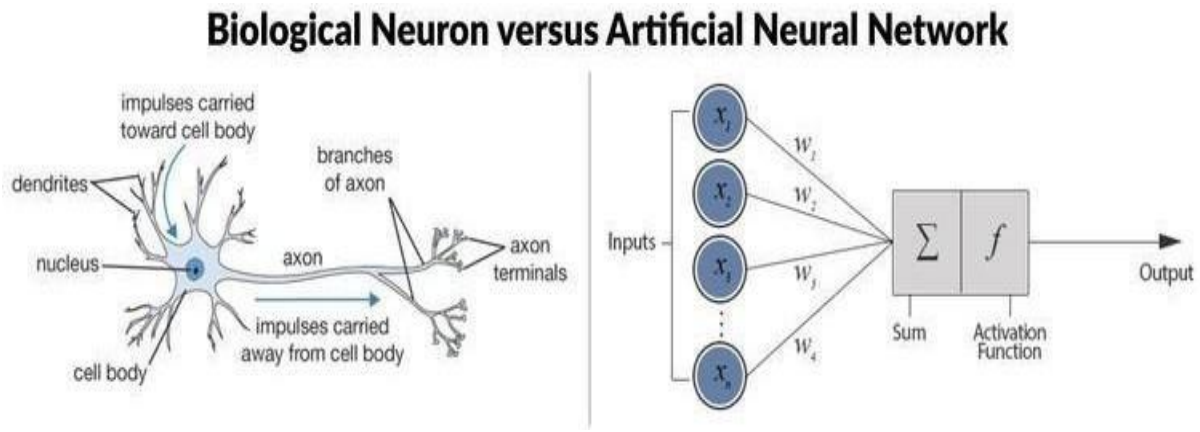
DEEP LEARNING

HANDOUT: ARTIFICIAL NEURAL NETWORKS

1. Introduction to ANN

An Artificial Neural Network (ANN) is a machine learning model inspired by the structure and functioning of the human brain. It consists of layers of interconnected nodes (neurons) that process and learn patterns from data. The purpose of ANN is to learn from examples and make predictions or decisions based on input data.

In biological systems, neurons send signals via synapses. Similarly, in ANN, artificial neurons receive input, perform calculations using weights and biases, and produce output via activation functions. ANNs are used because they can model complex and non-linear relationships that traditional algorithms struggle with.



Checkpoint Question

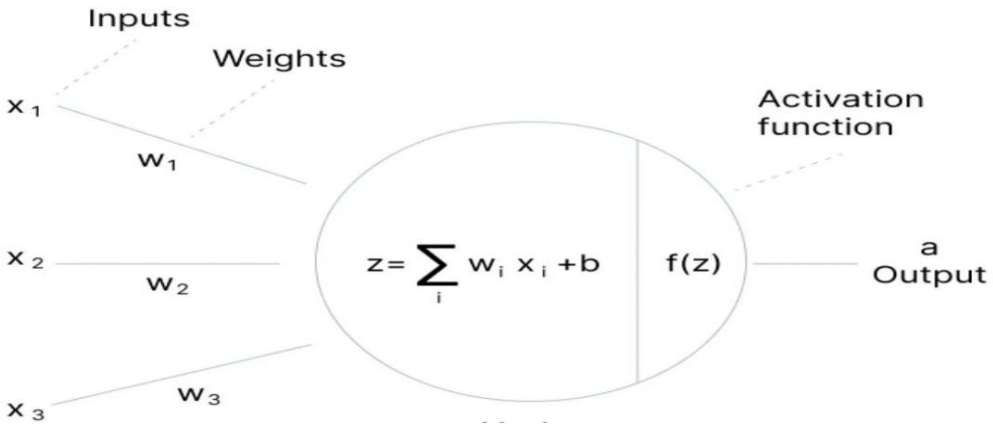
Scenario: A weather app needs to predict tomorrow's rainfall based on temperature, humidity, and pressure readings.

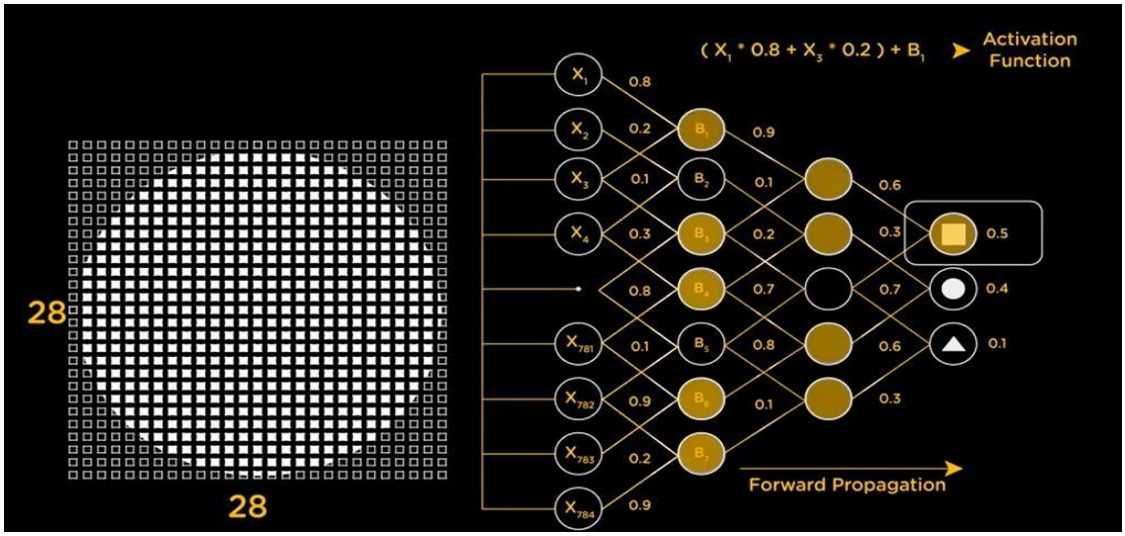
> Why might an ANN be more suitable than simple if-else rules for this task?
>
> Answer: Because ANN can learn complex, non-linear relationships between temperature, humidity, and pressure that are hard to capture using fixed rules. It adapts and generalizes better than static rule-based systems.

Understanding the Basics

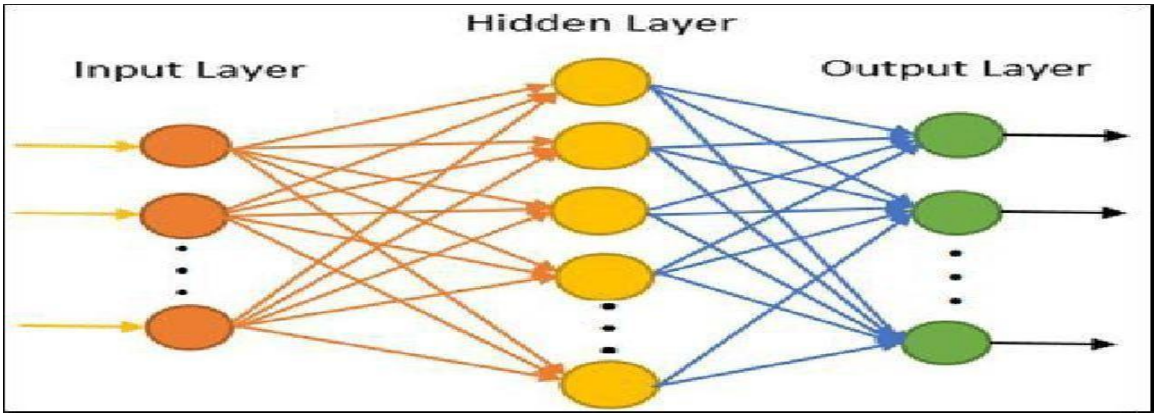
A single artificial neuron accepts multiple inputs, multiplies them by corresponding weights, adds a bias term, and passes the result through an activation function. This output is then passed to the next layer.

Mathematical model of a Neuron





2. ANN Architecture



Structure:

- **Input Layer**: Takes the input features.
- **Hidden Layers**: Apply transformations through weights, biases, and activation functions.
- **Output Layer**: Produces final predictions.

Components:

- **Weights**: Strength of connection between neurons.
- **Bias**: An offset added to shift the output.
- **Activation Function**: Introduces non-linearity.

Q&A Architecture:

Q. What's the role of weights in an ANN?

A. They determine how much influence an input feature has on the output. During training, weights are updated to reduce prediction errors.

Q. Why do we need a bias?

A. Bias lets the model fit data better by shifting activation outputs like a y-intercept in linear equations.

Hyperparameters include the number of layers, number of neurons in each layer, learning rate, batch size, and number of epochs. These control the capacity and speed of the network.

Why do we need Activation functions?

Activation functions are crucial in neural networks because they introduce **non-linearity** into the model, enabling the network to learn and model complex patterns in the data. Without activation functions, a neural network would essentially behave like a linear regression model, limiting its ability to capture the true underlying structure in most real-world problems.

- **Sigmoid**: Maps inputs to the range (0, 1). It's used in binary classification tasks but can suffer from vanishing gradient problems in deep networks.
- **Tanh (Hyperbolic Tangent)**: Maps inputs to the range (-1, 1), centering the output, which can sometimes lead to better convergence than sigmoid. However, it also suffers from vanishing gradient problems.
- **ReLU (Rectified Linear Unit)**: Introduces non-linearity by outputting the input directly if it's positive and zero otherwise. It's widely used in deep learning due to its simplicity and effectiveness. It also helps with the vanishing gradient problem.
- **Softmax**: Typically used in the output layer for multi-class classification problems. It converts logits (raw output values) into probabilities that sum to 1 across the classes

Used in layer	Activation Function	Details	Pros	Cons
Hidden / Output	**Sigmoid**	$\sigma(x) = 1/(1+e^{-x})$ Output range: [0,1]	- Smooth activation that outputs values between 0 and 1, making it suitable for binary classification - Historically popular	- Vanishing gradient problem due to saturated neurons - Output not zero-centered as sigmoid outputs are always positive - exp() operation is computationally intensive
Hidden	**Tanh (Hyperbolic tangent)**	$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$ Output range: [-1,1]	- Zero centered outputs that help networks train faster	- Suffers with vanishing gradient problem when saturated
Hidden	**ReLU** **(Rectified Linear Unit)**	$f(x) = max(0,x)$ Output range: [0, ∞)	- Does not saturate: avoids vanishing gradient issues for positive inputs - Computationally efficient since only certain number of neurons are activated at the same time - Much faster convergence compared to sigmoid/tanh	- Output not zero-centered - Prone to a "dying ReLU" problem, where neurons can get stuck during training and never activate again, leading to a dead neuron that doesn't update its weights. © AIML.com Research

Used in layer	Activation Function	Details	Pros	Cons
Hidden	**Leaky ReLU**	$f(x) = max(0.01x, x)$ Output range: (-∞, ∞)	- All benefits of ReLU - Addresses the "dying ReLU" problem by allowing a small gradient for negative inputs. Helps with training deeper networks	- Not as standardized as ReLU, and the slope of the leaky part is typically a hyperparameter (α) that needs tuning. This is also referred to as Parametric ReLU $f(x) = max(\alpha x, x)$
Hidden	**ELU (Exponential Linear Unit)**	where $\alpha > 0$: $f(x) = \begin{cases} x & for\ x \geqslant 0 \\ \alpha(e^{x} - 1) & for\ x < 0 \end{cases}$ α is commonly chosen as 1 Output range: (-α, ∞)	- All benefits of ReLU - Zero centered outputs that help networks train faster - ELUs saturate to a negative value when the argument gets smaller becoming more robust to noise	- Computationally more expensive due to exponential operation
Output	**Softmax**	For a given class *i*, probability $P(y_i)$ is: $P(y_i) = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}$ where, z_i is the raw score (logit) for class *i*, and *N* is the no. of classes Output range: (0,1)	- Used in the output layer of multi-class classification problems, converting model outputs into probability distributions.	- Not suitable for multi-label classification, as it enforces that only one class can be predicted. © AIML.com Research

3. Types of Neural Networks

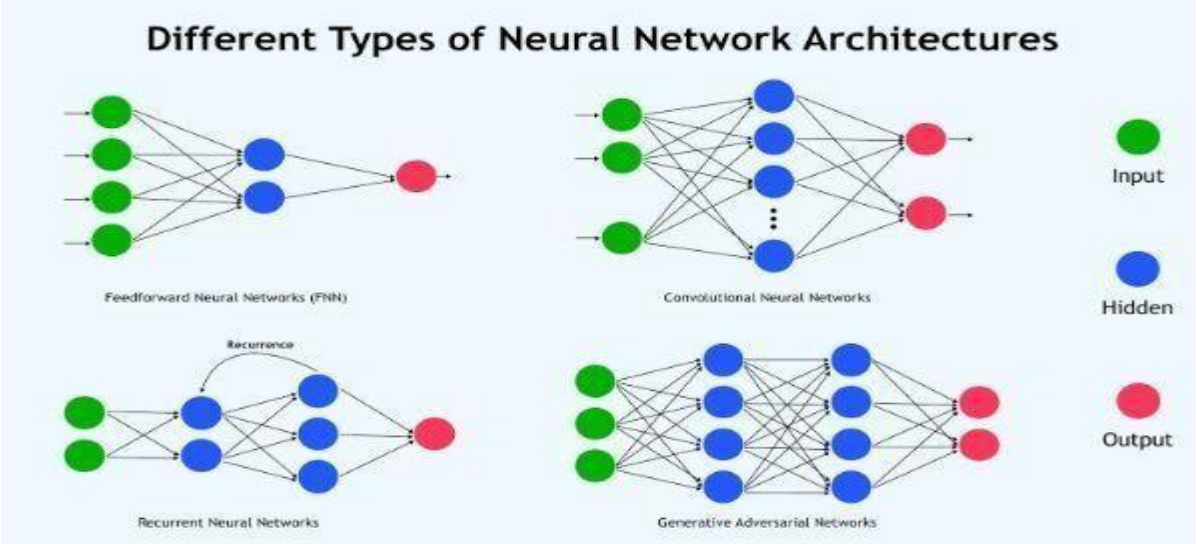
- Feedforward Neural Network (FNN): Data flows only in one direction.
- Multilayer Perceptron (MLP): A basic ANN with at least one hidden layer.
- Convolutional Neural Network (CNN): Ideal for image data.
- Recurrent Neural Network (RNN): Designed to handle sequential data like time-series or text.

Checkpoint Question

Scenario: You want to forecast future stock prices.

Which type of neural network would you use and why?

Answer: Recurrent Neural Network (RNN) because it can handle sequential data and learn from patterns over time.



4. Code Example Using Keras

The following is a basic ANN model in Keras for binary classification:

```
from tensorflow.keras.models import Sequential from
tensorflow.keras.layers import Dense
model = Sequential([
   Dense(16, input_dim=8, activation='relu'), Dense(8, activation='relu'),
   Dense(1, activation='sigmoid')
])
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy']) model.fit(X_train,
y_train, epochs=50, batch_size=10)
```

Checkpoint Question

Scenario: You get an error "input_dim not specified." Where should input_dim be set and why?

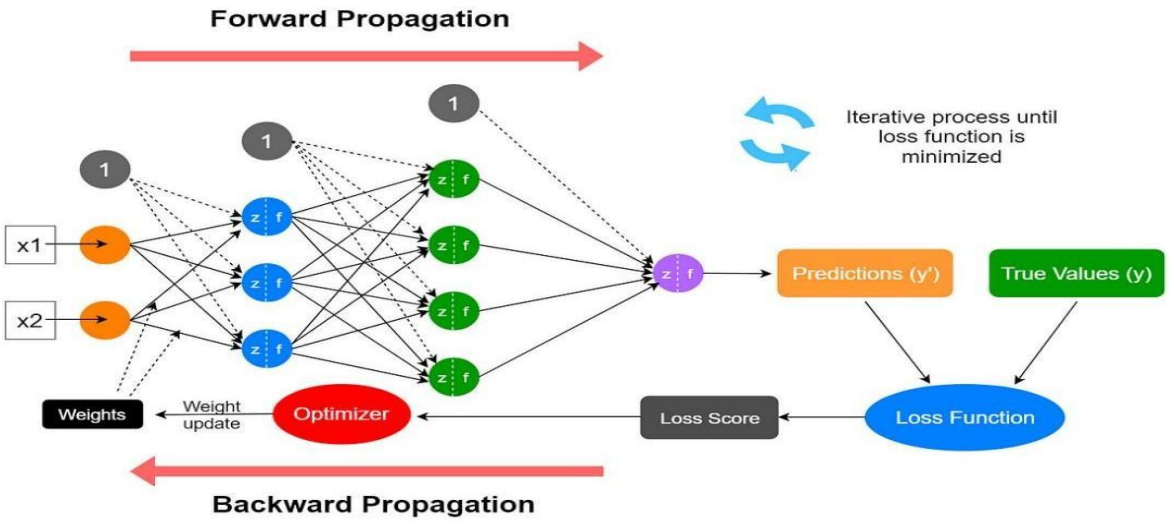
Answer: In the first Dense layer. It defines how many input features (columns) the model expects.

5. How ANN Learns

Training an ANN involves several steps:

- Forward propagation: Compute the output.
- Loss computation: Measure the error.
- Backpropagation: Adjust weights based on the error.
- Optimization: Use algorithms like Gradient Descent to minimize the loss.

The model learns better with good hyperparameter tuning and enough epochs to converge.



OPTIMIZERS IN DEEP LEARNING

Optimizers are algorithms or methods used to **update weights and biases** in a neural network to **minimize the loss function** during training.

They work using **gradient descent**, which involves:

1. Calculating the gradient of the loss function
2. Updating weights in the **opposite direction of the gradient** to minimize error

1. Gradient Descent

Gradient Descent, the bedrock of other optimization algorithms in deep learning, iteratively refines model parameters by computing the gradients of the entire dataset. It operates to minimize the loss function, adjusting weights based on the negative gradient multiplied by a fixed learning rate. While conceptually simple, its drawback lies in computational intensity, particularly for vast datasets,

often leading to slow convergence. The learning rate serves as a critical hyperparameter, influencing the size of weight updates at each iteration. Striking a balance with the learning rate is essential, as too small a value can prolong convergence, while a huge value may result in overshooting the optimal parameters.

2. Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) injects an element of randomness into the optimization process of the standard gradient descent algorithm by updating model parameters after processing individual training samples. This stochasticity brings computational advantages, significantly reducing per-iteration computation. The learning rate remains pivotal indetermining the step size for weight updates. While the randomness can lead to faster convergence, it introduces variability, potentially causing oscillations around the optimal solution. Despite this, SGD's efficiency is beneficial in scenarios with large datasets or non- convex optimization landscapes.

3. Mini-batch Gradient Descent

Mini-batch Gradient Descent strikes a balance between the deterministic nature of Gradient Descent and the stochasticity of SGD. It updates model parameters using small batches of data, offering a compromise that marries efficiency with accuracy. The batch size, a crucial parameter, influences the convergence speed and memory requirements. Mini-batch Gradient Descent inherits the benefits of both gradient descent algorithms, leveraging parallelization for accelerated computation while maintaining robustness against noise often present in stochastic approaches.

4. Adagrad

Adagrad (Adaptive gradient descent), an adaptive optimization algorithm, tailors the learning rate for each parameter based on their historical gradients. This adaptability is particularly beneficial in scenarios with sparse data, where certain features may require more refined updates. However, a potential downside arises from the cumulative sum of squared gradients in the denominator, causing the learning rates to diminish over time. Consequently, Adagrad may face challenges in scenarios where a diminishing learning rate adversely impacts convergence, especially in the later stages of training.

5. RMSprop

RMSprop, or Root Mean Square Propagation, shares similarities with Adagrad in adapting learning rates but mitigates its issue of diminishing rates. Instead of the cumulative sum of squared gradients, RMSprop utilizes a moving average of squared gradients. This modification allows for more stable and effective convergence, particularly in non-stationary environments. RMSprop's adaptive learning rate mechanism addresses challenges varying gradients pose, contributing to improved

optimization performance.

Learn more:

https://musstafa0804.medium.com/optimizers-in-deep-learning-7bf81fed78a0

https://medium.com/@tm.nidheesh95/optimizers-in-deep-learning-b12ef0cd4d8a

WHY ARE OPTIMIZERS IMPORTANT?

Because the **right optimizer improves convergence speed**, **stability**, and even **model accuracy**.

Different problems benefit from different optimizers.

Optimizer	Type	Learning Rate Adaptation	Uses Momentum	Suitable For	Pros	Cons
SGD	Basic	No	No	Simple models, large datasets	Easy to implement	Slow convergence
SGD + Momentum	Improved	No	Yes	Deep networks	Faster convergence	Needs extra hyperparameter
RMSprop	Adaptive	Yes	No	RNNs, time series	Adapts learning rates	Can stagnate
Adam	Adaptive + Momentum	Yes	Yes	Most DL models	Fast, efficient, widely used	Slightly complex, overfitting risk
Adagrad	Adaptive	Yes	No	Sparse data (NLP)	No manual tuning	Learning rate decays too fast
Adadelta	Adaptive	Yes	No	Online learning	Avoids decaying rate	Less used today

Imagine you are rolling down a hill (minimizing loss):

- SGD: Takes small, bumpy, zigzag steps
- Momentum: Gains speed while rolling downhill
- RMSprop: Adjusts how big/small the steps are based on terrain
- Adam: Combines the speed and smart stepping strategies together

Q. Why might SGD be too slow for deep networks?

A. Because it updates weights using small samples and without adaptive learning, it can take many steps to converge.

Q. When is Adam preferred?

A. In most practical cases, especially when you want fast and stable training without much tuning.

Q. Can one optimizer work for all problems?

A. Not necessarily. Try multiple optimizers and validate performance.

Checkpoint Question

Scenario: Loss decreases for 5 epochs then increases again. What could be happening and what can you do?

Answer: Overfitting. You can apply early stopping, dropout, or get more training data.

Q. What happens when loss is high?

A. The model is not learning correctly. High loss means poor predictions.

Q. Does low loss always mean good performance?

A. Not always …you might be **overfitting** (memorizing the training data).

Convergence & Epochs

- **Epoch** = 1 full pass through training data
- **Convergence** = point where loss no longer improves much

Q. How do you know the model is converging?

A. Loss decreases over epochs and flattens eventually.

Q. What if the model does not converge?

A. Check for: Too high/low learning rate Poor choice of optimizer Insufficient epochs Inadequate model complexity

6. Challenges and Solutions

Common issues in ANN training include:

- Overfitting: Model memorizes training data. Solution: Dropout, regularization.
- Vanishing Gradient: Small gradient updates. Solution: Use ReLU.
- Data Imbalance: Classes not equally represented. Solution: Use class weights or SMOTE.

Overfitting vs Underfitting

- Overfitting = Great on training data, poor on unseen data
- Underfitting = Poor on both training and validation

Causes of Overfitting:

- Model too complex
- Too many epochs
- Lack of regularization

Remedies:

- Early stopping
- Dropout layers
- Data augmentation
- L2 regularization

Q&A — Overfitting:

Q. How do you detect overfitting?

A. Training loss decreases, but validation loss increases.

Q. Is high accuracy always good?

A. Only if both training and validation accuracy are high.

Checkpoint Question

Scenario: Model is 98% accurate in training, but only 60% on test data. What's the issue and how do you fix it?

Answer: Overfitting. Try adding dropout layers, regularization, or early stopping.

7.ANN vs Traditional Algorithms

Feature	ANN	Traditional Algorithms
Handles Complexity	Yes	Limited
Feature Engineering	Learns automatically	Often manual
Interpretability	Lower	Higher
Speed	Slower on small data	Faster

Checkpoint Question

Scenario: Your dataset has only 500 samples. Would you use ANN or logistic regression? Why?

Answer: Logistic regression. ANN needs large datasets and may overfit with only 500 rows.

Checkpoint Question

Task: You trained a model for 20 epochs. After epoch 10, training loss kept dropping, but validation loss increased. What does it mean?

Answer: The model is overfitting. It's learning training data too well and failing to generalize.