DEEP LEARNING

HANDOUT: INTRODUCTION TO TENSORFLOW

1. Introduction to TensorFlow

TensorFlow is an open-source deep learning and machine learning framework developed by Google. It provides a comprehensive, flexible ecosystem of tools, libraries, and community resources for building and deploying machine learning applications across a wide range of platforms.

2.Role of TensorFlow in Building Deep Learning Models

- TensorFlow simplifies the model-building workflow across all stages:
- Data Ingestion: Load and preprocess data using tf.data pipelines.
- Model Definition: Use tf.keras.Sequential or the Functional API to define neural networks.
- Training: Compile models with optimizers and loss functions and fit them using .fit().
- Evaluation and Prediction: Use .evaluate() and .predict() on unseen data.
- Deployment: Export models with SavedModel or convert to mobile formats with TensorFlow Lite.

3.Key Features:

- Cross-Platform: Runs on CPUs, GPUs, TPUs, and mobile devices.
- Scalability: Suitable for training models on both local machines and distributed servers.
- Keras Integration: Offers a user-friendly API for rapid model building.
- Auto Differentiation: Simplifies backpropagation using tf.GradientTape.

4. What Are Tensors?

Tensors are multi-dimensional arrays used to represent data in deep learning models. They are the building blocks of all computations in TensorFlow — storing inputs, outputs, weights, gradients, etc. Tensors generalize matrices to higher dimensions. For example:

Scalar: 0-D tensor (e.g., 5)

Vector: 1-D tensor (e.g., [5, 6])

Matrix: 2-D tensor (e.g., [[1, 2], [3, 4]])

3D and beyond: for image, video, and sequence data

5.How Tensors Are Used to Build Neural Networks?

In TensorFlow, all model inputs, outputs, and intermediate data are represented as tensors. These tensors flow through the layers of a model, undergoing transformations through tensor operations like matrix multiplication, addition, and nonlinear activation functions.

For example, consider a simple dense neural network layer. The input tensor is multiplied by a weight tensor and then passed through an activation function to produce an output tensor:

```
import tensorflow as tf

x = tf.constant([[1.0, 2.0]]) # input tensor (1x2)

layer = tf.keras.layers.Dense(1) # single dense layer

output = layer(x) # output tensor (1x1)

print(output)
```

Code Explanation:

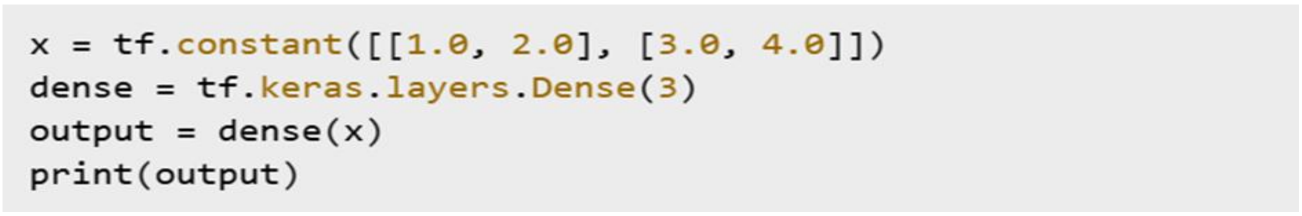
- x is a 2D tensor with shape [1, 2]
- The layer has a weight tensor of shape [2, 1]
- tf.matmul(x, weights) + bias is computed internally
- Result is passed through an activation (default: linear)

Tensors are not just containers for data; they enable efficient computation, automatic differentiation, and hardware acceleration on GPUs and TPUs.

Why does TensorFlow emphasize tensors instead of arrays?

Answer: Tensors support automatic differentiation, GPU execution, and dynamic shape manipulation.

Practice Task: Build a simple dense neural layer using tf.keras.layers. Dense and pass a 2D tensor input through it.



```
x = tf.constant([[1.0, 2.0], [3.0, 4.0]])
dense = tf.keras.layers.Dense(3)
output = dense(x)
print(output)
```

Code Explanation:

Line 1: Creating Input Tensor

x = tf.constant([[1.0, 2.0], [3.0, 4.0]])

- This creates a 2D tensor (or matrix) with shape (2, 2):

  [[1.0, 2.0],

  [3.0, 4.0]]

- Each row is treated as a separate input sample, and each column is a feature.
- So we are feeding 2 samples, each with 2 features, into the neural network.

Line 2: Defining a Dense Layer

dense = tf.keras.layers.Dense(3)

- Creates a Dense (fully connected) layer with:
  - 3 units (neurons) in the output
  - Input shape inferred from the input x (which has 2 features per sample)
  - Weights (W) shape: (2, 3)
  - Bias (b) shape: (3,)
- The output of this layer will be a tensor of shape (2, 3):
  - 2 rows for 2 input samples
  - 3 values per row for 3 neurons

Line 3: Forward Pass (Layer Execution)

output = dense(x)

This line performs the following operation for each input row:

output=xW+b

Where:

- x: input tensor of shape (2, 2)
- W: weights of shape (2, 3)
- b: bias of shape (3,)
- Result: tensor of shape (2, 3)

Example pseudo computation (not actual values):

output[0] = [1*W11 + 2*W21 + b1, 1*W12 + 2*W22 + b2, 1*W13 + 2*W23 + b3]

output[1] = [3*W11 + 4*W21 + b1, 3*W12 + 4*W22 + b2, 3*W13 + 4*W23 + b3]

Note: The weights and biases are initialized randomly unless manually set.

Line 4: Output

print(output)

This will print a 2x3 matrix (a tensor of shape (2, 3))

```
tf.Tensor(
[[ 0.123 -0.456 0.789]
 [ 0.321 -0.654 0.987]], shape=(2, 3), dtype=float32)
```

Each row corresponds to the dense layer output for one input sample.

Summary Table

Component	**Shape**	**Meaning**
Input x	(2, 2)	2 samples, 2 features each
Weights W	(2, 3)	Learnable parameters
Bias b	(3,)	Learnable offsets for each neuron
Output	(2, 3)	Dense layer result for 2 samples

6. Data Types in TensorFlow

TensorFlow supports multiple data types (dtypes) that affect memory use, precision, and speed. Choosing the right type ensures efficiency and accuracy.

Category	Examples	Description
Floating-point	tf.float16, tf.float32, tf.float64	Decimal numbers
Integer	tf.int8, tf.int16, tf.int32	Whole numbers
Boolean	tf.bool	True/False values
String	tf.string	Text data
Complex	tf.complex64, tf.complex128	Complex numbers

Example

```
import tensorflow as tf

a = tf.constant(3.14, dtype=tf.float32)

b = tf.constant(42, dtype=tf.int64)

c = tf.constant("Hello", dtype=tf.string)
```

Your model needs to process both images (0-255 integers) and text data. Which dtypes would you use?

Answer: Images: tf.uint8, Text: tf.string

You encounter a type mismatch error between operations. What will you check?

Answer: Ensure all tensors in the operation share compatible dtypes.

Problem: Create a tensor of shape [4, 4] with float values, then cast it to int32.

Solution:

```
import tensorflow as tf

float_tensor = tf.random.uniform([4, 4], dtype=tf.float32)

int_tensor = tf.cast(float_tensor, tf.int32)

print("Original tensor:", float_tensor)

print("Casted tensor:", int_tensor)
```

7. Numerical Precision

Precision controls how numbers are stored:

- float16 is faster but may lose accuracy (underflows)
- float32 is standard for neural networks
- float64 is slower but preserves precision

Too low precision can lead to numerical instability.

Example

```
tiny_float16 = tf.constant(1e-7, dtype=tf.float16)

tiny_float32 = tf.constant(1e-7, dtype=tf.float32)
```

Your model outputs NaN during training. What are two possible fixes?
Answer: Use tf.clip_by_norm() or switch to float32.

You observe inconsistent outputs across devices (CPU vs GPU). Could precision be a factor?

Answer: Yes. Different hardware may use different default precisions.

Practice Problem

Try computing the dot product of two large vectors using float16 and float64. Observe differences.

Solution:

```
import tensorflow as tf

v1 = tf.random.uniform([1000000], dtype=tf.float16)

v2 = tf.random.uniform([1000000], dtype=tf.float16)

dot_16 = tf.tensordot(v1, v2, axes=1)

v1_64 = tf.cast(v1, tf.float64)

v2_64 = tf.cast(v2, tf.float64)

dot_64 = tf.tensordot(v1_64, v2_64, axes=1)

print("Dot product with float16:", dot_16.numpy())

print("Dot product with float64:", dot_64.numpy())
```

8. Tensors vs Variables

Tensors in TensorFlow are immutable. To store weights or other learnable parameters that change during training, use tf.Variable, which allows updates.

Feature	tf.Tensor	tf.Variable
Mutability	Immutable	Mutable
Use Case	Input data	Trainable model weights

Example

```
weights = tf.Variable(tf.random.normal([3, 2]))

weights.assign_add(tf.ones([3, 2]))
```

Why can't we use regular tensors for neural network weights?

Answer: Because weights need to be updated during training.

How would you freeze a pretrained layer in TensorFlow?

Answer: Set trainable=False on its variables.

Practice Problem

Create a trainable variable initialized with zeros of shape [5,5] and increment its values by 1.

Solution:

```
import tensorflow as tf

var = tf.Variable(tf.zeros([5, 5]))

var.assign_add(tf.ones([5, 5]))

print(var)
```

9. Creating Tensors

TensorFlow provides a variety of methods to create tensors for different purposes from constant values to random initialization. These are fundamental when building models.

Method	**Example**	**Use Case**
tf.constant()	tf.constant([[1, 2], [3, 4]])	Fixed data
tf.zeros()	tf.zeros([2, 3])	Initialization

tf.ones()	tf.ones([2, 2])	Fill with ones
tf.eye()	tf.eye(3)	Identity matrix
tf.random.normal()	tf.random.normal([2,2], mean=0, stddev=1)	Random weights

How would you initialize a weight matrix with small random values for a neural layer?

Answer: Use tf.random.normal() with small stddev.

You want to create a batch of 5 grayscale images (28x28 pixels) initialized to 0. What method would you use?

Answer: tf.zeros([5, 28, 28])

Practice Problem

Create a 3x3 tensor with values ranging from 0 to 8.

Solution:

```
import tensorflow as tf

t = tf.reshape(tf.range(9), [3, 3])

print(t)
```

10. Indexing and Slicing

TensorFlow supports powerful slicing and indexing operations similar to NumPy. These help extract or manipulate subsets of data.

Type	**Example**	**Description**
Basic	tensor[0]	First element or row
Slicing	tensor[1:3, :2]	Slice rows and columns
Boolean	tensor[tensor > 0]	Masked selection

Example

```
t = tf.constant([[1, 2, 3], [4, 5, 6]])

print(t[:, 1:])  # [[2, 3], [5, 6]]
```

How do you access the second column of a matrix?

Answer: tensor[:, 1]

Extract every even-indexed row from a batch of 10 images.

Answer: tensor[::2]

Practice Problem

Given a 4x4 tensor, extract the diagonal elements.

Solution:

```
t = tf.reshape(tf.range(16), [4, 4])

diagonal = tf.linalg.diag_part(t)

print(diagonal)
```

11. Dimensional Operations

These operations allow changing or manipulating tensor shapes to match the requirements of models or functions.

Operation	**Example**	**Effect**
reshape()	tf.reshape(t, [6,10])	Changes shape
expand_dims()	tf.expand_dims(t, axis=0)	Adds new dimension
squeeze()	tf.squeeze(t)	Removes dim=1

How to convert a batch of images from [28,28,1] to [28,28]?

Answer: tf.squeeze(t, axis=-1)

Practice Problem

Create a tensor of shape [3, 1, 4] and convert it to [3, 4].

Solution:

```
t = tf.random.normal([3, 1, 4])
```

```
t_flat = tf.squeeze(t)

print(t_flat.shape)
```

12. Broadcasting

Broadcasting lets TensorFlow automatically expand tensors of smaller shapes to match larger ones when performing element-wise operations.

Rules

- Align shapes from the right
- If dimensions differ and one of them is 1, broadcasting occurs

Example

```
a = tf.constant([1, 2, 3])      # (3,)

b = tf.constant([[10], [20]])  # (2,1)
```

```
print(a + b)                # [[11,12,13],[21,22,23]]
```

What's the shape of [3,1] * [1,4]?

Answer: [3,4]

You get a shape mismatch error in element-wise multiplication. What's likely the issue?

Answer: Incompatible shapes without broadcasting rules being satisfied.

Practice Problem: Multiply a tensor of shape [2,3] with [3] using broadcasting.

Solution:

```
a = tf.constant([[1, 2, 3], [4, 5, 6]])

b = tf.constant([1, 2, 3])

print(a * b)
```

13. Math Operations

TensorFlow supports a wide range of mathematical operations, including element-wise arithmetic, matrix multiplication, and reductions like mean or sum.

Type	Examples	Description

Arithmetic	tf.add(), tf.subtract()	Element-wise operations
Matrix	tf.matmul(), tf.linalg.inv()	Matrix algebra
Reduction	tf.reduce_sum(), tf.reduce_mean()	Aggregation functions

Example

```
a = tf.constant([[1, 2], [3, 4]])

print(tf.reduce_sum(a, axis=1))  # Output: [3, 7]
```

How do you compute the mean of all elements in a tensor?

Answer: tf.reduce_mean(tensor)

What is the difference between tf.multiply() and tf.matmul()?

Answer: tf.multiply() is element-wise; tf.matmul() is matrix multiplication.

Practice Problem: Compute the dot product of two vectors [1, 2, 3] and [4, 5, 6].

Solution:

```
v1 = tf.constant([1, 2, 3])
v2 = tf.constant([4, 5, 6])
dot_product = tf.tensordot(v1, v2, axes=1)
print(dot_product)  # Output: 32
```

14. Merge and Split

TensorFlow allows joining and separating tensors for reshaping or batching operations.

Operation	Example	Use Case
concat()	tf.concat([a, b], axis=0)	Join along an axis
stack()	tf.stack([a, b], axis=0)	Create new dimension
split()	tf.split(tensor, num=3, axis=1)	Split into parts

How would you combine two tensors of shape [2,3] vertically?

Answer: Use tf.concat([a, b], axis=0)

You need to split a 6x2 tensor into three 2x2 matrices. Which function do you use?

Answer: tf.split(tensor, num_or_size_splits=3, axis=0)

Practice Problem: Stack two 1D tensors [1, 2, 3] and [4, 5, 6] to make a 2x3 tensor.

Solution:

```
a = tf.constant([1, 2, 3])

b = tf.constant([4, 5, 6])

st = tf.stack([a, b], axis=0)

print(st)
```

15. Statistics

Statistical operations help summarize or rank values inside tensors.

Function	Example	Description
reduce_mean()	tf.reduce_mean(t)	Mean of all elements
argmax()	tf.argmax(t, axis=1)	Index of max values
top_k()	tf.math.top_k(t, k=3)	Top K values and indices

How do you find the most likely class from softmax output?

Answer: Use tf.argmax(predictions, axis=1)

You want to find the 3 highest predictions from a tensor. What function do you use?

Answer: tf.math.top_k()

Practice Problem: Given t = [3.2, 1.5, 7.8, 5.5], return the top 2 values.

Solution:

```
t = tf.constant([3.2, 1.5, 7.8, 5.5])

values, indices = tf.math.top_k(t, k=2)

print("Top values:", values.numpy())
```

16. Tensor Comparison

TensorFlow supports comparison operations such as equality, greater-than, and conditional selection.

Operation	**Example**	**Description**
tf.equal()	tf.equal(a, b)	Element-wise equality
tf.greater()	tf.greater(a, b)	Element-wise greater check
tf.where()	tf.where(condition, x, y)	Element from x or y based on condition

How do you replace all negative values in a tensor with 0?

Answer: tf.where(tensor < 0, 0, tensor)

How would you find where two tensors have equal values?

Answer: Use tf.equal(tensor1, tensor2)

Practice Problem: Given a tensor t = [-1, 3, -5, 2], replace negative values with 0.

Solution:

```
t = tf.constant([-1, 3, -5, 2])

result = tf.where(t < 0, tf.zeros_like(t), t)

print(result)
```

17. Fill and Copy

Useful for creating fixed-value tensors or duplicating tensor contents.

Function	Example	Description
tf.fill()	tf.fill([2, 2], 5)	Tensor filled with value
tf.identity()	tf.identity(t)	Clone of input tensor

How do you create a 10x10 tensor filled with value 7?

Answer: tf.fill([10, 10], 7)

What's the purpose of tf.identity() in computation graphs?

Answer: It allows keeping computation paths separate and preserved

Practice Problem: Create a 3x3 tensor filled with the value -2.

Solution:

```
t = tf.fill([3, 3], -2)

print(t)
```

18. Data Limiting

TensorFlow provides utilities to clip tensor values or norms to maintain numerical stability, especially in deep learning.

Method	Use Case
clip_by_value()	Restrict range of values
clip_by_norm()	Prevent exploding gradients

How would you limit pixel intensities to [0, 255]?

Answer: tf.clip_by_value(image, 0, 255)

Practice Problem: Clip values in t = [-10, 0, 5, 20] between 0 and 10.

Solution:

```
t = tf.constant([-10, 0, 5, 20], dtype=tf.float32)

clipped = tf.clip_by_value(t, 0, 10)

print(clipped)
```

19. Advanced Operations

TensorFlow includes advanced tensor types and tools for custom training workflows.

Feature	Use Case
GradientTape	Manual gradient computation
SparseTensor	Efficient for sparse data storage
RaggedTensor	Handling sequences of varying lengths

You want to compute gradient of y = x^2 manually. What do you use?

Answer: tf.GradientTape()

Practice Problem: Use GradientTape to compute the derivative of y = x^3 at x = 2.

Solution:

```
x = tf.Variable(2.0)

with tf.GradientTape() as tape:

    y = x ** 3

grad = tape.gradient(y, x)

print(grad)

# Output: 12.0
```

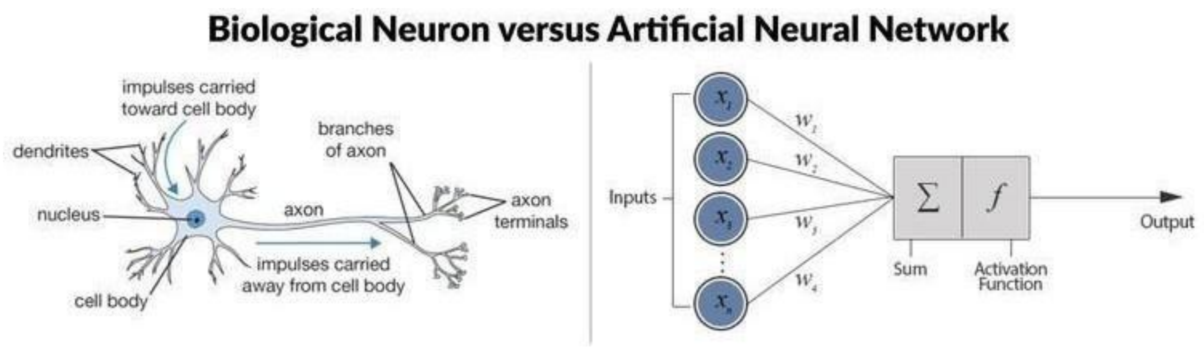
DEEP LEARNING

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.

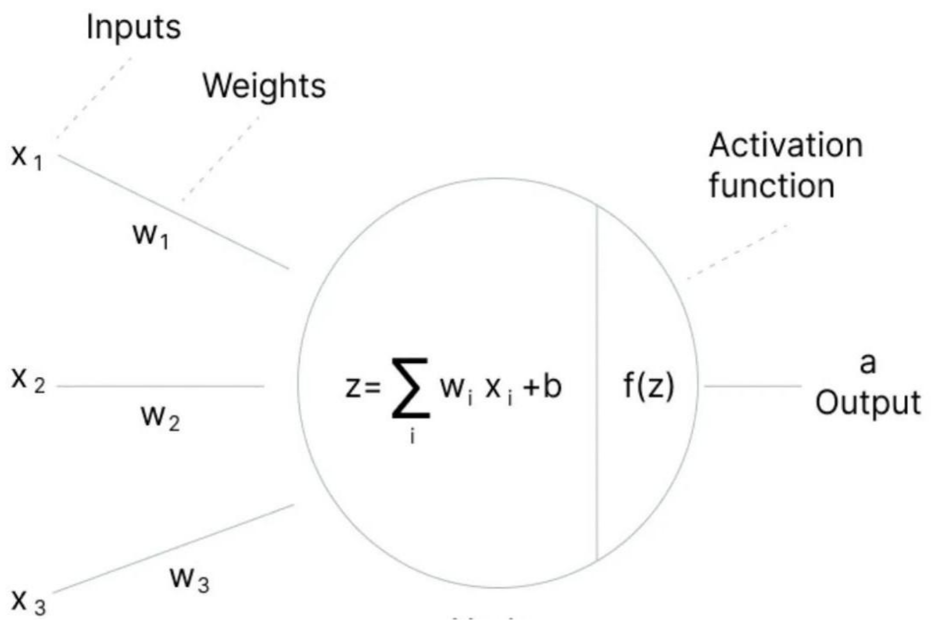
Why might an ANN be more suitable than simple if-else rules for this task?

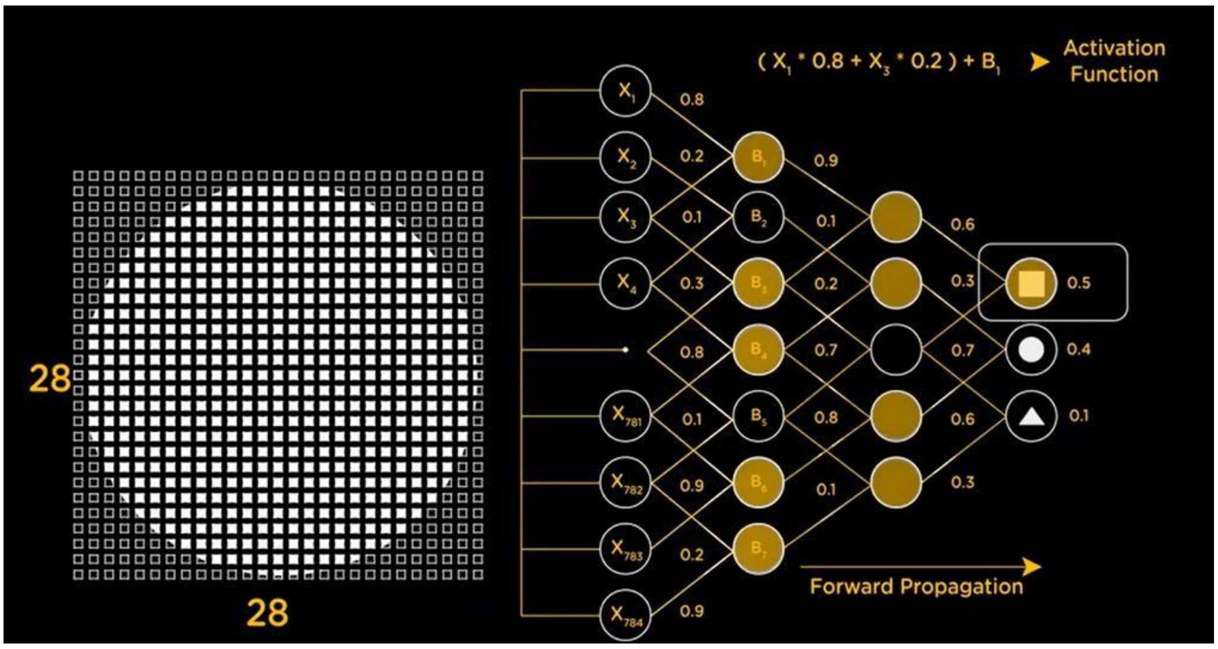
Answer: Because ANN can learn complex, non-linear relationships between temperature, humidity, and pressure that are hard to capture using fixed rules. It adapts and generalizes better than static rule-based systems.

2. Understanding the Basics

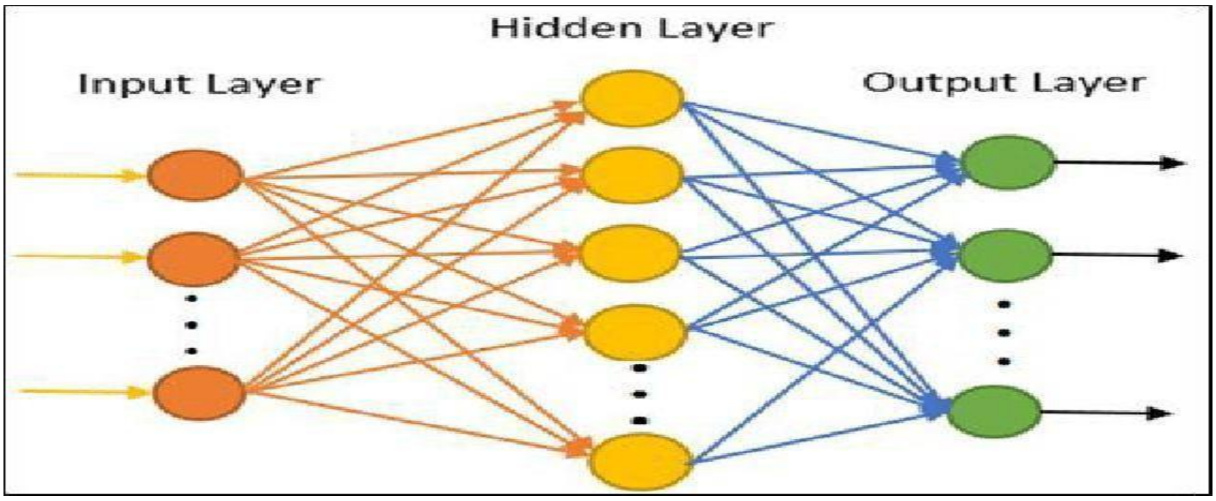
A single artificial neuron accepts multiple inputs, multiplies them by corresponding weights, adds a bias term, and passes the result through an activation function. This output is then passed to the next layer.

Mathematical model of a Neuron





3. 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:

Q1. What's the role of weights in an ANN?

A1. They determine how much influence an input feature has on the output. During training, weights are updated to reduce prediction errors.

Q2. Why do we need a bias?

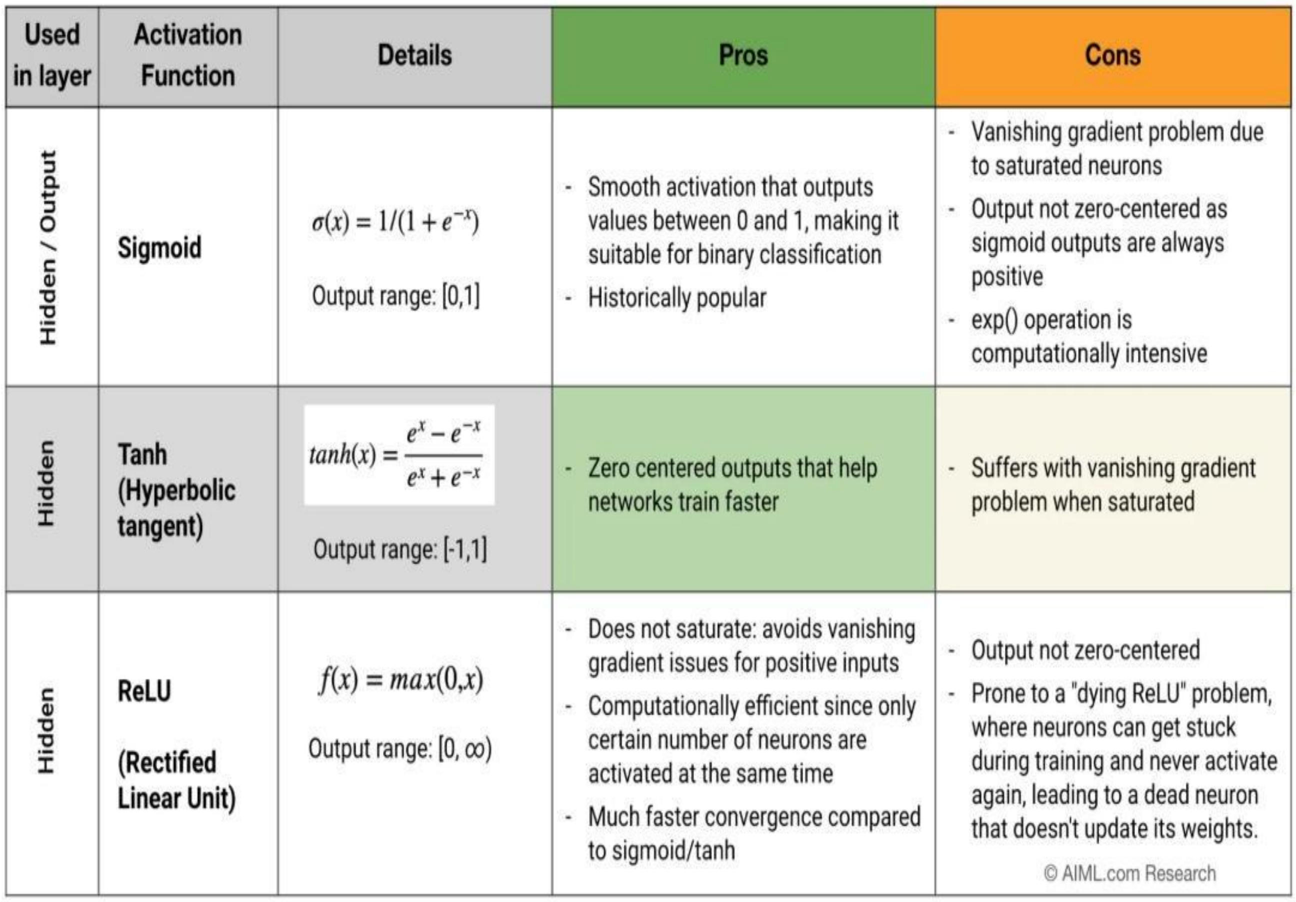
A2. Bias lets the model fit data better by shifting activation outputs — like a y-intercept in linear equations.

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?

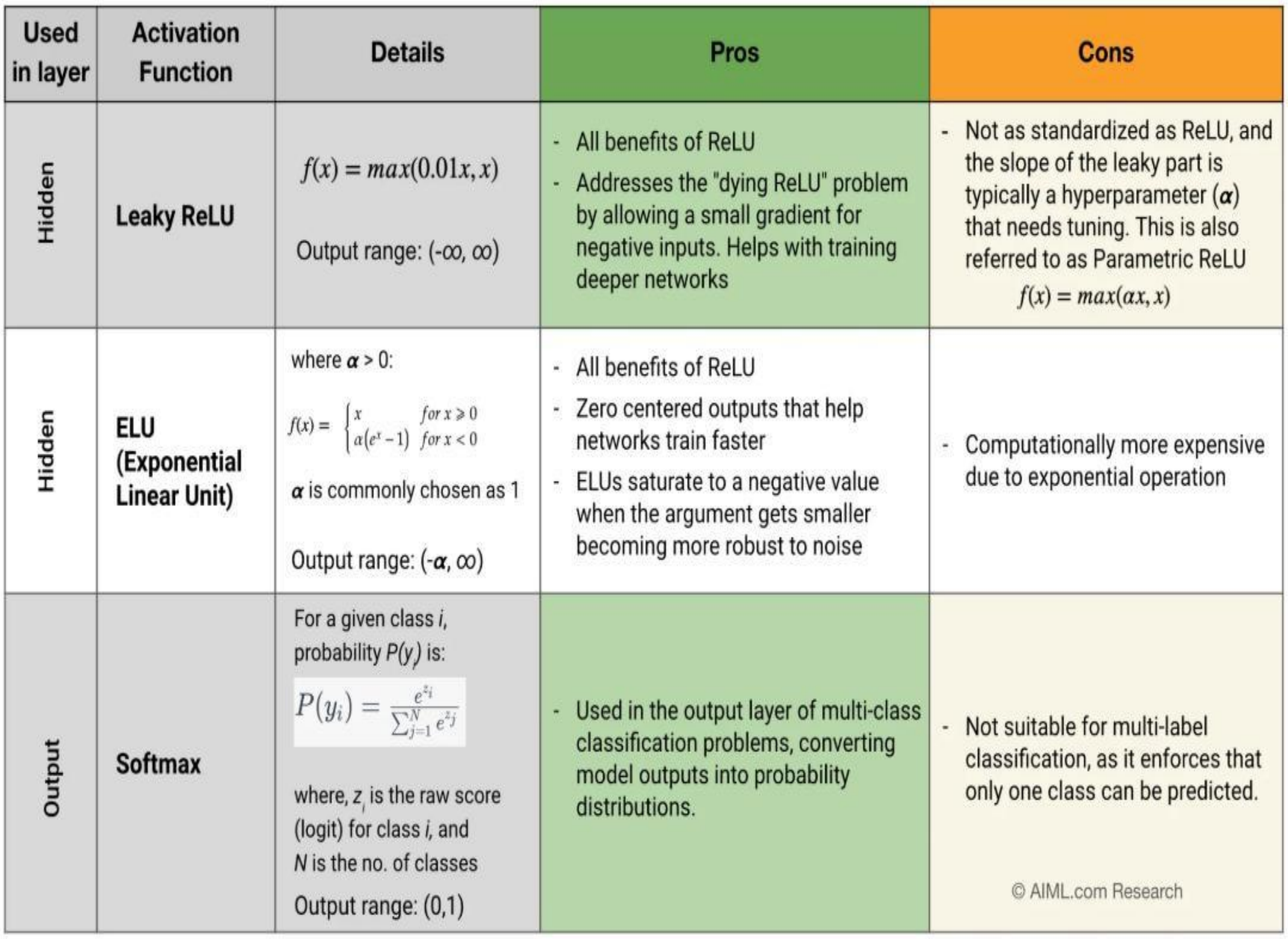
Activation functions are crucial in neural networks because they introduce **non-linearity** into the model, enabling the network to learn and model complex patterns in the data. Without activation functions, a neural network would essentially behave like a linear regression model, limiting its ability 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: $(-\infty, \infty)$	- 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: $(-\alpha, \infty)$	- 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

4. Types of Neural Networks

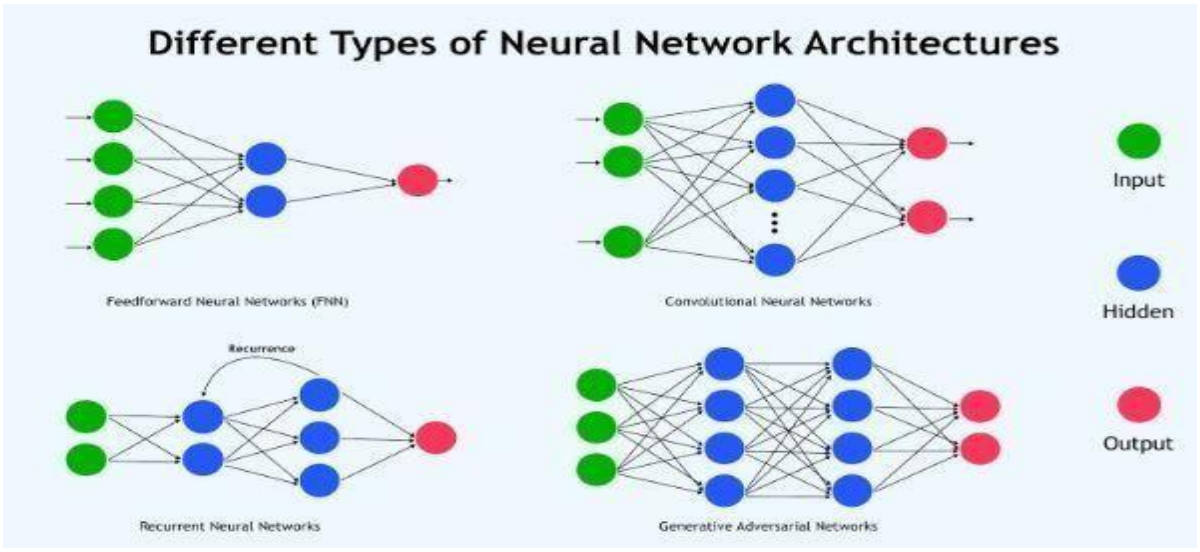
- Feedforward Neural Network (FNN): Data flows only in one direction.
- Multilayer Perceptron (MLP): A basic ANN with at least one hidden layer.
- Convolutional Neural Network (CNN): Ideal for image data.
- Recurrent Neural Network (RNN): Designed to handle sequential data like time-series or text.

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.


Different Types of Neural Network Architectures
Feedforward Neural Networks (FNN)
Convolutional Neural Networks
Recurrence
Recurrent Neural Networks
Generative Adversarial Networks
Input
Hidden
Output

5. 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?

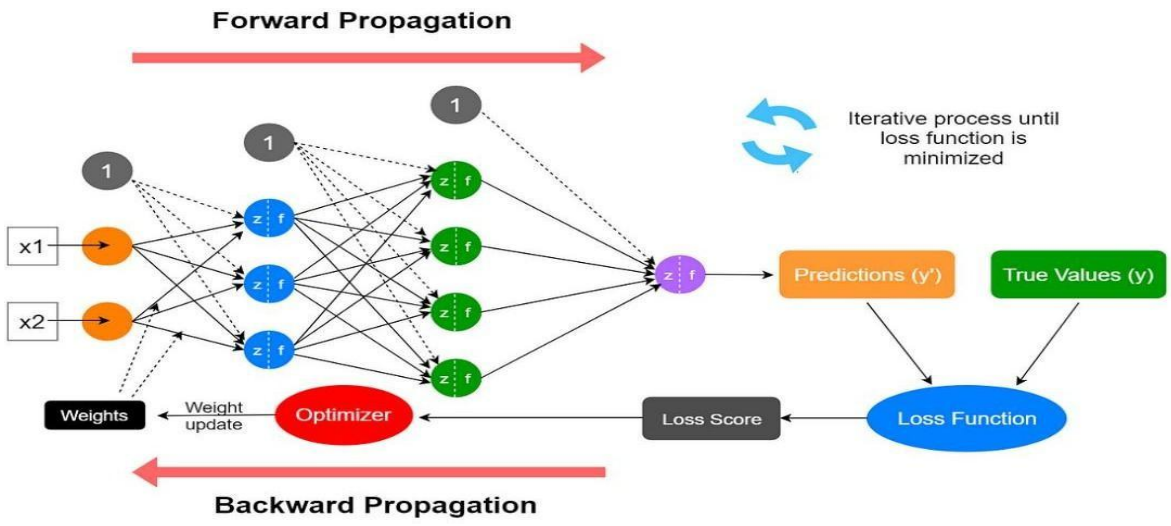
Answer: In the first Dense layer. It defines how many input features (columns) the model expects.

6. 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 in determining 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

Q3. Why might SGD be too slow for deep networks?

A3. Because it updates weights using small samples and without adaptive learning, it can take many steps to converge.

Q4. When is Adam preferred?

A4. In most practical cases, especially when you want fast and stable training without much tuning.

Q5. Can one optimizer work for all problems?

A5. 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.

Q6. What happens when loss is high?

A6. The model is not learning correctly. High loss means poor predictions.

Q7. Does low loss always mean good performance?

A7. 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

Q8. How do you know the model is converging?

A8. Loss decreases over epochs and flattens eventually.

Q9. What if the model does not converge?

A9. Check for:

Too high/low learning rate

Poor choice of optimizer

Insufficient epochs

Inadequate model complexity

7. 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:

Q10. How do you detect overfitting?

A10. Training loss decreases, but validation loss increases.

Q11. Is high accuracy always good?

A8. 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.

8. 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.