Stage 1: Foundations of Deep Learning

Goal: Understand the basics and build your first neural network.

Topics:

- What is Deep Learning?
- Neural Networks: Neurons, Layers, Activation Functions
- Loss Functions & Optimizers
- Forward & Backward Propagation
- Hands-on: Build a neural network from scratch using NumPy

Stage 2: Practical Deep Learning with PyTorch

Goal: Learn to use PyTorch (industry standard) to build and train models.

Topics:

- Tensors and Autograd
- Building models with nn.Module
- Training loops and optimizers
- Hands-on: Image classification with MNIST dataset

Stage 3: Advanced Architectures

Goal: Learn powerful models used in real-world applications.

Topics:

- Convolutional Neural Networks (CNNs)
- Recurrent Neural Networks (RNNs), LSTMs
- Transformers (basic intro)
- Hands-on: Build a CNN for image classification

Stage 4: Real-World Projects

Goal: Apply your skills to real datasets and problems.

Projects:

- Image classification (e.g., cats vs dogs)
- Sentiment analysis (NLP)
- Object detection or segmentation
- Deploying a model with Flask or Streamlit

Stage 5: Industry Readiness

Goal: Prepare for interviews and real-world engineering roles.

Topics:

- Model evaluation and tuning
- Working with large datasets
- Reading research papers
- Git, Docker, and cloud basics (AWS/GCP)
- Resume & portfolio building

The Foundation of Deep Learning

🤖 What is Deep Learning?

Deep Learning is a type of **Machine Learning** that uses **neural networks** with many layers to learn from data — just like how our brain works with neurons!

🧩 Key Concepts

Concept	Meaning
Neuron	A single unit that takes input, does math, and gives output
Layer	A group of neurons working together
Activation Function	Adds non-linearity (like ReLU or Sigmoid)
Loss Function	Measures how wrong the model is
Optimizer	Helps the model learn better (like SGD or Adam)

🤖 **Deep Learning vs. Statistical Machine Learning:**

Feature	Statistical ML	Deep Learning
Data Requirement	Works well with **small to medium** datasets	Needs **large** datasets to perform well
Feature Engineering	Requires **manual feature extraction**	Learns features **automatically**
Interpretability	More **interpretable** (e.g., decision trees, linear models)	Often a **black box**
Training Time	Usually **faster** to train	Can be **computationally expensive**
Examples	Logistic Regression, SVM, Random Forest	CNNs, RNNs, Transformers
Best For	Tabular data, small datasets, explainability	Images, audio, text, large-scale data

🧠 **When to Use What?**

✅ **Use Statistical ML when:**

- You have **limited data**
- You need **explainability** (e.g., in healthcare or finance)
- You're working with **structured/tabular data**
- You want **faster training and deployment**

✅ **Use Deep Learning when:**

- You have **lots of data** (e.g., millions of images or text samples)
- You're working with **unstructured data** (images, audio, video, text)
- You want **state-of-the-art performance**
- You can afford **high compute resources**

💉 **Real-World Examples**

Problem	Best Approach
Predicting loan default from customer data	Statistical ML (e.g., XGBoost)
Classifying cat's vs dogs in images	Deep Learning (CNNs)
Sentiment analysis on tweets	Deep Learning (LSTM/Transformer)
Predicting house prices	Statistical ML (Linear Regression, Random Forest)

Neural Network Architectures:

Neural network architectures are like different blueprints for solving different types of problems.

🧠 **Core Neural Network Architectures**

1. Feedforward Neural Network (FNN)

- **Structure**: Input → Hidden Layers → Output
- **Use Case**: Basic classification/regression
- **Example**: Predicting house prices

2. Convolutional Neural Network (CNN)

- **Structure**: Convolution Layers + Pooling + Fully Connected
- **Use Case**: Image and video processing
- **Example**: Face recognition, object detection

3. Recurrent Neural Network (RNN)

- **Structure**: Loops over time steps
- **Use Case**: Sequence data (text, time series)
- **Example**: Language modelling, stock prediction

4. Long Short-Term Memory (LSTM) / GRU

- **Improved RNNs**: Handle long-term dependencies
- **Use Case**: Text generation, speech recognition

5. Autoencoders

- **Structure**: Encoder → Bottleneck → Decoder
- **Use Case**: Dimensionality reduction, anomaly detection

- **Example**: Denoising images

6. Generative Adversarial Networks (GANs)

- **Structure**: Generator vs Discriminator (two networks)
- **Use Case**: Image generation, deepfakes
- **Example**: Creating realistic human faces

7. Transformers

- **Structure**: Self-attention + Feedforward layers
- **Use Case**: NLP, vision, multi-modal tasks
- **Example**: ChatGPT, BERT, Vision Transformers

⚔️ **PyTorch vs TensorFlow: A Side-by-Side Comparison**

Feature	PyTorch	TensorFlow
Ease of Use	🟢 More Pythonic and intuitive	🟡 Steeper learning curve
Debugging	🟢 Dynamic computation graph (eager execution) makes debugging easy	🟡 Static graph (TF 1.x), improved in TF 2.x
Community & Adoption	🟢 Widely used in research and academia	🟢 Widely used in production and industry
Model Deployment	🟡 TorchServe, ONNX (less mature)	🟢 TensorFlow Serving, TensorFlow Lite, TensorFlow.js
Visualization	🟡 Basic with Matplotlib	🟢 TensorBoard (very powerful)
Mobile & Edge	🟡 PyTorch Mobile (growing)	🟢 TensorFlow Lite (more mature)
Integration	🟢 Easy with Python tools	🟢 Strong support for multiple languages (Python, JavaScript, C++)
Performance	🟢 Excellent with TorchScript	🟢 Excellent with XLA and TF-Serving

🧠 **When to Use What?**

✅ **Choose PyTorch if:**

- You're learning or doing research
- You want fast prototyping and debugging
- You prefer clean, readable Python code

✅ **Choose TensorFlow if:**

- You're deploying models to production (especially on mobile or web)
- You need advanced tools like TensorBoard or TFX
- You're working in a Google Cloud ecosystem

💉 **Real-World Usage**

Company	Framework	Use Case
Meta (Facebook)	PyTorch	Research, computer vision
Google	TensorFlow	Search, Translate, mobile AI
Tesla	PyTorch	Self-driving car vision
Airbnb	TensorFlow	Price prediction, personalization
Hugging Face	PyTorch	Transformers, NLP models

Fundamentals of Neural Networks

🧠 **What is a Neuron in Deep Learning?**

A **neuron** is the **basic building block** of a neural network — just like a cell in your brain!

🔧 **What does it do?**

A neuron:

1. **Takes input values** (like numbers)
2. **Applies some math** (weighted sum + bias)
3. **Passes it through an activation function**
4. **Outputs a value** (used by the next layer)

🧱 **Simple Formula of a Neuron:**

$$\boldsymbol{Output = Activation(w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b)}$$

$$\mathbf{Input(s)}(x_i)\ \mathbf{weights(w_i)} \rightarrow \left(y = \sum_{i=0}^{n} w_i x_i + b \,\middle|\, z = \frac{1}{1+e^{-y}}\right) \rightarrow output$$

Neuron

Where:

- $x_1, x_2, \ldots$ = inputs
- $w_1, w_2, \ldots$ = weights
- b = bias
- **Activation** = function like ReLU or Sigmoid
- **Z** is between 0 and 1

🧠 **Real-Life Analogy:**

Imagine a **light switch**:

- Inputs = how hard you press
- Weights = how sensitive the switch is
- Bias = how easy it is to turn on
- Activation = whether the light turns on or not

What is a Perceptron?

A **Perceptron** is the **simplest type of neural network** — it has:

- **One layer**
- **One or more inputs**
- **One output**
- A **step function** as the activation

🔧 **Formula:**

$$Output = Activation(w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b)$$

$$input(s) weights(s) \rightarrow sum\ of\ inputs\ and\ its\ weights \rightarrow step\ function\ (0\ or\ 1) \rightarrow output$$

✅ **Use Case:**

- Works only for **linearly separable** problems (like AND, OR)
- Cannot solve XOR

🧠 **What is a Multilayer Perceptron (MLP)?**

An **MLP** is a **stack of perceptron's** — it has:

- **Input layer**
- **One or more hidden layers**

- **Output layer**
- Uses **non-linear activation functions** (like ReLU, Sigmoid)

🔧 **Why is it powerful?**

- Can solve **non-linear problems** (like XOR)
- Learns **complex patterns** in data

🔍 **Key Differences**

Feature	Perceptron	MLP
Layers	Single layer	Multiple layers
Activation	Step function	Non-linear (ReLU, Sigmoid, etc.)
Problem Type	Linear	Non-linear
Learning	Simple rule	Backpropagation
Power	Limited	Very powerful

$age(x_1)w_{11} + education(x_2)w_{12} + income(x_3)w_{13} + savings(x_4)w_{14}$ ➔ input layer

$Awareness(h_1) + affordability(h_2)$ ➔ Hidden layer

$h_1 = \sigma(w_{11} * x_1 + w_{12} * x_2 + w_{13} * x_3 + w_{14} * x_4)$

$h_2 = \sigma(w_{21} * x_1 + w_{22} * x_2 + w_{23} * x_3 + w_{24} * x_4)$

$0 = \sigma(w_{01} * h_1 + w_{02} * h_2)$ ➔ will buy insurance ➔ output 0>0.5(buy) | o<=0.5 (no)

w_{01}, w_{02} are the weights of the hidden layer passed to the output layer

🧠 **Purpose of an Activation Function**

🔦 **1. Adds Non-Linearity**

Without activation functions, a neural network would just be a linear function — no matter how many layers you add. That means it could only learn straight-line relationships.

Activation functions allow the network to learn complex patterns — like curves, edges, and language!

⚙️ **2. Controls the Output of a Neuron**

It decides whether a neuron should be activated or not — based on the input it receives.

🔁 3. Helps with Learning

Some activation functions (like ReLU, Sigmoid, Tanh) help the network learn faster and better by controlling gradients during backpropagation.

🪀 Common Activation Functions

Function	Output Range	Use Case
ReLU (Rectified Linear Unit)	**0 to ∞**	**Most common in hidden layers**
Sigmoid	**0 to 1**	**Binary classification, output layers**
Tanh	**-1 to 1**	**Centred version of sigmoid**
Softmax	**0 to 1 (sums to 1)**	**Multi-class classification**

🧠 Analogy:

Think of an activation function like a decision gate:

- If the input is strong enough → let it pass
- If not → block or reduce it

1. Sigmoid

- Output: between **0 and 1**
- Good for: **binary classification**
- Problem: can cause **vanishing gradients**
- Example: Will buy insurance? Transaction Fraud

$$x = \frac{1}{1 + e^{-z}}$$

2. ReLU (Rectified Linear Unit)

- Output: **0** for negative inputs, **linear** for positive
- Good for: **hidden layers** in deep networks
- Fast and efficient, but can suffer from **"dead neurons"**
- Default choice for neurons in hidden layers
- Less prone to **Vanishing gradients**

$$ReLU(z) = \max(0, x)$$

3. Tanh

- Output: between **-1 and 1**
- Good for: **centered data**
- Still suffers from **vanishing gradients**, but better than sigmoid

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

4. Softmax

- Output: **probabilities** that sum to 1
- Good for: **multi-class classification**
- Used in the **output layer** of classification models
- Example: Hand written Digits Recognition, Clothe detection

PyTorch

🔥 **What is PyTorch?**

PyTorch is an open-source deep learning library developed by **Meta (Facebook)**. It's known for:

- **Easy-to-read code**
- **Dynamic computation graphs** (great for debugging)
- Widely used in **research and production**

🧱 **PyTorch Basics: Building Blocks**

Here's what you'll learn first:

1. **Tensors** – the core data structure (like NumPy arrays but with GPU support)
2. **Autograd** – automatic differentiation for backpropagation
3. **nn.Module** – for building neural networks
4. **Optimizers** – like SGD, Adam for training
5. **Training Loop** – how to train a model step-by-step

🧮 **Matrix Fundamentals for Deep Learning**

🔢 **1. What is a Matrix?**

A **matrix** is a **2D array of numbers** arranged in rows and columns.

Example:

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

🔁 2. Matrix Operations You Must Know

Operation	Description	Example
Addition	Add corresponding elements	A + B
Scalar Multiplication	Multiply every element by a number	2 × A
Matrix Multiplication	Dot product of rows and columns	A × B
Transpose	Flip rows and columns	A^T
Identity Matrix	Diagonal of 1s, rest 0	I × A = A
Inverse	A^{-1} such that A × A^{-1} = I	Only for square matrices

🔍 3. Why Matrices Matter in Deep Learning

- **Inputs** (images, text, etc.) are stored as matrices
- **Weights** in neural networks are matrices
- **Forward pass** = matrix multiplication
- **Backpropagation** = matrix calculus

🧮 Operations Explained:

1. **Addition (A + B)**
   Add corresponding elements of two matrices.
2. **Subtraction (A - B)**
   Subtract corresponding elements.
3. **Scalar Multiplication (2 × A)**
   Multiply every element in the matrix by a scalar (number).
4. **Matrix Multiplication (A @ B)**
   Multiply rows of A with columns of B using the **dot product**.

🔢 Types of Multiplication in Matrix Math

1. Scalar Multiplication

- Multiply **every element** of a matrix by a single number (scalar).

$$2 * \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

2. Element-wise Multiplication (Hadamard Product)

- Multiply **corresponding elements** of two matrices of the same shape.
- Symbol: ⊙
- Example:

$$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} \odot \begin{bmatrix}5 & 6\\7 & 8\end{bmatrix} = \begin{bmatrix}5 & 12\\21 & 32\end{bmatrix}$$

$$\begin{bmatrix}1*5 & 2*6\\3*7 & 4*8\end{bmatrix}$$

3. Matrix Multiplication (Dot Product)

- Multiply **rows of the first matrix** with **columns of the second**.
- Only possible when the **number of columns in A = number of rows in B**.

$$[1\ 2]\ \mathrm{x} \begin{bmatrix}3\\4\end{bmatrix} = [11]$$

$$[1*3+2*4=11]$$

$$\begin{bmatrix}100 & 50 & 30\\80 & 70 & 20\\60 & 40 & 90\end{bmatrix} \cdot \begin{bmatrix}1.15 & 1.05 & 0.85\\1.05 & 1.02 & 1.1\\1.1 & 1 & 0.9\end{bmatrix} = \begin{bmatrix}200.5 & 186 & 167\\21 & 32\end{bmatrix}$$

$$\begin{bmatrix}100*1.15 + 50*1.05 + 30*1.1 & 100*1.05 + 50*1.02 + 30*1 & 100*0.85 + 50*1.1 + 30*0.9\\80*1.15 + 70*1.05 + 20*1.1 & 80*1.05 + 70*1.02 + 20*1 & 80*0.85 + 70*1.1 + 20*0.9\\60*1.15 + 40*1.05 + 90*1.1 & 60*1.05 + 40*1.02 + 90*1 & 60*0.85 + 40*1.1 + 90*0.9\end{bmatrix}$$

🧠 **What is a Tensor?**

A **tensor** is a **generalized container for data** — like a matrix, but it can have more dimensions.

You can think of it like this:

Tensor Type	Shape	Example
Scalar	0D	5
Vector	1D	[5, 10, 15]
Matrix	2D	[[1, 2], [3, 4]]

Tensor Type	Shape	Example
3D Tensor	3D	Stack of matrices (e.g., RGB image)
nD Tensor	nD	Used in deep learning models

📦 **Real-Life Analogy:**

- **Scalar**: A single number (like temperature)
- **Vector**: A list of numbers (like speed in 3 directions)
- **Matrix**: A table (like a grayscale image)
- **3D Tensor**: A cube of numbers (like a color image)
- **4D+ Tensor**: A batch of images, videos, or sequences

🔍 **Why Tensors Matter in Deep Learning**

- **Inputs** (images, text, audio) are stored as tensors
- **Weights** and **biases** in neural networks are tensors
- All operations (like matrix multiplication) are done on tensors

PyTorch Tensor Basics :

Tensor is the common term for all the scalar, vector , matrix and nd matrix

Import torch

torch.tensor(9) → scalar

revenue = torch.tensor([100,23,45]) → vector

revenue.t() → to transpose

```
import torch
units = torch.tensor([
  [200,220],
  [150,180],
  [300,330]
]dtype=)
units.shape
units.device
units.view() # reshapes (-1 makes a flat list)
units[2,1] # index
```

units[2,1] = 345 # assinging the values

torch.ones

torch.zeros

Derivative (Single-variable calculus)

- A derivative measures how a function changes as its input changes.
- It tells you the slope or rate of change of a function at a specific point.
- You measure the non-linear curve in the graph using derivatives
- Slope is a constant and used for linear equation
- Derivative is a function and use for non linear equation

Example:

If $f(x)=x^2$ then the derivative is $f'(x)=2x$

This means at any point *x*, the slope of the curve is 2x.

If: $f(x)=x^3$ Then the **derivative** is: $f'(x)= 3x^2$

🔍 **What does this mean?**

- For **every small change in *x***, the change in *y* (i.e., the slope of the curve) is approximately $\mathbf{3x^2}$.
- So, at:
  - $x = 1x = 1, the\ slope\ is\ 3(1)2 = 3$
  - $x = 2x = 2, the\ slope\ is\ 3(2)2 = 12$
  - $x = 3x = 3, the\ slope\ is\ 3(3)2 = 27$

  This tells you how **steep** the curve is at any point.

Partial Derivative (Multivariable calculus)

- A partial derivative is used when a function has more than one variable.
- It measures how the function changes with respect to one variable, while keeping the others constant.

Example:

If $f(x,y) = x2 + y2$ then Partial derivative with respect to $x = \frac{\partial x}{\partial f} = 2x$

Partial derivative with respect to *y*: $\partial y/\partial f=2y$

$$Cost(m,l) = 3 * m2 + 2 * l2 + 5 * m * l + 23$$

$$\frac{\partial cost}{\partial m} = 6 * m + 5 * l$$

$$\frac{\partial cost}{\partial l} = 4 * l + 5 * m$$

✅ **Here's the key idea:**

When you're taking the partial derivative with respect to *m*:

- You treat l as a constant.
- So, you only differentiate terms that contain m.
- Any term that does not contain *m* is treated as a constant and its derivative is 0.

Chain Rule:

The chain rule is a fundamental concept in calculus used to compute the derivative of a composite function — that is, a function inside another function.

🔗 **Chain Rule Formula**

If you have a function: $\boldsymbol{y = f(g(x))}$

Then the derivative is: $\boldsymbol{\frac{\partial x}{\partial y} = f'(g(x)) \cdot g'(x)}$

🧠 **Intuition:**

You're **chaining** the derivatives together:

- First, take the derivative of the **outer function** (with the inner function still inside).
- Then multiply it by the derivative of the **inner function**.

🔍 **Example:**

Let's say: $y = (3x + 2)4y = (3x + 2)4$

This is a composite function:

- **Outer function**: $f(u) = u4$
- **Inner function**: $g(x) = 3x + 2g(x) = 3x + 2$
- Using the chain rule: $\frac{dy}{dx} = 4(3x + 2)^3 \cdot 3 = 12(3x + 2)3$

🔁 **What is Autograd?**

Autograd is PyTorch's way of **automatically computing gradients of tensors**. It builds a dynamic computation graph on-the-fly as operations are performed, which is then used to compute derivatives during backpropagation.

🧱 **Core Components**

Component	Description
Tensor	Main data structure. Set requires_grad=True to track operations.
grad	Stores the gradient of the tensor after .backward() is called.
grad_fn	References the function that created the tensor (if any).
.backward()	Computes gradients for all tensors in the graph.
with torch.no_grad()	Context manager to disable gradient tracking (useful for inference).

Numpy Arrays vs PyTorch Tensors:

PyTorch tensors and numpy arrays have similar functionality but tensor offers 3 key benefits over numpy arrays that are useful in deep learning.

- Benefit 1: Tensor come within built support to leverage GPU acceleration.
- Benefit 2: Tensors have autograd features that computes gradients automatically. Numpy arrays do not have this feature.
- Benefit 3: Tensors are tightly integrated with PyTorch ecosystem that makes it easier to use with deep learning tasks.

Neural Networks training:

🧠 **1. Define the Neural Network**

You create a model by subclassing torch.nn.Module and defining layers in __init__ and the forward pass in forward**.**

```
import torch
import torch.nn as nn
import torch.nn.functional as F
class SimpleNet(nn.Module):
  def __init__(self):
    super(SimpleNet, self).__init__()
    self.fc1 = nn.Linear(2, 4)  # Input layer
    self.fc2 = nn.Linear(4, 1)  # Output layer
  def forward(self, x):
    x = F.relu(self.fc1(x))
```

```
        x = self.fc2(x)
        return x
```

📦 2. Prepare Data

Use torch.utils.data.DataLoader to load and batch your dataset.

```
from torch.utils.data import DataLoader, TensorDataset
# Dummy data
X = torch.tensor([[0.0, 0.0], [1.0, 1.0]], dtype=torch.float32)
y = torch.tensor([[0.0], [1.0]], dtype=torch.float32)
dataset = TensorDataset(X, y)
loader = DataLoader(dataset, batch_size=1, shuffle=True)
```

⚙️ 3. Set Loss Function and Optimizer

```
model = SimpleNet()
criterion = nn.MSELoss()  # Mean Squared Error for regression
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

🔁 4. Training Loop

```
for epoch in range(100):
  for inputs, targets in loader:
    # Forward pass
    outputs = model(inputs)
    loss = criterion(outputs, targets)
    # Backward pass and optimization
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
  if epoch % 10 == 0:
    print(f"Epoch {epoch}, Loss: {loss.item():.4f}")
```

✅ 5. Evaluation

Use torch.no_grad() to evaluate the model without tracking gradients:

```
with torch.no_grad():
  test_input = torch.tensor([[1.0, 1.0]])
  prediction = model(test_input)
```

print("Prediction:", prediction.item())

🔁 Training Through Backpropagation

Backpropagation is the **learning algorithm** used to train neural networks by adjusting weights to minimize error.

🧠 Step-by-Step Breakdown

1. Forward Pass

- Input data flows through the network.
- Each neuron computes:

$$z = w \cdot x + b \; then \; a = activation(z)$$

- Final output is compared to the true label using a **loss function**.

2. Loss Calculation

- Measures how far the prediction is from the actual value.
- Common loss: **Mean Squared Error**, **Cross-Entropy**

3. Backward Pass (Backpropagation)

- Calculates the **gradient** of the loss with respect to each weight using the **chain rule**.
- Gradients tell us **how much each weight contributed to the error**.

4. Weight Update (Gradient Descent)

- Weights are updated to reduce the error:

$$w = w - \eta \cdot \frac{\partial L}{\partial w}$$

where:

- η = learning rate

$\frac{\partial L}{\partial w}$ = gradient of loss w.r.t. weight

🔂 This process repeats for many epochs until the model learns.

Example:

Will score century or not

Hand eye coordination(x1) Diet(x2) fitness routine(x3) height(x4) + bias → Input layer

$$h_1 = \sigma\left(w_{11} * x_1 + w_{12} * x_2 + w_{13} * x_3 + w_{14} * x_4 + w_{15} * x_5 + b\right)$$

$$Running = \sigma(0.7 * Hand\ Eye\ Cordination + 2.1 * Diet + 2.7 * Fitness\ Routine + \ldots)$$

Running(h_1) Timing(h_2) Technique (h_3) muscle power(h_4) Luck(h_5) → Hidden layer

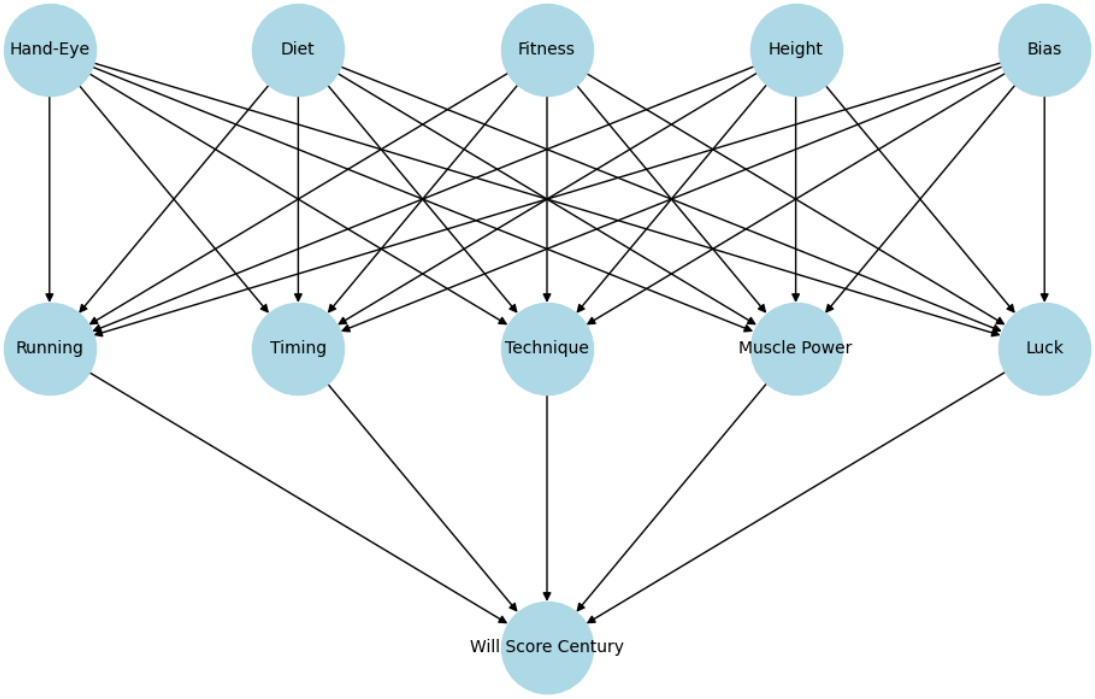
Each input feature is connected with every other hidden layer with weights

And these hidden layers weights is calculated and based on the total you will get the out put

$$y = \sigma\left(w_1 * h_1 + w_2 * h_2 + w_3 * h_3 + w_4 * h_4 + w_5 * h_5 + b\right)$$

$$y = \sigma\left(1.1 * h_1 + 3 * h_2 + 4 * h_3 + 2.7 * h_4 + 0.1 * h_5 + 1.2\right)$$

$$Will\ Score\ Century\ (y) = \sigma(1.1 * Running + 3 * Timing + 4 * Technique + 2.7 * Muscle\ Power + .. + 1.2)$$



Error1 = $(y(1) - \hat{y}_1(over\ all\ calculation\ of\ hidden\ layer))^2$ for one sample(record)

Total Error = $error1 + error2 + \cdots + error\ N$

$$\boldsymbol{Total\ Error} = \sum_{i=1}^{n} (\boldsymbol{y} - \hat{\boldsymbol{y}}_i)^2$$

Step3:

Back propagation:

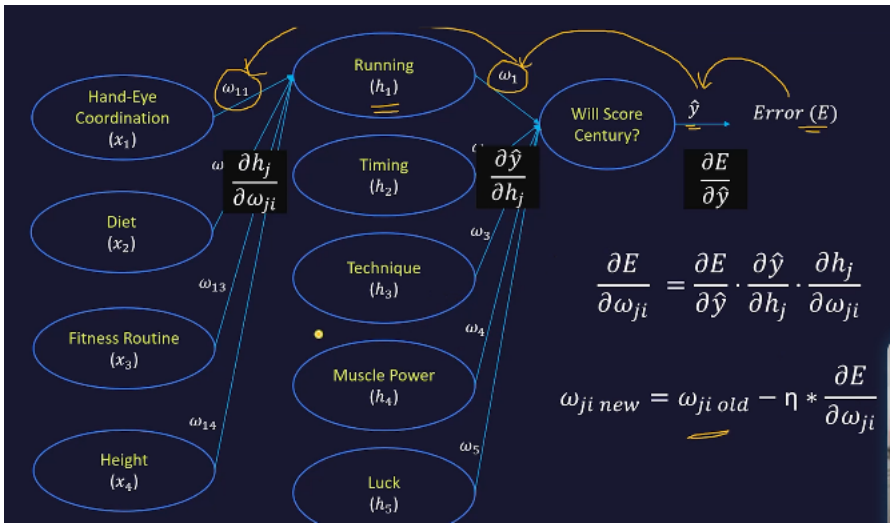
After calculating the error how much error change in the unit change like nobs

$$\frac{\partial E}{\partial w_j} = \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_j}$$

How much change in the error by the change in the weight (E) error wj over all weights

$$w_{jnew} = w_{jold} - \eta \cdot \frac{\partial E}{\partial w_j}$$

Error gets back propagated



Step 4: repeat this until minimize the error called epoch

Summary

1. Initialize neural network with random weights
2. Feed forward training samples and calculate prediction error
3. Back propagate error to adjust weights using gradient descent
4. Repeat the process until certain number of iterations (epochs) or error is reduced significantly
5. Evaluate Neural Network Performance by using test or validation set

🧠 What is Gradient Descent?

Gradient Descent is an optimization algorithm used to **minimize a loss function** by iteratively updating the model's parameters (weights and biases).

📉 The Core Idea

Imagine you're standing on a hill and want to reach the **lowest point** (the valley). You take steps **in the direction of the steepest descent** — that's what gradient descent does!

🧮 The Math Behind It

Let's say:

- $L(w)$ is the **loss function** (depends on weights w*w*)

- $\nabla L(w)$is the **gradient** (slope) of the loss

🔁 **Update Rule:**

$$w := w - \eta \cdot \nabla L(w)$$

Where:

- w = current weight
- η = learning rate (step size)
- $\nabla L(w)$= gradient of the loss with respect to w

📌 **Key Concepts**

Concept	Meaning
Gradient	Direction and rate of fastest increase of the loss
Learning Rate (η)	Controls how big each step is
Convergence	When the loss stops decreasing significantly
Local Minima	A point where the loss is lower than nearby points
Global Minimum	The absolute lowest point of the loss function

📊 **Types of Gradient Descent**

Type	Description
Batch GD	Uses the whole dataset for each update
Stochastic GD (SGD)	Uses one sample at a time
Mini-Batch GD	Uses small batches (most common in DL)

📘 **Gradient Descent: Theory Explained**

A trial-and-error method to find the optimal value for m(slope) and b(intercept)

Gradient Descent is an optimization algorithm used to minimize a cost function by iteratively moving in the direction of the steepest descent (i.e., the negative gradient).

Minimum value

The purpose of partial derivate is to measure how a function changes as of its variable is varied while keeping the other variable constant.

$$\boldsymbol{mse} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

$$\boldsymbol{mse} = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \left(m_{x_i} + b\right)\right)^2$$

How function mse changing with respect to m

$$\frac{\partial \boldsymbol{mse}}{\partial \mathrm{m}} = \frac{2}{n}\sum_{i=1}^{n} -x_i\left(y_i - \left(m_{x_i} + b\right)\right)$$

How function mse changing with respect to b

$$\frac{\partial \boldsymbol{mse}}{\partial \mathrm{b}} = \frac{2}{n}\sum_{i=1}^{n} -\left(y_i - \left(m_{x_i} + b\right)\right)$$

For $\frac{\partial \boldsymbol{mse}}{\partial \mathrm{b}}$ we will have an initial value b1 $b2 = b1 - \frac{\partial \boldsymbol{mse}}{\partial \mathrm{b}} * \text{learining rate}(\alpha)$

Learning rate is direction

Gradient Descent: PyTorch Implementation

```
import torch


# Extract features and target from the DataFrame

performance = torch.tensor(df['performance'].values, dtype=torch.float32) # torch take numpy array by taking the df[feature].value gives the np array

years_of_experience = torch.tensor(df['years_of_experience'].values, dtype=torch.float32)

projects_completed = torch.tensor(df['projects_completed'].values,dtype=torch.float32)

bonus = torch.tensor(df['bonus'].values, dtype=torch.float32)
```

Step -1 : Initialize neural network with random weights

Epochs and learning rate too

```
w1 = torch.rand(1, requires_grad=True)

w2 = torch.rand(1, requires_grad=True)

w3 = torch.rand(1, requires_grad=True)

bias = torch.rand(1, requires_grad=True)


# Learning rate and number of iterations

learning_rate = 0.006

epochs = 5000
```

step 2. Feed forward training samples and calculate prediction error

Step 3: Training loop for gradient descent

```
for epoch in range(epochs):

  # Compute the predicted bonus using the current weights and bias

  predicted_bonus = w1 * performance + w2 * years_of_experience + w3 *
projects_completed + bias

  # Compute the Mean Squared Error (MSE) loss

  loss = ((predicted_bonus - bonus) ** 2).mean() # error

  # Perform backpropagation to compute gradients of the loss with respect to
w1, w2, w3, and bias

  loss.backward()

  # Update the weights and bias using the computed gradients

  with torch.no_grad():

    w1 -= learning_rate * w1.grad

    w2 -= learning_rate * w2.grad

    w3 -= learning_rate * w3.grad

    bias -= learning_rate * bias.grad

  # Zero the gradients after updating

  w1.grad.zero_()

  w2.grad.zero_()

  w3.grad.zero_()

  bias.grad.zero_()

  # Print the loss at regular intervals

  if (epoch + 1) % 100 == 0:
```

print(f"Epoch [{epoch + 1}/{epochs}], Loss: {loss.item():.4f}")

1. Batch Gradient Descent (BGD)

Batch Gradient Descent is an optimization algorithm that updates model parameters by calculating gradients across the entire dataset in each iteration minimize a loss function.

- **How it works**: Uses the **entire training dataset** to compute the gradient of the loss function.
- **Pros**:
    - Stable convergence.
    - Deterministic updates.
- **Cons**:
    - Very slow for large datasets.
    - Requires a lot of memory.
- **Use case**: When the dataset is small and fits in memory.

2. Stochastic Gradient Descent (SGD):

Stochastic Gradient Descent (SGD) updates model parameters using gradient calculated using a single, random data point per iteration, adding noise but speeding up convergence

- **How it works**: Updates the model parameters using **one training example at a time**.
- **Pros**:
    - Fast updates.
    - Can escape local minima due to its noisy updates.
- **Cons**:
    - High variance in updates can lead to oscillations.
    - Less stable convergence.
- **Use case**: Online learning or very large datasets.

1. Mini-Batch Gradient Descent

Mini Batch Gradient Descent is a variant of batch gradient descent that updates model parameters by calculating gradients using a small subset of the dataset balancing efficiency and stability

- **How it works**: Uses a **small subset (mini batch)** of the training data to compute the gradient.
- **Pros**:

    - Combines the advantages of BGD and SGD.
    - Faster convergence than BGD.
    - More stable than SGD.
    - Efficient use of vectorized operations on GPUs.
- **Cons**:
    - Still requires tuning of batch size.
- **Use case**: Most commonly used in deep learning frameworks.

Comparison Table:

Feature	Batch GD	Mini-Batch GD	SGD
Data per update	All data	Subset (e.g., 32)	One sample
Speed per update	Slow	Moderate	Fast
Convergence stability	High	Moderate	Low (noisy)
Memory usage	High	Moderate	Low
Parallelization	Poor	Good	Poor
Common in practice	Rare	Very common	Sometimes (online)

Neural Networks in Pytorch:

Import Libraires

Split train test data

Convert train test data into tensors

NN class for the problem

Create a sequential network

With input layers(input features and output neuron)

Model = class name

Criterion = loss function

Optim = optimizer (model. Parameters,lr=)

Epochs

Loop to run the

1.Forward pass

2. backward pass

3.update weights

Model Evaluation

Datasets and Data Loaders:

Predefine Datasets

Data Loader segregates to batches

torch.Size([64, 1, 28, 28]) → a batch of 64 images with 1(cmap) and 28X28 pixels

Handwritten Digits Classification:

Step-1:

For every image of its size pixels(28X28) we have to convert it into a flat list for the input layer

Normalize the values in range [0,2) for zero cantered makes it more convenient to train

```
transform = transforms.Compose([
    transforms.ToTensor(), # 0--> 255 ==> [0,1)
    transforms.Normalize((0.5,),(0.5,)), # [0,1) ==> [-1,1]
])
# normalize = (pixel-mean)/std
training_data = datasets.MNIST(
    root = 'data',
    download=True,
    train= True,
    transform=transform
)
test_data = datasets.MNIST(
    root = 'data',
    download=True,
    train= False,
    transform=transform
)
train_dataloader = DataLoader(training_data,batch_size=64,shuffle=True)
```

```
test_dataloader = DataLoader# neural network class


class DigitsClassifier(nn.Module):
  def __init__(self):
    super().__init__()
    self.network  = nn.Sequential(
    nn.Flatten(),  # flattening to array
    nn.Linear(28*28, 128), # input layer to hidden layer
    nn.ReLU(), # activation function
    nn.Linear(128,64), # hidden layer of 128 input and 64 output
    nn.ReLU(),# activation function
    nn.Linear(64,10) # output layer
  )
  def forward(self,x):
    return self.network(x)


(test_data,batch_size=64,shuffle=True)
model = DigitsClassifier()
criterion = nn.CrossEntropyLoss() # for binary classification
optimizer = optim.Adam(model.parameters(),lr=0.001)
epochs = 5
for epoch in range(epochs):
  for images, labels in train_dataloader:
    # 1.forward_pass
    outputs = model(images)
    loss = criterion(outputs,labels)


    running_loss = loss.item()
    # 2. backward pass
    optimizer.zero_grad()
    loss.backward()
```

```
        # weight updates
        optimizer.step()
    print(f'Epoch[{epoch+1}/{epochs}],Loss: {running_loss/len(train_dataloader)}')


model.eval()
total = 0
correct = 0
with torch.no_grad():
    for images, labels in test_dataloader:
        outputs  = model(images)
        _, predicted = torch.max(outputs.data,1)
        correct+= (predicted == labels).sum().item()
        total += labels.size(0)
print(f'Accuarcy on the test set:{100 * correct/ total:.2f}%')
```

Cost Function: Binary Cross Entropy (a.k.a. Log Loss):

$$cost(MSE) = \frac{1}{n}\sum_{i=1}^{n}(y - \hat{y}_i)^2$$

MSE wont work because of **non-convex function** where you will both **global and local minima** your best fit line it wont properly **fits**

To overcome the Issue, we use the **Binary cross Entropy (Log Loss)**

Y= actual value, p = predicted probability

$$log\ loss\ = -(y * log(p) + (1 - y) * log(1 - p))$$

$$When\ y\ =\ 1\ \ log\ loss\ = -log(p)$$

$$When\ y\ =\ 0\ log\ loss\ = -log(1 - p)$$

$$Log\ Loss = -\frac{1}{n}\sum_{i=1}^{n} y_i * log(p_i) + (1 - y_i) * log(1 - p_i)$$

Reasons for using Binary cross Entropy

Aligns perfectly with probabilistic outputs, providing a natural fit for binary outcomes

Produces convex cost function(when used with sigmoid) which is good for global minimum convergence)

Provides strong gradient updates, especially for confident, incorrect predictions

Incorrect predictions are penalized logarithmically, encouraging accuracy and discouraging overconfidence in errors.

- Cost functions like MSE may not work well for binary classification problems as the cost surface will not be convex, and you may get stuck in local minima
- Binary cross entropy (BCE) along with sigmoid activation gives a smooth, convex surface for cost function which makes convergence easier.
- BCE also penalizes high confidence errors which eventually helps in efficient discovery of global minimum.
- BCE aligns perfectly with probabilistic outputs, providing a natural fit for binary outcomes.

Cost Function: Cross Entropy (Multi class)

$$\boldsymbol{H(p,q) = -\sum_{x\epsilon\chi} p(x) * log(q(x))}$$

P = Actual labels(true distribution of data)

q = predicted probabilities

$\boldsymbol{\chi = All\ the\ classes}$

$\boldsymbol{H(p,q) = -\ (y * log(p) + (1-y) * log(1-p))}$

Model Optimization : Training Algorithm

Model Optimization is a process of finding the best way to train a model such that we can train it faster by using less compute resources and the model performs well during prediction phase.

Training Algorithms(optimizers)

Gradient Descent

GD with Momentum

RMSProp

Adam

Regularization Techniques

Hyper parameters

Training Algorithms(optimizers):

Exponentially Weighted Moving Average:

An average of last N values

$$v_{new} = \beta \cdot v_{old} + (1-\beta) \cdot x$$

$$\beta = decay\ factor$$

First do the average of all the previous values then do the weighted moving average

- Exponentially Weighted Moving Average (EWMA) gives more weight to recent data, smoothing out fluctuations over time.
- The decay factor (ß) controls how much past data influences the current average, with higher values giving longer memory. People tend to use 0.9 value for this factor a lot.

Gradient Descent with Momentum:

$W_{new} = W_{old} - \eta \cdot \frac{\partial E}{\partial \omega}$ → without momentum

In the GD momentum instead of current gradient we take previous gradients exponentially weighted moving average

$$v_{new} = \beta \cdot v_{old} + (1 - \beta) \cdot \frac{\partial E}{\partial \omega}$$

We have to keep the learning rate in control.

$$\beta = 0.9$$

- GD with momentum accelerates convergence by building on past gradients, reducing the time to reach the minimum.
- The momentum term smooths out oscillations, especially in regions with steep, narrow valleys, leading to a more stable optimization path
- Momentum helps GD escape small local minima and flat regions, making it more effective in complex loss landscapes.
- The momentum coefficient (B) controls how much influence previous gradients have on the current update.
- GD with momentum is widely used in deep learning to achieve faster and more reliable training.

RMSprop: Root mean square

Optimizers or Technique or algorithm to Find the global minimum.

$$v_{new} = \beta \cdot v_{old} + (1 - \beta) \cdot x^2$$

$$W_{new} = W_{old} - \frac{\eta}{\sqrt{v_{new}}} \cdot g$$

$$W_{new} = W_{old} - \frac{\eta}{\sqrt{v_{new} + \epsilon}} \cdot g$$

$$\beta = 0.999$$

- RMSProp uses a exponentially weighted moving average of squared gradients to reduce oscillations.
- Helps models converge faster and works well with noisy gradients.

Adam:

Combination of GD Momentum and RMSprop

$$W_{new} = W_{old} - \frac{\eta}{\sqrt{v_{new} + \epsilon}} \cdot v_{new}$$

- Adam combines momentum and RMSProp for efficient updates.
- Tracks both mean and squared gradients to stabilize weight updates.

Optimizers in Action:

Do trial and error with optimizers for the model

Model Optimization Regularization Techniques:

Regularization technique is used to reduce the overfitting

Overfitting: Training the model too much it passes through all the data points but fail at the test data

Underfitting: A Simple model where train and test error is high

Dropout Regularization:

Randomly dropping the hidden layer neurons due to this we don't have dependency of single neuron [YOU DON'T HAVE TO DEPEDNCY ON ONE SINGLE] More robust

Add dropout in nn.Dropout(=0.5)

- Dropout regularization drops certain neurons in each hidden layer during the training process. This generalizes the model and stops the network from learning specific details of training samples.
- Dropout ratio can be 50%, 20%, 30% etc. based on the situation. In PyTorch one can add a dropout layer after activation layer using nn.Dropout(p=0.5) where p indicates the percentage of neurons being dropped out. 0.5 means drop 50% of the neurons.

L1 Regularization:

$$E = E + \lambda \sum_{i=1}^{n} w_i \quad \lambda = configurable\ paramater$$

$$W_{new} = W_{old} - \eta \left(\frac{\partial \mathrm{E}}{\partial \omega_{old}} + \lambda * sign(w_{old}) \right.$$

L2 Regularization:

$$E = E + \lambda \sum_{i=1}^{n} w_i^2 \quad \lambda = configurable\ paramater$$

$$W_{new} = W_{old} - \eta \left(\frac{\partial \mathrm{E}}{\partial \omega_{old}} + \lambda w_{old}^2 \right)$$

- Both L1 and L2 help prevent overfitting by adding a penalty to the cost function for large weights

Batch Normalization:

Batch normalization (BN) is a technique used in training deep neural networks to stabilize and accelerate the learning process. It also helps with regularization.

$$x_{norm} = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

$$y = x_{norm} * \gamma + \beta$$

$$\gamma = scale$$

$$\beta = shift$$

Stabilize learning , Higher Learning rate, Regularization effect

- Batch normalization normalizes layer inputs to have zero mean and unit variance, enhancing model performance.
- Allows for higher learning rates, reducing the training time for deep networks.
- Adds robustness to model initialization, making it less sensitive to initial weights.

Early Stopping:

See the value accuracy if there is no further improvement in 5 scores then stop the model and whatever the higher accurate value take that

Save the model using torch.save(model.pt)

Model Optimization Hyperparameter Tuning:

What is Hyperparameter Tuning?

All the weights and bias are called the parameters

Hyperparameter are learning Rate, batch size, number of layers, regularization param(lambda),epochs, number of Neurons in each Layers, there are manual user should tune in order to optimize the model

Benefits:

Improve model accuracy, reduce overfitting underfitting, optimize training time, compute resources

GridSearchCV:

For each parameter(hyper) it tries every permutation and combination with other parameters(hyper)

RandomSearchCV:

Due to high volume data in Deep Learning mostly people uses the RandomSearchCV because it saves time and money.

- GridSearchCV exhaustively searches over specified hyperparameter values, ensuring the best combination but can be computationally expensive.
- RandomSearchCV samples a random subset of parameter combinations, making it faster and more efficient for large search spaces.
- Both methods use cross-validation to assess model performance, reducing the risk of overfitting to the training data.
- Choosing between GridSearchCV and RandomSearchCV depends on the time available and the size of the hyperparameter space.

Optuna:

Bayesian Optimization:

Gradient Optimization:

Evolutionary Algorithms:

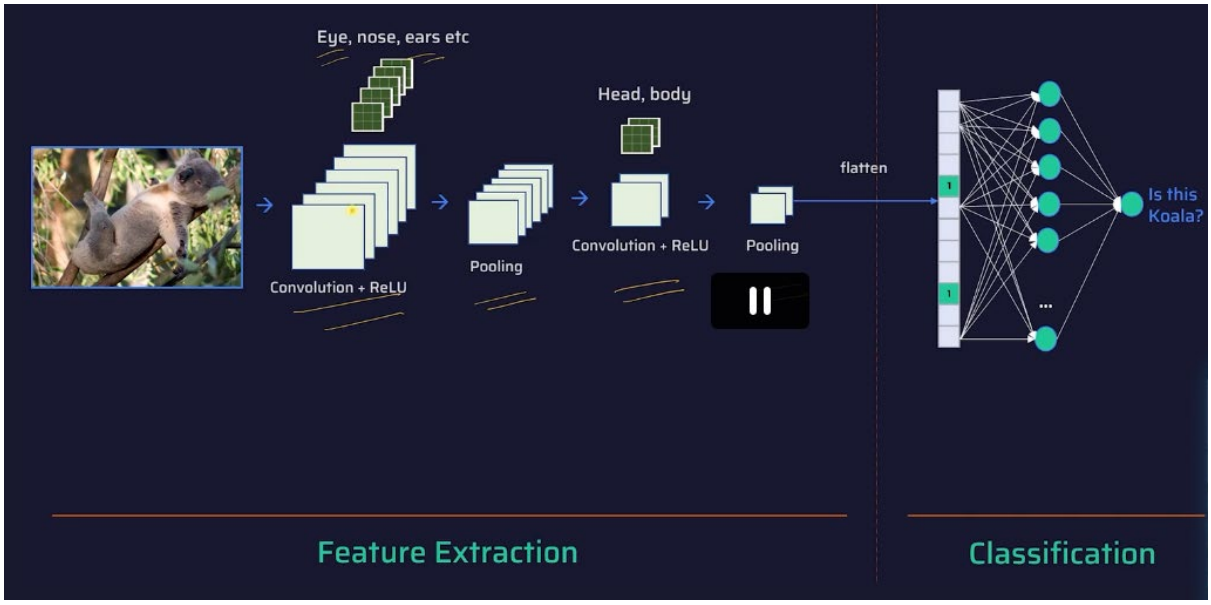
In objective trail function we can add all the learning rates and hidden dim to carry out trails

```
def objective(trial):
learning_rate = trial. suggest_loguniform('learning_rate', 1e-4, 1e-1)
hidden_dim = trail.suggest_int('hidden_dim', 16, 128)
model = SimpleNN(input_dim=20, hidden_dim=hidden_dim)
criterion = nn. CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
# Training Loop
        epochs = 20
```

- Optuna is a modern, automated hyperparameter optimization framework that uses an efficient, trial-based search.
- It leverages techniques such as bayesian optimization to find optimal hyperparameters faster than grid or random search.
- Optuna allows for dynamic pruning, stopping unpromising trials early to save computation.
- Ideal for deep learning tasks with large search spaces, where traditional tuning methods may be inefficient.

Convolutional Neural Networks (CNN):

What is CNN? Convolution, Kernels, Pooling and Beyond:



🧠 What is a CNN?

A CNN is designed to automatically and adaptively learn spatial hierarchies of features from input images. It consists of layers that apply convolutions, non-linear activations, and pooling operations to extract and reduce features.

🔍 1. Convolution

- Purpose: Extract features like edges, textures, shapes(loopy circle pattern).
- How it works: A small matrix called a kernel (or filter) slides over the image and performs element-wise multiplication and summation.(convolution operation)
- Result: A feature map that highlights specific patterns.
- Location invariant (don't depend on the location)

Example:

If you have a 5×5 image and a 3×3 kernel, the kernel slides over the image and produces a smaller output (e.g., 3×3 feature map).

🧩 2. Kernels (Filters)

- Definition: Small matrices (e.g., 3×3, 5×5) used in convolution.
- Learned during training: The values in the kernel are adjusted to detect useful features.
- Multiple kernels: Each detects different features (e.g., vertical edges, horizontal edges).

🏊 3. Pooling (Subsampling)

- Purpose: Reduce the spatial size of the feature maps, making the model faster and less prone to overfitting.
- Types:
    - Max Pooling: Takes the maximum value in a region.

    - Average Pooling: Takes the average value in a region.
- Benefits of pooling: Reduces Dimensions, computation, overfitting , model is tolerant towards variations, distortions

Example:

A 2×2 max pooling on a 4×4 feature map reduces it to 2×2 by taking the max value in each 2×2 block.

🧱 **CNN Architecture Overview**

1. Input Layer: Raw image (e.g., 28×28 pixels).
2. Convolution Layer: Applies filters to extract features.
3. Activation Function: Usually ReLU (adds non-linearity).
4. Pooling Layer: Down samples the feature maps.
5. Fully Connected Layer: Final classification or regression.
6. Output Layer: Produces predictions (e.g., class probabilities).

Image size = 1920X1080 X3 is a typical hd image 3 is a RGB channel from range(0-255)

First layer neurons == 1920 X 1080 X3 -6 million

Hidden layer neurons = let's say you keep it – 1000 neurons

Weights between input and hidden layer = 6 billion

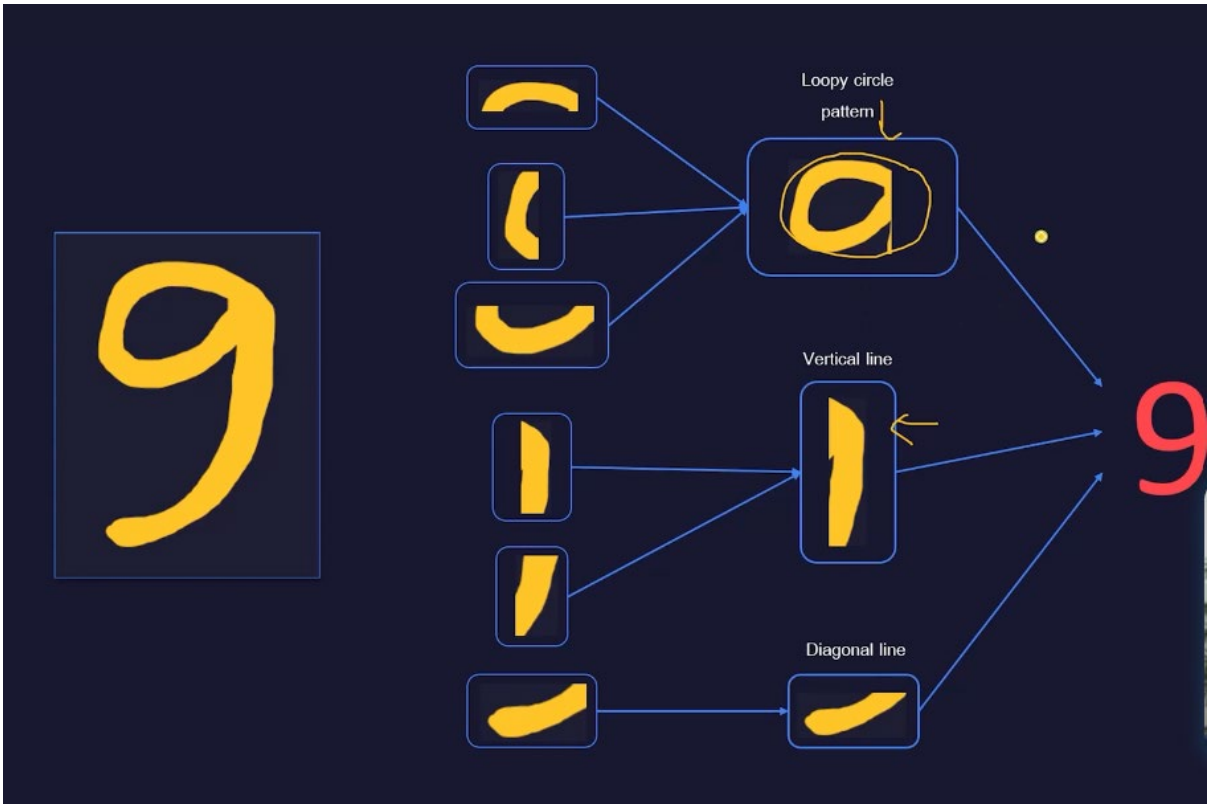
Disadvantage of using FCN(fully connected Neural Net ANN, Multilayer perceptron's

) for image classification

Too much computation

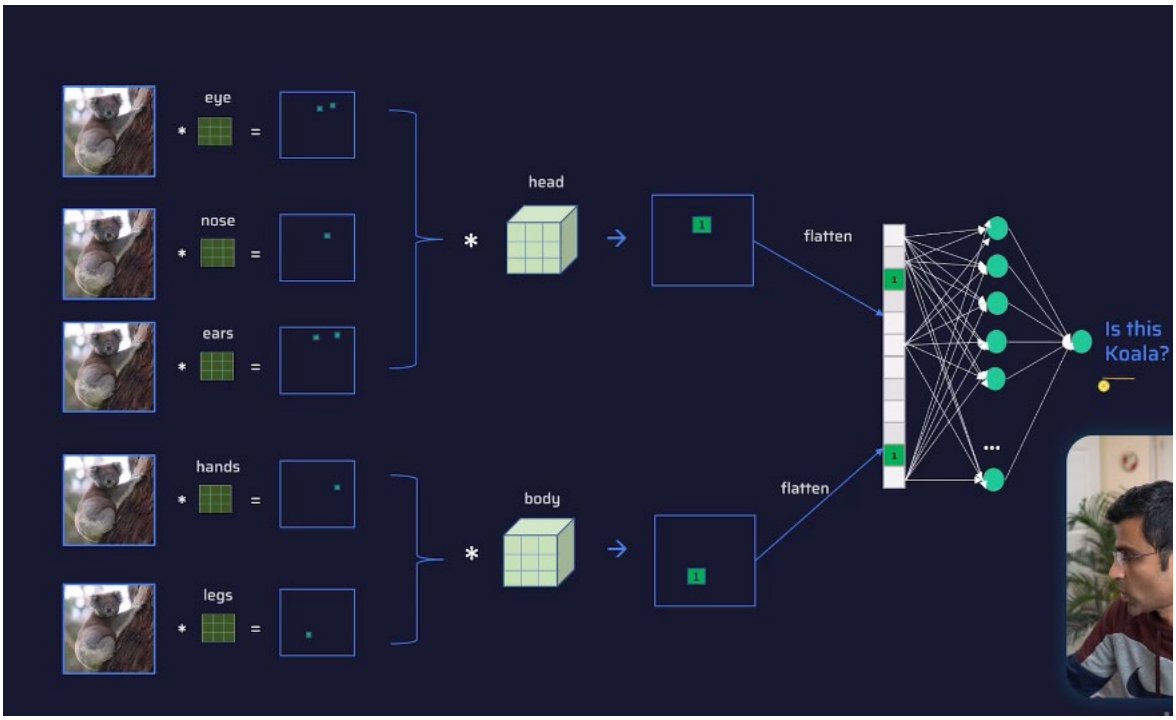
Loss of spatial information

It doesn't capture the relation of an image



For the digit 9 image we can have 3 filters for each loopy circle pattern, straight line pattern, diagonal pattern and it generates a feature maps of each convolution operations

In koalas we can have eyes, nose, ears convolution operations all 3 combines and forms into a filter for head and again this is another convolution operation 2



Above is feature extraction and classification(from flattened) to finding

FCN helps to recognises the features

🧱 1. Padding

Padding refers to adding extra pixels (usually zeros) around the border of the input image **before** applying the convolution operation.

🔹 Why use padding?

- To **preserve the spatial dimensions** of the input (e.g., keep width and height the same).
- To allow the kernel to **fully cover the edges** of the image.

🔹 Types of Padding:

- **Valid Padding** (padding=0): No padding. Output size shrinks by 2
- **Same Padding** (padding='same' or calculated): Pads so output size = input size.

📐 Example:

If you have a 5×5 image and a 3×3 kernel:

- Without padding → output is 3×3.
- With padding of 1 → output is 5×5.

🏃 2. Stride

Stride is the number of pixels the kernel moves (or "strides") over the input image.

🔹 Effects of stride:

- **Stride = 1**: Kernel moves one pixel at a time → more overlap → larger output.
- **Stride = 2**: Kernel skips every other pixel → smaller output → faster computation.

📐 Example:

For a 5×5 image and a 3×3 kernel:

- **Stride 1** → output is 3×3.
- **Stride 2** → output is 2×2.

🧮 Output Size Formula

For a 2D convolution:

$$\text{Output Size} = \left[\frac{(W - K + 2P)}{S}\right] + 1$$

Where:

- W = input width/height
- K = kernel size
- P = padding
- S = stride

CIRF10

```
class CNN(nn.Module):
  def __init__(self, *args, **kwargs):
    super().__init__(*args, **kwargs)
    self.network = nn.Sequential(
      nn.Conv2d(3,32,kernel_size=(3,3),padding='same'), # output = 32 32 32
      nn.ReLU(),
      nn.MaxPool2d(kernel_size=(2,2),stride=(2,2)),# output (32,16,16) only spatial size gets shrinks
      nn.Conv2d(32,64,kernel_size=(3,3)),# output(64,14,14)  due to no padding(default) shrinks by -2
      nn.ReLU(),
      nn.MaxPool2d(kernel_size=(2,2),stride=(2,2)),# output (64,7,7)
    )
    self.fc_layers = nn.Sequential(
      nn.Flatten(),
      nn.Linear(64*7*7,600),
      nn.ReLU(),
      nn.Linear(600,120),
      nn.ReLU(),
      nn.Linear(120,10)
    )
  def forward(self,x):
    x= self.network(x)
    x= self.fc_layers(x)
    return x
```

Data Augmentation:

```
transform = transforms.Compose([
  transforms.RandomHorizontalFlip(),
  transforms.RandomRotation(15),
  transforms.ColorJitter(brightness=0.2,contrast=0.2,saturation=0.2),
  transforms.ToTensor(),
```

transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))

])

Transfer Learning:

Transfer Learning is a powerful technique in deep learning where a model developed for one task is reused as the starting point for a model on a second, related task.

🔁 **Why Use Transfer Learning?**

Training deep neural networks from scratch requires:

- A large dataset
- High computational power
- Time

Transfer learning helps by leveraging pre-trained models (like ResNet, VGG, BERT) that have already learned useful features from large datasets (like ImageNet or Wikipedia).

🧠 **How It Works**

1. Pretraining: A model is trained on a large dataset (e.g., ImageNet for images).
2. Transfer: The learned weights (features) are reused for a new task.
3. Fine-tuning:
    - Freeze early layers (generic features like edges, textures).
    - Retrain later layers (task-specific features).

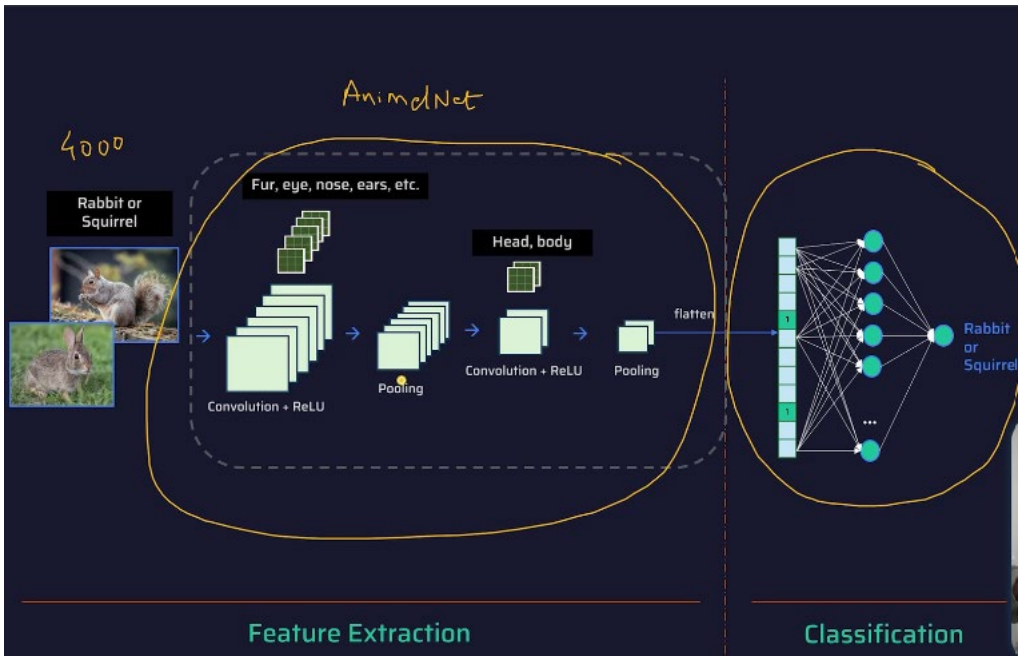
🧱 Common Scenarios

Scenario	What You Do
Small dataset, similar task	Freeze most layers, retrain last few
Large dataset, similar task	Fine-tune more layers
Different task	Use model as a feature extractor

🔍 **Popular Pretrained Models**

- **Image**: ResNet, VGG, EfficientNet, MobileNet
- **Text**: BERT, GPT, RoBERTa
- **Audio**: Wav2Vec, HuBERT

If you take any existing pretrained model their feature extraction part is frozen and you need to train the classification.



Other cases people freeze the only some features in some cases no freezing based on the problem

- Transfer learning leverages pre-trained models to solve new, related tasks with limited data.
- It significantly reduces training time by reusing learned features from large datasets.
- Commonly used in tasks like image classification and natural language processing to achieve high accuracy with minimal effort.
- Transfer learning involves fine-tuning a pre-trained model or using it as a fixed feature extractor.
- Ideal for scenarios with limited data, enabling effective learning without starting from scratch.

Pre-trained Models – ResNet, EfficientNet, MobileNet etc.

Model	Year Introduced	Key Feature	Variants	Use Cases
AlexNet	2012	First breakthrough in deep learning on ImageNet; simple architecture.	None	Benchmarking, educational purposes.
ResNet	2015	Residual connections for training very deep networks.	ResNet-18, -34, -50, -101, -152	General-purpose, feature extraction.
EfficientNet	2019	Scales depth, width, and resolution efficiently for better accuracy.	EfficientNet-B0 to EfficientNet-B7	Image classification, efficient inference.
MobileNet	2017	Lightweight network using depthwise separable convolutions.	MobileNetV1, V2, V3	Mobile apps, IoT devices.
YOLO	2016	Real-time object detection with a single neural network for predictions.	YOLOv3, v4, v5, v8	Object detection, localization.

Caltech101 Classification Using Transfer Learning:

From trochvision import models

Can do with hugging face

Sequential Models:

A **Sequential Model** in deep learning—especially in frameworks like **Keras** (part of TensorFlow)—is a **linear stack of layers**, where each layer has **exactly one input tensor and one output tensor**. It's ideal for building simple neural networks where layers are added one after another. Refers to data where the order of elements matters, such as time series, text, audio, video, etc.

✅ **When to Use Sequential Models**

- When your model has **one input and one output**.
- When layers are stacked **in a straight line** (no branching or merging).
- For tasks like:
    - Image classification
    - Simple regression
    - Binary classification

❌ **When Not to Use**

- If your model has **multiple inputs or outputs**.
- If you need **shared layers**, **residual connections**, or **attention mechanisms**.
- In such cases, use the **Functional API** or **Subclassing**

🔍 Example Use Cases

Task	Layers
MNIST Digit Classification	Conv2D → Flatten → Dense
House Price Prediction	Dense → Dense
Sentiment Analysis	Embedding → LSTM → Dense

- Sequence models process sequential data, making them ideal for tasks like time series, speech, and text processing.
- Examples of sequence models include RNNs, LSTMs, GRUs, and Transformers, each designed for specific challenges.
- Sequence models capture dependencies between elements, enabling predictions based on context.
- Applications range from language translation and text generation to stock price prediction and speech recognition.

RNN (Recurrent Neural Network):

Issue # 1: Regular Neural Network requires a fixed size input whereas sequences vary in length.

Issue # 2: Regular Neural Networks do not consider order of elements in a sequence.

Issue # 3: No parameter sharing

An RNN (Recurrent Neural Network) is a type of neural network designed to handle sequential data, where the order of the data matters—like time series, text, or audio.

🔁 **What Makes RNNs Special?**

Unlike traditional neural networks, RNNs have loops in them, allowing information to persist. This means they can use their internal memory to process sequences of inputs.

🧠 **How RNNs Work**

At each time step *t*, the RNN takes:

- The current input *xt*
- The previous hidden state *ht*−1

And computes:

- A new hidden state *ht*, which is passed to the next time step
- Optionally, an output *yt*

Mathematically:

$$\boldsymbol{ht = tanh\,(W_{hh}h_{t-1} + W_{xh}x_t + b)}$$

🔁 **RNN Unrolled**

An RNN can be "unrolled" over time to visualize how it processes a sequence:

$$x_1 \rightarrow [RNN] \rightarrow h_1 \rightarrow$$

$$x_2 \rightarrow [RNN] \rightarrow h_2 \rightarrow$$

$$x_3 \rightarrow [RNN] \rightarrow h_3 \rightarrow \ldots$$

Each [RNN] block shares the same weights.

⚠️ **Limitations of Vanilla RNNs**

- Vanishing gradients: Hard to learn long-term dependencies.
- Exploding gradients: Can cause unstable training.

🧬 Variants of RNNs

Model	Description
LSTM	Long Short-Term Memory – handles long-term dependencies using gates
GRU	Gated Recurrent Unit – simpler than LSTM, similar performance
Bidirectional RNN	Processes input in both forward and backward directions
Stacked RNN	Multiple RNN layers stacked for deeper learning

📦 **Applications**

- Text generation
- Machine translation
- Speech recognition
- Time series forecasting
- Music composition

Vocabulary:

Representation of words into numbers

One hot Encoding Representation

Dense Semantic representation

Some kind of semantic relationship between two words

RNN process the words in sequence

EX: That sounds Great

For that it have [0.2 1.4, 0.11] dense semantic representation

Feed forward network and hidden state

🧠 **Imagine This Scenario:**

You want to train a model to predict the **next word** in a sentence.

Let's say the sentence is:

"I love deep" → ?

You want the model to predict:

"learning"

🔁 **How RNN Handles This**

1. **Input Sequence**:
   The sentence is broken into words and fed one at a time:
   - Step 1: "I"
   - Step 2: "love"
   - Step 3: "deep"
2. **Hidden State (Memory)**:
   The RNN has a **hidden state** that acts like a memory is a vector(has previous memory at first 0).
   At each step, it updates this memory based on:
   - The **current word**
   - The **previous memory**
3. **Prediction**:
   After seeing "I", "love", and "deep", the RNN uses its memory to predict the **next word**:

"learning"

🔄 **What's Happening Internally?**

At each time step *t*:

- Input: x_t (e.g., "love")
- Hidden state: h_{t-1} (memory from previous step)
- Output: y_t (e.g., prediction)

The RNN updates its memory:

$$h_t = tanh(W_{xh} \cdot x_t + W_{hh} \cdot h_{t-1} + b)$$

Then it predicts the next word:

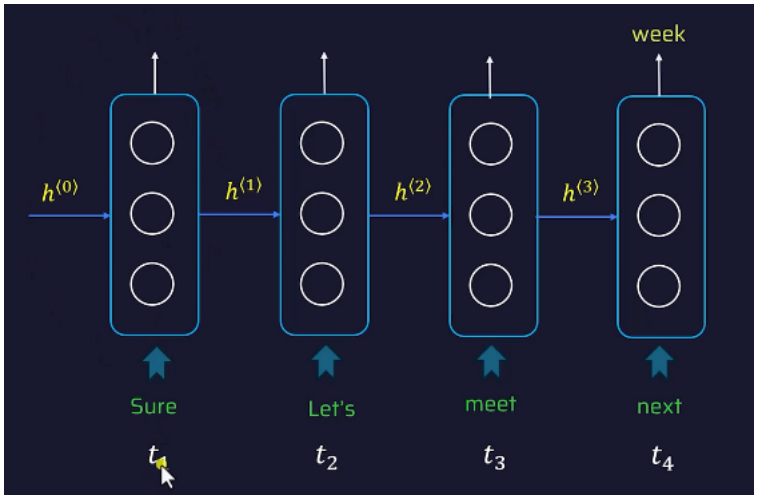
$$y_t = softmax(W_{hy} \cdot h_t + b_y)$$

🔹 Terms Explained:

Term	Meaning
x_t	The input vector at time step t (e.g., the word embedding for the current word)
h_{t-1}	The hidden state from the previous time step (memory from the past)
W_{xh}	The weight matrix that connects the input x_t to the hidden state
w_{hh}	The weight matrix that connects the previous hidden h_{t-1} to the current hidden state
b	The bias vector added to the transformation
tanh	The activation function (hyperbolic tangent), which squashes the output between -1 and 1 to introduce non-linearity

🧠 **Simple Analogy**

Think of the RNN like a person reading a sentence **one word at a time**, remembering what they've read so far, and then **guessing the next word** based on that memory.



Types of RNN:

Named Entity Recognition(NER) This is the type of Many to Many RNN

A **Many-to-Many RNN** is a type of Recurrent Neural Network architecture where:

- The model **takes a sequence of inputs**.

- And **produces a sequence of outputs**, where the number of outputs can be equal to or different from the number of inputs.

Language Translation:

Sentiment Analysis:

Many to one RNN

One to Many → gives one input image then get the description of that image

Vanishing Gradient Problem

The **Vanishing Gradient Problem** is a common issue in training deep neural networks, especially **Recurrent Neural Networks (RNNs)** and very deep feedforward networks.

🧠 **What Is It?**

During backpropagation, gradients (used to update weights) are **multiplied repeatedly** through each layer. If these gradients are **small numbers**, they can **shrink exponentially** as they move backward through the network.

As a result:

- **Early layers** (closer to the input) receive **very small gradients**.
- These layers **learn very slowly** or **not at all**.
- The network **fails to capture long-term dependencies**.

🔬 **Why Does It Happen?**

In RNNs, the hidden state is updated like this:

$$h_t = tanh(W_{xh} \cdot x_t + W_{hh} \cdot h_{t-1} + b)$$

When computing gradients during backpropagation through time (BPTT), the chain rule causes repeated multiplication of derivatives like:

$$\frac{\partial h_t}{\partial h_{t-1}} \cdot \frac{\partial h_{t-1}}{\partial h_{t-2}} \ldots\ldots$$

If these derivatives are **< 1**, the product becomes **very small** → gradients vanish.

⚠️ **Consequences**

- **Slow or no learning** in early layers or earlier time steps.
- **Poor performance** on tasks requiring long-term memory (e.g., language modeling, translation).

✅ Solutions

Technique	How It Helps
LSTM / GRU	Use gates to control information flow and preserve gradients

Technique	How It Helps
ReLU activation	Reduces vanishing compared to sigmoid/tanh
Gradient clipping	Prevents gradients from becoming too small or too large
Batch normalization	Stabilizes training and helps maintain gradient flow
Residual connections	Allow gradients to flow directly across layers

While calculating the gradient during the backward propagation while applying chain rule the gradient will become 0 as it is too small.

The **exploding gradient** problem in fully connected neural networks occurs when gradients grow uncontrollably during backpropagation, causing unstable training and large weight updates.

- Vanishing gradients occur when gradients become too small during backpropagation, hindering effective weight updates.
- It primarily affects deep networks with activation functions like sigmoid or tanh, leading to slow or stalled learning.
- Layers closer to the input experience smaller gradients, causing them to learn much slower than deeper layers.
- Solutions include using activation functions like ReLU, batch normalization, or architectures like LSTMs with gating mechanisms.
- Addressing vanishing gradients is critical for training deep neural networks effectively and efficiently.

LSTM (Long Short-Term Memory)

An LSTM (Long Short-Term Memory) is a special type of Recurrent Neural Network (RNN) designed to remember information for long periods and solve the vanishing gradient problem that standard RNNs struggle with.

🧠 **Why LSTM?**

Standard RNNs forget long-term dependencies because gradients shrink during backpropagation. LSTMs solve this by introducing a memory cell and gates that control the flow of information.

🔁 **LSTM Cell Structure**

An LSTM cell has three gates and a cell state:

1. **Forget Gate** f_t:
   Decides **what to forget** from the previous cell state.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

1. **Input Gate i_t**:
   Decides **what new information to store**.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

2. **Candidate Values $\tilde{c}_t$**:
   New candidate values to add to the cell state.

$$\tilde{c}_t = tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$

3. **Update Cell State c_t**:
   Combine old and new information.

$$C_t = f_t . C_{t-1} \ + i_t . \tilde{c}_t$$

4. **Output Gate o_t**:
   Decides **what to output**.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

5. **Hidden State h_t**:
   Final output of the LSTM cell.

$$h_t = o_t . \tanh(C_t)$$

🧱 Summary of Components

Component	Role
Cell State C_t	Long-term memory
Hidden State h_t	Short-term output
Forget Gate	Removes unneeded info
Input Gate	Adds new info
Output Gate	Controls what to output

📦 Applications

- Language modeling
- Machine translation
- Speech recognition
- Time series forecasting

📚 Predicting the next word in a sentence

🧠 Imagine this sentence:

"**I love deep ___**"

You want the model to predict the next word: "**learning**".

🔄 How LSTM Helps

Think of an LSTM like a **smart reader** with a **notebook** (memory) and **filters** (gates) that help it decide:

- What to **remember**
- What to **forget**
- What to **write down**
- What to **say next**

🧱 Step-by-Step Example

1. **Input 1: "I"**
    - LSTM reads "I"
    - It writes "I" into its memory (cell state)
    - It doesn't say anything yet
2. **Input 2: "love"**
    - LSTM reads "love"
    - It remembers that "I love" is a strong phrase
    - Updates its memory to include this
3. **Input 3: "deep"**
    - LSTM reads "deep"
    - It thinks: "Hmm, people often say 'deep learning'"
    - It keeps "deep" in memory and prepares to predict
4. **Prediction:**
    - Based on memory ("I love deep"), it predicts the next word: **"learning"**

🧠 What's Happening Inside?

At each step, the LSTM uses **gates**:

Gate	What it does
Forget Gate	Decides what old info to forget
Input Gate	Decides what new info to add
Output Gate	Decides what to show as output

So, the LSTM **remembers useful words** like "love" and "deep" and **forgets unimportant ones** if needed.

GRU (Gated Recurrent Unit)

A **GRU (Gated Recurrent Unit)** is a type of Recurrent Neural Network (RNN) that is similar to an **LSTM**, but **simpler and faster**. It was introduced to solve the **vanishing gradient problem** and to help RNNs remember long-term dependencies in sequences.

🧠 What Makes GRU Special?

GRUs use **gates** to control the flow of information, just like LSTMs, but they combine some of the gates to make the architecture more efficient.

🔁 GRU Components

1. **Update Gate** *zt*:
   Decides how much of the **past information** to keep.

$$zt = \sigma(Wz \cdot [ht - 1, xt])$$

2. **Reset Gate** *rt*:
   Decides how much of the **past information to forget**.

$$rt = \sigma(Wr \cdot [ht - 1, xt])$$

3. **Candidate Hidden State** $\tilde{h}_t$:
   New memory content.

$$\tilde{h}_t = tanh(W \cdot [rt \cdot ht - 1, xt])$$

4. **Final Hidden State** *ht*:
   Combines old and new information.

$$ht = (1 - zt) \cdot ht - 1 + zt \cdot \tilde{h}_t$$

🧱 Summary of GRU vs LSTM

Feature	GRU	LSTM
Gates	2 (Update, Reset)	3 (Input, Forget, Output)

Feature	GRU	LSTM
Memory Cell	No	Yes
Speed	Faster	Slower
Performance	Similar	Similar

📦 **Use Cases**

- Text generation
- Sentiment analysis
- Time series forecasting
- Speech recognition

🧠 **Simple Analogy**

Imagine GRU as a **smart filter**:

- It decides what to **keep** from the past.
- What to **forget**.
- And what to **add** from the current input.

This helps it **remember important things** and **ignore noise**, making it great for tasks like predicting the next word in a sentence.

Transformer:

Introduction to Transformer Architecture:

Chat GPT is powered by a model called GPT which is based on a deep learning architecture called Transformers

Language Model

Large Language Model

Its basically a predict a next word in the sentence

The **Transformer architecture** is a deep learning model introduced in the paper **"Attention is All You Need"** by Vaswani et al. in 2017. It revolutionized natural language processing (NLP) and is the foundation of models like **BERT**, **GPT**, and **T5**.

🧠 **Core Idea**

Unlike RNNs or LSTMs, Transformers **do not process data sequentially**. Instead, they use a mechanism called **self-attention** to process the entire input **in parallel**, making them faster and better at capturing long-range dependencies.

🧱 Transformer Architecture Overview

The Transformer consists of two main parts:

🔹 1. Encoder

- Takes the input sequence (e.g., a sentence).
- Outputs a representation of the input.

Each encoder block contains:

- **Multi-head self-attention**
- **Feed-forward neural network**
- **Layer normalization & residual connections**

🔹 2. Decoder

- Takes the encoder's output and generates the target sequence (e.g., translated sentence).
- Each decoder block has:
    - **Masked multi-head self-attention**
    - **Encoder-decoder attention**
    - **Feed-forward network**
    - **Layer normalization & residual connections**

🔍 Key Components

Component	Description
Self-Attention	Allows the model to focus on different parts of the input when encoding a word
Multi-Head Attention	Runs multiple attention layers in parallel to capture different relationships
Positional Encoding	Adds information about the position of words (since there's no recurrence)
Feed-Forward Network	Applies transformations to each position independently
Residual Connections	Helps gradients flow and stabilizes training

Component	Description
Layer Normalization	Normalizes inputs to each sub-layer

📦 **Applications**

- Machine translation (e.g., English → French)
- Text summarization
- Question answering
- Code generation
- Image captioning (with vision transformers)

Word Embeddings:

ML models do not understand the words they only understand the numbers

Word Embedding is a technique in Natural Language Processing (NLP) where words are represented as dense vectors of real numbers. These vectors capture the semantic meaning of words, so that words with similar meanings have similar vector representations.

🧠 **Why Use Word Embeddings?**

Traditional methods like one-hot encoding:

- **Represent words as sparse vectors (mostly zeros)**
- **Don't capture meaning or relationships between words**

Word embeddings solve this by placing similar words closer together in a high-dimensional space.

🔍 **Example**

Word	Vector (simplified)
king	**[0.8, 0.2, 0.6]**
queen	**[0.79, 0.21, 0.59]**
apple	**[0.1, 0.9, 0.3]**

Here, "king" and "queen" are close in vector space, while "apple" is far away.

🧱 **Popular Word Embedding Techniques**

Method	Description
Word2Vec	**Learns embeddings using context (CBOW or Skip-gram)**
GloVe	**Uses word co-occurrence statistics**
FastText	**Considers subword information (good for rare words)**
ELMo / BERT	**Contextual embeddings (word meaning changes with context)**

Googles word2vec is a 300 dimensions vector to represent a single word each dimension [features for the word] can be question the vector is represented in a 300 dimension space so based on the direction distance we can see the relationship between the words

🧠 Word Embedding as a Set of Questions

Let's say a word is represented by a 300-dimensional vector:

- Each dimension could be thought of as answering a question like:
    - *"Is this word related to royalty?"*
    - *"Is this word used in sports?"*
    - *"Does this word have a positive sentiment?"*
    - *"Is this word a verb?"*
    - *"Is this word related to technology?"*

The value in each dimension (e.g., 0.8, -0.3) reflects how strongly the word **aligns** with that concept.

🔍 But Here's the Catch

These "questions" are **not explicitly defined**. The model **learns them automatically** from data. So while we can interpret them loosely, we usually don't know exactly what each dimension means.

However, in practice:

- Words like **"king"** and **"queen"** end up close together.
- Words like **"run"** and **"walk"** cluster together.
- Words like **"apple"** (fruit) and **"apple"** (company) may be separated in **contextual embeddings** (like BERT).

Word2Vec and GloVe are two popular static embedding models these embeddings are trained (deep learning) on huge data

- Word embedding is a way to represent a word in a numeric format such that it captures the semantic meaning of that word.
- Word2vec, Glove, etc., are popular techniques to produce static word embeddings.
- With word embeddings, you can do math like King - man + woman = Queen or Russia – Moscow + Delhi = India.

Contextual embeddings :

are a type of word embedding where the **meaning of a word depends on its context** in a sentence. This is a major improvement over traditional embeddings like Word2Vec or GloVe, which assign **one fixed vector per word**, regardless of how it's used.

🧠 Why Context Matters

Take the word **"bank"**:

- In "I went to the **bank** to deposit money," it means a financial institution.
- In "The boat is near the **bank** of the river," it means the side of a river.

Traditional embeddings would give **"bank"** the same vector in both cases.
Contextual embeddings give **different vectors** depending on the sentence.

🔍 How It Works

Contextual embeddings are generated by **deep language models** like:

Model	Description
ELMo	Uses bidirectional LSTMs to generate context-aware embeddings
BERT	Uses Transformers and attention to understand context in both directions
GPT	Uses Transformers with left-to-right context (good for generation)

These models **read the entire sentence** and generate a unique vector for each word **based on its surrounding words**.

📦 Example (BERT)

Sentence:

"The **bass** was too loud."
"He caught a **bass** in the lake."

- BERT will generate **different embeddings** for "bass" in each sentence.
- It understands that one is about **sound**, the other about a **fish**.

🧠 Summary

Feature	Traditional Embeddings	Contextual Embeddings
One vector per word	✅	❌
Context-aware	❌	✅
Handles ambiguity	❌	✅
Based on	Co-occurrence	Deep language models

Take the embedding of the word and context of its adjectives combing them gives the contextual embedding and it will be easy to predict the next word

- **Context matters**: The meaning of a word is influenced by surrounding words (like adjectives).
- **Combining information** from the word and its context helps in **understanding meaning** and **predicting the next word**.

🔍 **What Actually Happens in Contextual Embeddings:**

In models like **BERT**:

- Each word (e.g., "apple") is **not embedded in isolation**.
- Instead, the model looks at the **entire sentence** (e.g., "She ate a red apple") and uses **self-attention** to weigh how much each surrounding word (like "red") contributes to the meaning of "apple".
- The final embedding of "apple" is a **blend of its own identity and its context**.

So yes—**adjectives, verbs, and other nearby words** all influence the final embedding.

🧠 **Example:**

Sentence:

“The **smart** student solved the problem.”

- The embedding for **"student"** will be influenced by **"smart"**, making it different from:

“The **lazy** student...”

📌 **Summary**

Your Idea	Actual Mechanism
Combine word + adjective embeddings	Use self-attention to blend all contextual signals
Helps predict next word	Yes, especially in models like GPT

Overview of Encoder and Decoder:

🧱 1. Encoder

The **encoder** takes an input sequence (like a sentence) and **converts it into a meaningful representation** (called a context or hidden state).

🔹 **What it does:**

- Reads the entire input sequence.
- Captures the meaning and relationships between words.
- Outputs a set of vectors (one for each input token).

🔹 **Example:**

Input: "I love deep learning"
Encoder → Outputs a set of embeddings that represent this sentence.

🧱 2. Decoder

The **decoder** takes the encoder's output and **generates a new sequence**, one token at a time.

🔹 **What it does:**

- Uses the encoder's output (context).
- Generates the output sequence step-by-step.
- Each step depends on the previous output and the context.

🔹 **Example:**

If the task is translation: Input: "I love deep learning"
Output: "J'adore l'apprentissage profond"

🔁 How They Work Together (Encoder-Decoder)

1. **Encoder** reads the input sentence and creates a context.
2. **Decoder** uses that context to generate the output sentence.

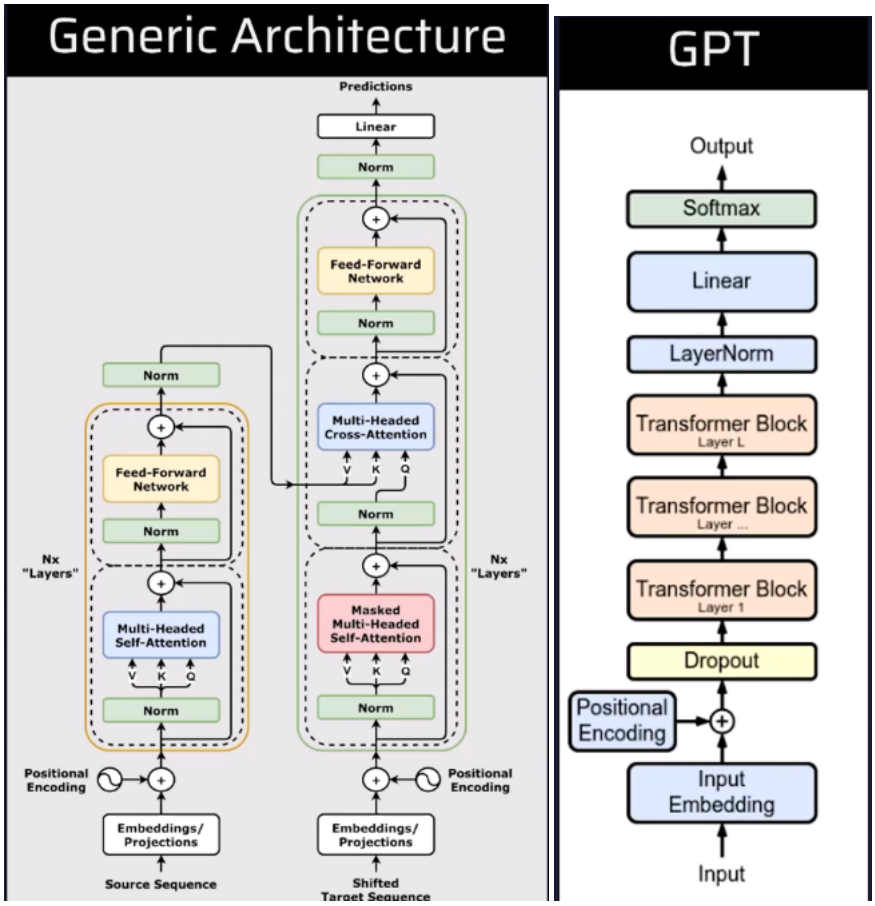
This is the core of **sequence-to-sequence (Seq2Seq)** models.

📦 Applications

Task	Input	Output
Machine Translation	English sentence	French sentence
Text Summarization	Long article	Short summary
Chatbots	User message	Bot reply
Image Captioning	Image features (as input)	Text description

Training and Inference Models

The model that is working on the real-world problem then it is called Inference



GPT will have only the encoder BERT have only the Decoder

🧩 1. Tokenization

Tokenization is the process of breaking text into smaller units (tokens) that a model can understand.

🔹 **Types of Tokens:**

- Words: "I love AI" → ["I", "love", "AI"]

- Subwords: "unhappiness" → ["un", "happi", "ness"]
- Characters: "AI" → ["A", "I"]

🔹 **Why Tokenize?**

- Computers don't understand raw text.
- Tokenization converts text into numerical IDs using a vocabulary.

📍 **2. Positional Embeddings**

Transformers don't have recurrence (like RNNs), so they need a way to understand the order of tokens. That's where positional embeddings come in.

🔹 **What They Do:**

- Add information about position to each token embedding.
- Help the model know that "Transformers are amazing" is different from "Amazing are Transformers".

🔹 **How It Works:**

- Each token gets a position vector (e.g., position 0, 1, 2...).
- This is added to the token's word embedding.

🔹 **Types:**

- Learned: The model learns position vectors during training.
- Sinusoidal: Uses fixed sine/cosine functions (used in the original Transformer paper).

🧠 **Summary**

Concept	Purpose	Example
Tokenization	**Breaks text into model-friendly units**	**"I love AI" → ["I", "love", "AI"]**
Positional Embedding	**Adds order info to tokens**	**Position 0 → [0.1, 0.2, ...]**

Each word in the sentence will have its own embedded vector and is multiplied by the postitional embedding vector which have the knowledge of the words postions because transformers works in the sequence model

Attention Mechanism:

The Attention Mechanism is a key innovation in deep learning, especially in Transformer models, that allows the model to focus on the most relevant parts of the input when making predictions.

🧠 Why Attention?

In tasks like translation or summarization, not all words in a sentence are equally important. Attention helps the model weigh the importance of each word in the input when generating each word in the output.

[words needs attention]

🔍 How It Works (Simplified)

Let's say we're translating the sentence:

"The cat sat on the mat."

When generating the word "s'est" in French, the model should pay more attention to "sat" than to "cat" or "mat".

🧮 Core Formula (Scaled Dot-Product Attention)

Given:

- Query (Q): What we're looking for[book in a library]
- Key (K): What we have [book rack and book description,index]
- Value (V): What we return [book content is the value]

The attention is computed as:

$$\boldsymbol{Attention}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}) = \boldsymbol{softmax}(\frac{QK^T}{\sqrt{d_k}})\boldsymbol{V}$$

- QK^T: Measures similarity between query and keys T is Transpose
- d_k: Scaling factor dimesions of the key vector (eg: 128 for GPT)
- Softmax: Turns scores into probabilities
- Multiply by *V*: Weighted sum of values

🧠 Intuition

- The model asks a question (Query)
- It compares that question to all words in the input (Keys)
- It uses the answers (Values) to generate the output

🔁 Types of Attention

Type	Description
Self-Attention	**Each word attends to all other words in the same sentence**
Cross-Attention	**Decoder attends to encoder outputs (used in translation)**

Type	Description
Multi-Head Attention	**Runs multiple attention layers in parallel to capture different relationships**

📦 Applications

- Machine translation
- Text summarization
- Question answering
- Image captioning
- Speech recognition

How much adjectives are attending to the main words (attention score) all the words will have an attention score with each other

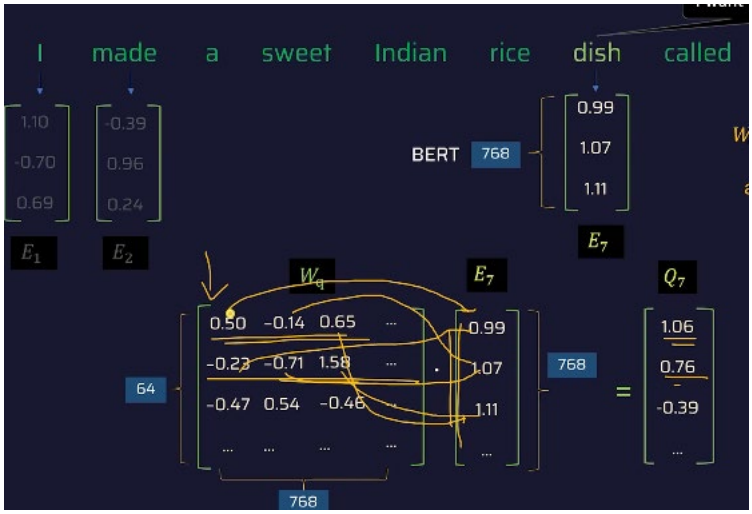
Query on the main word(attention word) in a sentence and remaining words will have the answers(like what that word(modifiers) represents for the attention word) called **key**

The component of the modifiers what value do they add to the attention word called **value**

Each value is a static embedding, and you will get the context aware embedding

Let's walk through a **simple example** of how **Q (Query), K (Key), and V (Value)** work in

Each vector it then calculated with $W_q \, . E_7 \; vector \; with \; dimensions \; as \; 64X768$ berth model this will build a query vector same for the Wk also

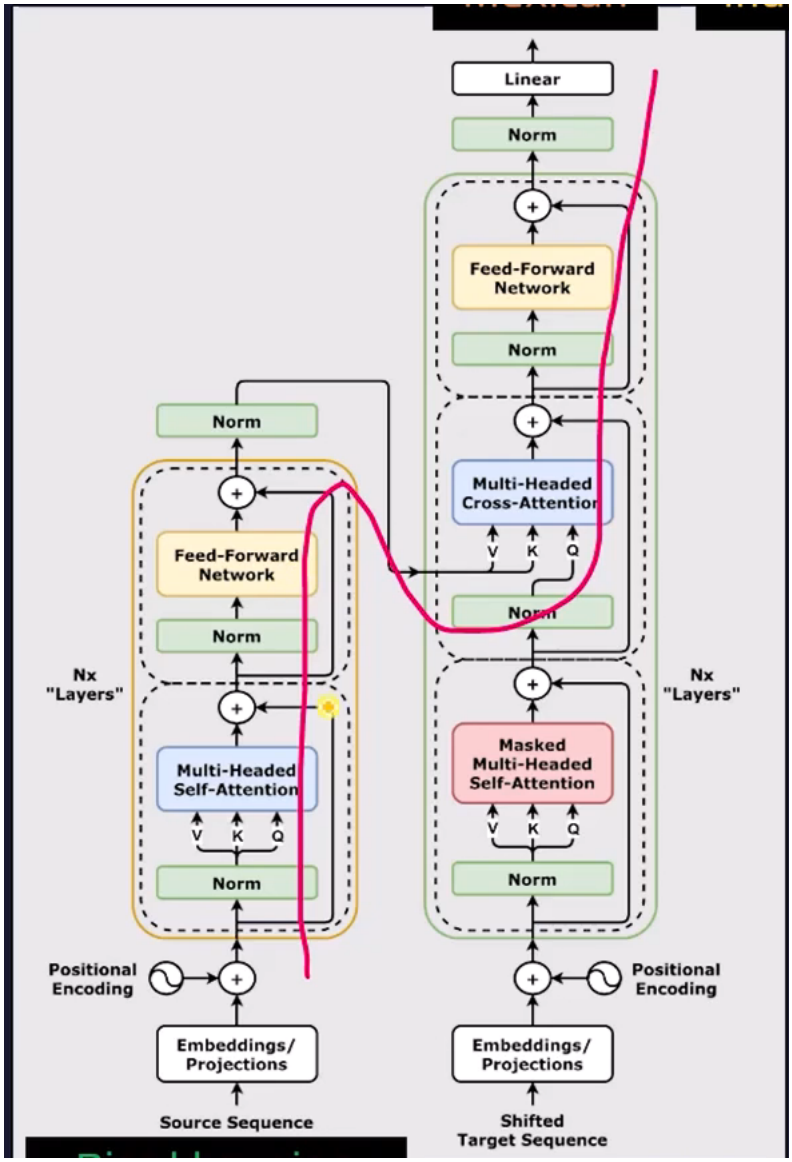


Wq knows how to encode query of a token for attention computation will have the query vector

Wk know how to encode key of a token for attention computation will have the key vector

Wv know how to encode values of a token for attention computation will have the value vector

WHEN THE MODEL IS TRAINED IT WILL HAVE THE **STATIC WORD EMBEDDING MATRIX AND** W_q, W_K, W_v these values will get by the back propagation



📄 **Sentence:**

"The cat sat on the mat."

Let's say we want to compute **self-attention** for the word **"sat"**.

🧠 **Step-by-Step:**

1. Token Embeddings

Each word is first converted into a vector (embedding). For simplicity, let's assume:

Word	Embedding (3D)
the	[1, 0, 1]
cat	[0, 1, 0]

Word	Embedding (3D)
sat	[1, 1, 0]
on	[0, 1, 1]
mat	[1, 0, 0]

2. Linear Projections to Q, K, V

We apply learned weight matrices to get:

- **Query (Q)** for "sat"
- **Keys (K)** for all words
- **Values (V)** for all words

Let's say:

- $Qsat = [0.5,1.0]$
- $Kthe = [1.0,0.0]$
- $Kcat = [0.5,0.5]$
- $Ksat = [0.5,1.0]$
- $Kon = [0.0,1.0]$
- $Kmat = [1.0,1.0]$

3. Compute Attention Scores

We compute dot products between the **query** and each **key**:

- $Score("sat","the") = Q \cdot K = 0.5 \times 1 + 1 \times 0 = 0.5Q \cdot K = 0.5 \times 1 + 1 \times 0 = 0.5$
- $Score("sat","cat") = 0.5 \times 0.5 + 1 \times 0.5 = 0.750.5 \times 0.5 + 1 \times 0.5 = 0.75$
- $Score("sat","sat") = 0.5 \times 0.5 + 1 \times 1 = 1.250.5 \times 0.5 + 1 \times 1 = 1.25$
- $Score("sat","on") = 0.5 \times 0 + 1 \times 1 = 1.00.5 \times 0 + 1 \times 1 = 1.0$
- $Score("sat","mat") = 0.5 \times 1 + 1 \times 1 = 1.50.5 \times 1 + 1 \times 1 = 1.5$

4. Softmax the Scores

Convert scores to probabilities (attention weights):

5. Weighted Sum of Values

Each word has a **Value (V)** vector. Multiply each V by its attention weight and sum them to get the **final output vector** for "sat".

✅ Final Output

The output vector for "sat" is a **blend of all the words**, weighted by how relevant they are to "sat"—this is what makes attention powerful.

Multi Headed Attention:

🧠 What is Multi-Head Attention?

Multi-Head Attention is an extension of the attention mechanism used in Transformer models. Instead of computing a single attention output, it computes multiple attention outputs in parallel, each with its own set of learned weights.

🔍 Why Use Multiple Heads?

Each attention head can focus on different parts of the input or capture different types of relationships. For example:

- One head might focus on syntax (e.g., subject-verb agreement),
- Another on semantics (e.g., word meaning),
- Another on position or long-range dependencies.

🧮 How It Works (Simplified)

1. Input: A sequence of token embeddings.
2. Linear Projections: The input is projected into multiple sets of Q (Query), K (Key), V (Value) using different weight matrices.
3. Parallel Attention: Each head computes attention independently:

$$head_i = Attention(Q_i, K_i, V_i)$$

4. Concatenation: All heads are concatenated:

$$MultiHead(Q, K, V) = Concat(head_1, \dots, head_h)W^0$$

Final Linear Layer: The concatenated output is passed through a final linear layer.

🧱 Summary Table

Component	Role
Multiple QKV sets	**Learn different attention patterns**
Parallel heads	**Capture diverse relationships**
Concatenation	**Combine all insights**

Component	Role
Final projection	**Merge into one output vector**

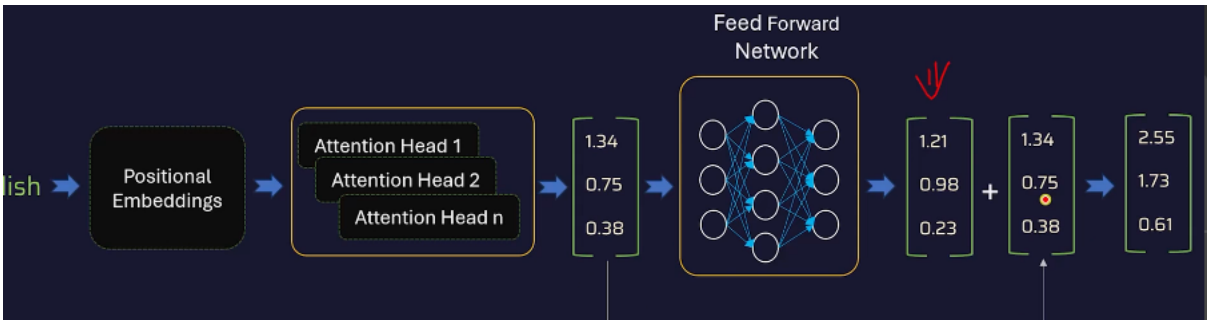
📦 **Example Use Case**

In a translation task:

- One head might align verbs,
- Another might align nouns,
- Another might track word order.

Each attention block works on different aspects like noun, verb etc.

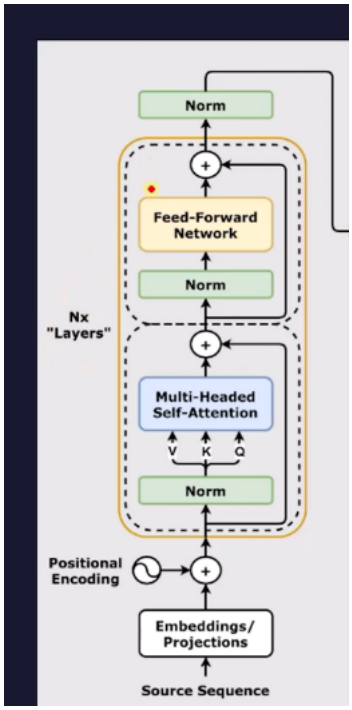
The purpose of multiple attention heads is to allow the model to focus on different aspects or types of relationships between tokens (e.g., semantic, positional, syntactic) simultaneously, enriching the contextual understanding of each token.



The **Feed-Forward Network (FFN)** enriches each token's embedding by applying non-linear transformations independently, enabling the model to capture complex patterns and higher-order features beyond contextual relationships.

Normalization layer ensures stable learning, improving the gradient flow.

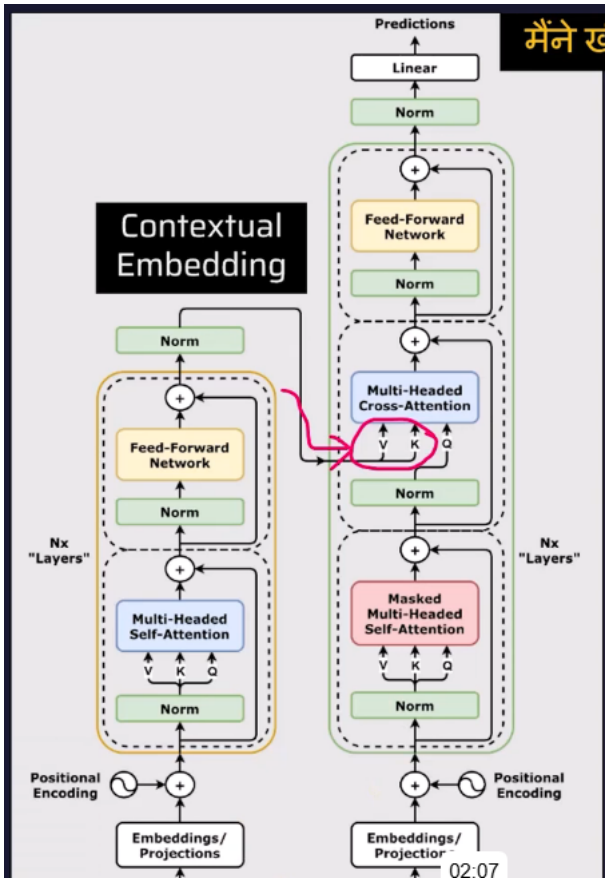
Nx Layers in the BERT Base 12 layers and in BERT large 24 layers



Sentence→embeddings→position embeddings→

Nx layer →

Decoder:



- The decoder in Transformer architecture generates the output sequence step-by-step, one token at a time.
- It uses masked self-attention to ensure predictions depend only on previously generated tokens.

- The decoder integrates encoder outputs through cross-attention to incorporate contextual information from the input sequence.
- Fully connected layers in the decoder refine the processed information for final token prediction.
- The decoder is central to tasks like language translation and text generation, where sequential output is crucial.

🔄 **Training a Transformer: Step-by-Step**

1. Input Preparation

- **Tokenization**: Text is split into tokens (e.g., words or subwords).
- **Embedding**: Tokens are converted into vectors.
- **Positional Encoding**: Added to embeddings to give the model a sense of word order.

2. Encoder Block

- Each token embedding passes through **N layers** of:
    - **Multi-Head Self-Attention**: Each word attends to all others in the input.
    - **Feed-Forward Network**: Applies transformations to each token.
    - **Layer Normalization**: Stabilizes training.

The encoder outputs a **contextual representation** of the input sequence.

3. Decoder Block

- Takes the **target sequence** (e.g., in translation, the sentence in the target language).
- Each token passes through **N layers** of:
    - **Masked Multi-Head Self-Attention**: Prevents looking ahead at future tokens.
    - **Cross-Attention**: Attends to encoder outputs (input context).
    - **Feed-Forward Network** and **Normalization**.

4. Prediction & Loss

- The decoder's output is passed through a **linear layer** to predict the next token.
- The prediction is compared to the actual target using a **loss function** (e.g., cross-entropy).

5. Backpropagation & Optimization

- Gradients are computed and used to update model weights using an optimizer like **Adam**.

6. Repeat

- This process is repeated over many **batches and epochs** until the model learns to generate accurate outputs.

How are Transformers Trained?

The approach they use called self-supervised learning

In Self-supervised learning, labels are generated from the data itself without requiring manual annotations.

The method of predicting next word is called Casual Language Modeling (CLM) GPT (unidirectional) only after the word prediction

Masked Language Modeling (MLM) masks the words randomly (bi directional) BERT

Hugging Face: BERT Basics

Hugging Face is a company and open-source community that builds tools for natural language processing (NLP) and machine learning (ML). It's best known for its Transformers library, which provides easy access to thousands of pretrained models for tasks like:

- Text classification
- Sentiment analysis
- Question answering
- Text generation
- Translation
- Named entity recognition (NER)
- And more

🔧 **Key Features of Hugging Face:**

1. 🤗 **Transformers Library**
   A Python library that provides APIs to use state-of-the-art models like BERT, GPT, T5, RoBERTa, etc.
2. **Model Hub**
   A central place to share and download models. You can find models trained by researchers, companies, or the Hugging Face team itself.
   🌐 https://huggingface.co/models
3. **Datasets Library**
   A collection of ready-to-use datasets for ML and NLP tasks.
4. **Tokenizers Library**
   Fast and efficient tokenization tools, crucial for preparing text data for models.
5. **Inference API**
   Run models in the cloud without needing to set up infrastructure.
6. **Auto Classes**
   Like AutoTokenizer, AutoModel, etc., which automatically load the correct class for a given model name.

🧠 **Example Use Case**

```
from transformers import pipeline
classifier = pipeline("sentiment-analysis")
result = classifier("I love Hugging Face!")
print(result)
```

padding

max_length

truncation