M.Sc. Applied Mathematics

School of Emerging Science & Technology

Gujarat University, Ahmedabad

AMS Sem – 4

Project presentation

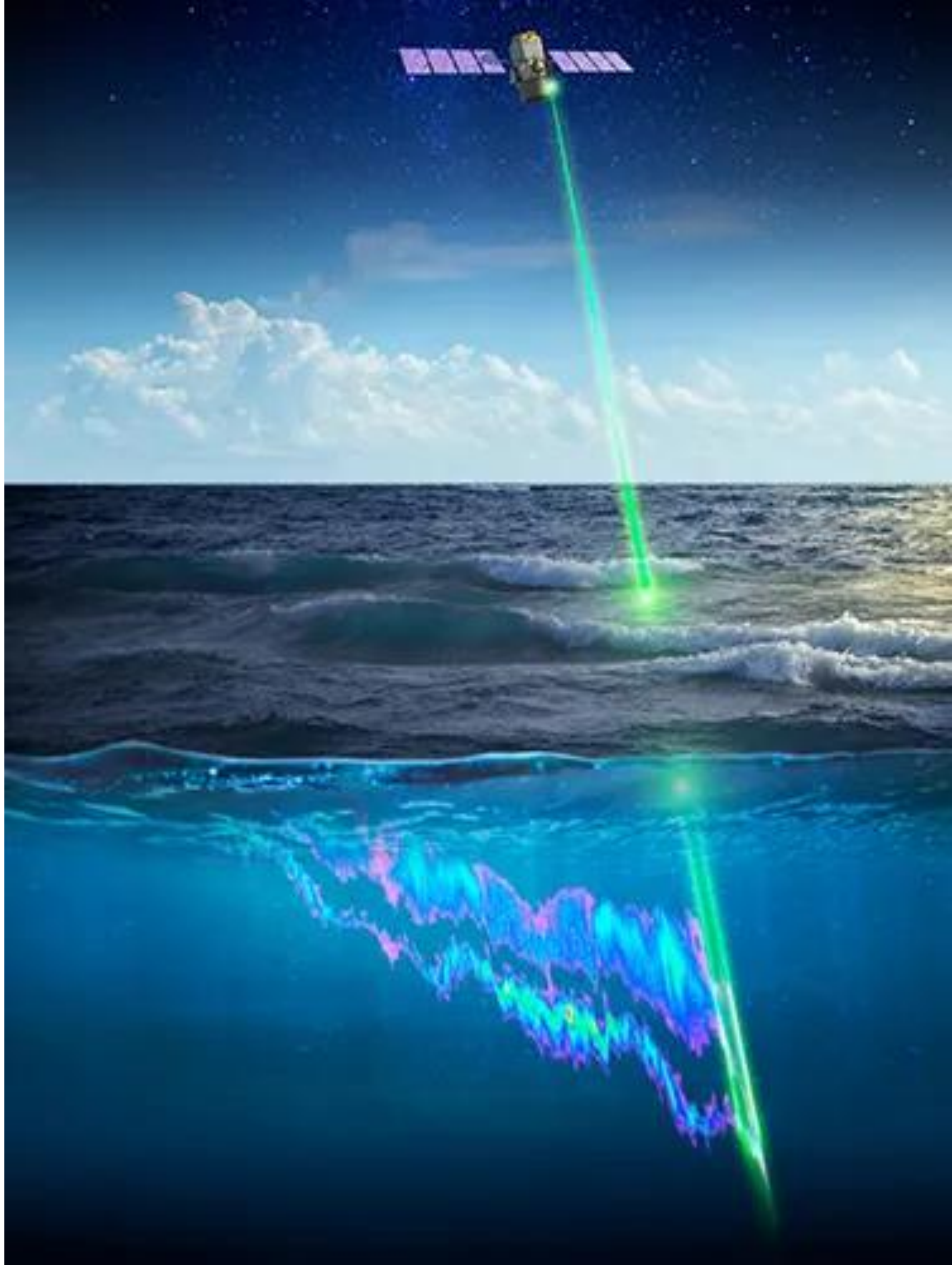
Milansinh Gohil :- 06

Yashkumar Joshi :- 33

Problem statement

1. Prediction of sea surface currents by using Deep neural network and satellite observations of sea surface winds, height and temperatures in the North Indian Ocean.

Mentor : Jaikumar Pundir



Basics about data

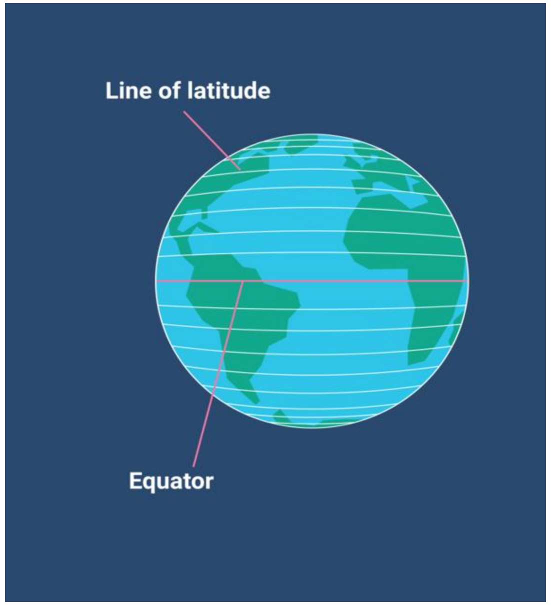
- We have 4 columns in our data Time, Longitude, Latitude, ADT

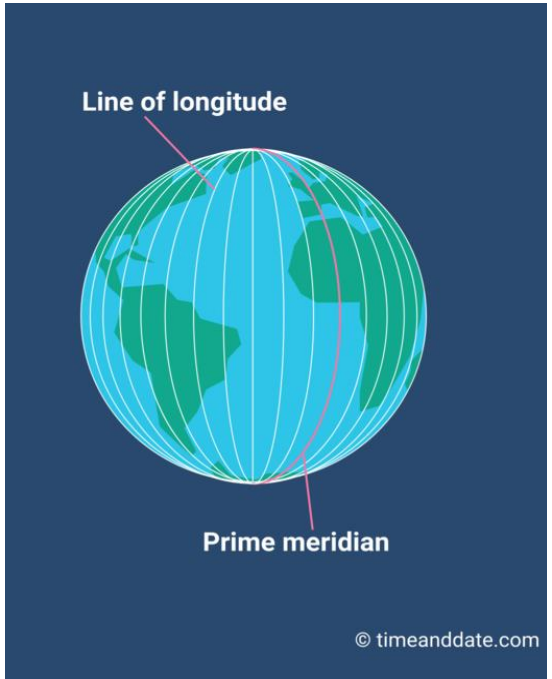
Which are important parameters for our data

- **Latitude**: Lines that run east-west, showing how far north or south a place is from the equator.

- **Longitude**: Lines that run north-south, showing how far east or west a place is from the Prime Meridia
- Together, they create a grid to pinpoint any location on Earth.

- **Time**: Time changes when you cross time zones, but your longitude and latitude remain the same, keeping your exact location on Earth unchanged.

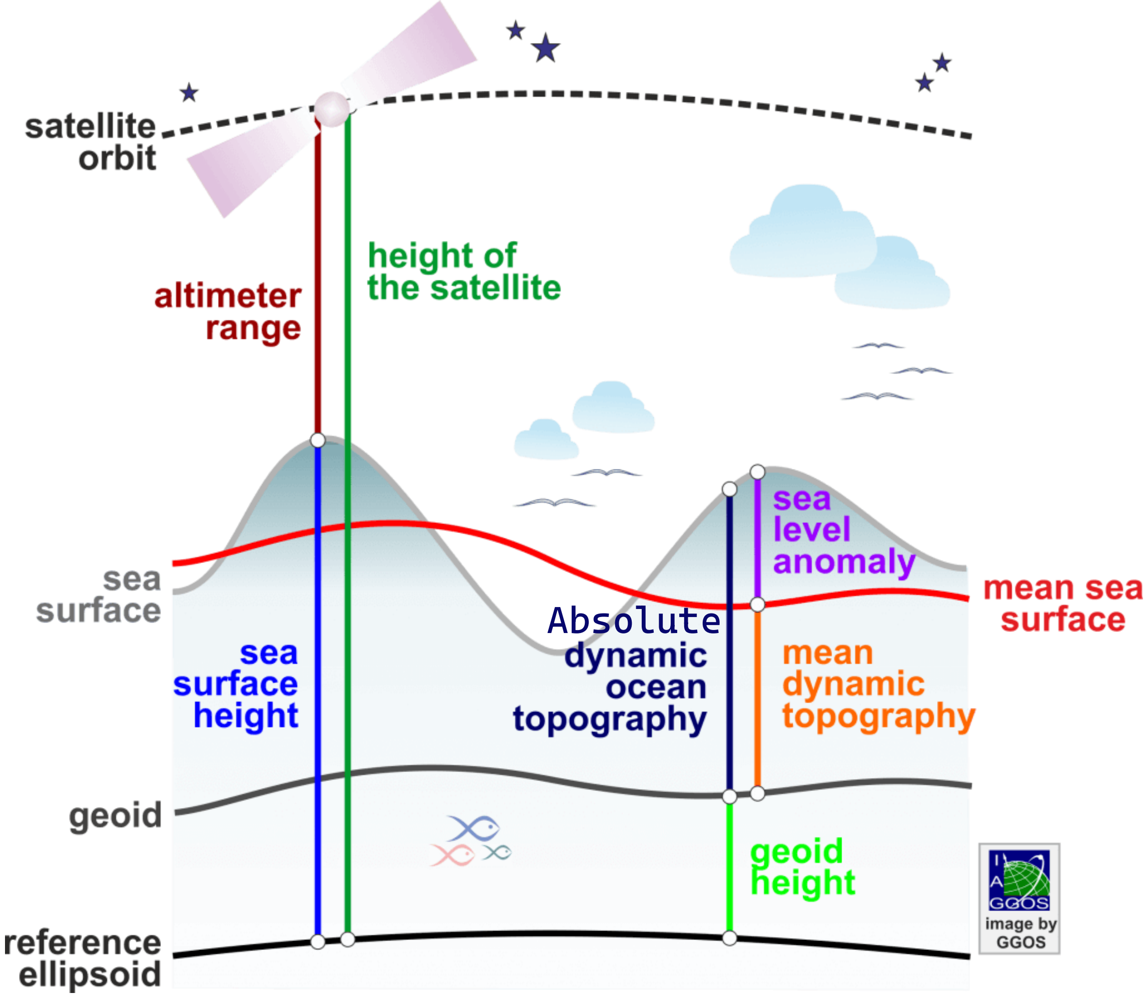




ADT (Absolute Dynamic Topography)

- It measures the height of the sea surface compared to a reference level (geoid), showing ocean currents.
  - Higher ADT means warmer, less dense water
  - lower ADT means cooler, denser water.

- The absolute dynamic topography is the sea surface height above geoid the adt is obtained as adt = sla + mdt



SSH: Sea Surface Height
SLA: Sea Level Anomaly
MSS : Mean Sea Surface
MDT : Mean Dynamic Topography

Why .nc files?

- A .nc file is a NetCDF file, which is a format for storing scientific data. NetCDF stands for Network Common Data Form.
- We have a datafile in .nc format so let's see what do we know about .nc files.

- .nc files can store a variety of scientific data in a structured and efficient way.

1. **Multidimensional data** (e.g., time, depth, latitude, longitude).
2. **Variables** (e.g., temperature, wind speed).
3. **Metadata** (e.g., units, descriptions).
4. **Gridded data** (e.g., maps, satellite images).
5. **Large datasets** efficiently.

Perfect for climate, weather, and ocean data

What is Xarray?

- Xarray is Python library for working with labeled multi-dimensional data, like .nc files.
- Xarray makes it easy to handle complex datasets. (e.g., Time, latitude, longitude)
- To import .nc file as shown in code, This lodes the data into a structured format for easy analysis

Benefits

- Simple to use.
- Great for visualizing and analyzing Ocean, Weather, or Climate data.

In [1]:
```python
import xarray as xr
ds=xr.open_dataset('adt.nc')
adt_data=ds['adt']
```

In [2]:
```python
adt_data
```

Out[2]:

xarray.DataArray 'adt' (**time**: 1254, **latitude**: 120, **longitude**: 240)

[36115200 values with dtype=float64]

▾ Coordinates:

latitude	(latitude)	float32	0.125 0.375 0.625 ... 29.62 29.88
axis :	Y		
bounds :	lat_bnds		
long_name :	Latitude		
standard_name :	latitude		
units :	degrees_north		
valid_max :	29.875		
valid_min :	0.125		
longitude	(longitude)	float32	40.12 40.38 40.62 ... 99.62 99.88
axis :	X		
bounds :	lon_bnds		
long_name :	Longitude		
standard_name :	longitude		
units :	degrees_east		
valid_max :	99.875		
valid_min :	40.125		
time	(time)	datetime64[ns]	2020-01-01 ... 2023-06-07
valid_min :	25567		
valid_max :	26820		

Data Description

Here we have, Columns : 4, Rows : 36,115,200

- As we can see this is Time series data.

- In data as we can see we have time & for that particular time we have different values of latitude , longitude & adt
- For particular 1 day of time we have 120 values of latitude & for 1 value of latitude we have 240 values of longitude .

  so , for one value(day) of time we have $(120 \times 240 = 28800)$ rows. & same amount of adt values
- We have the data of 1254 values(days) of time.

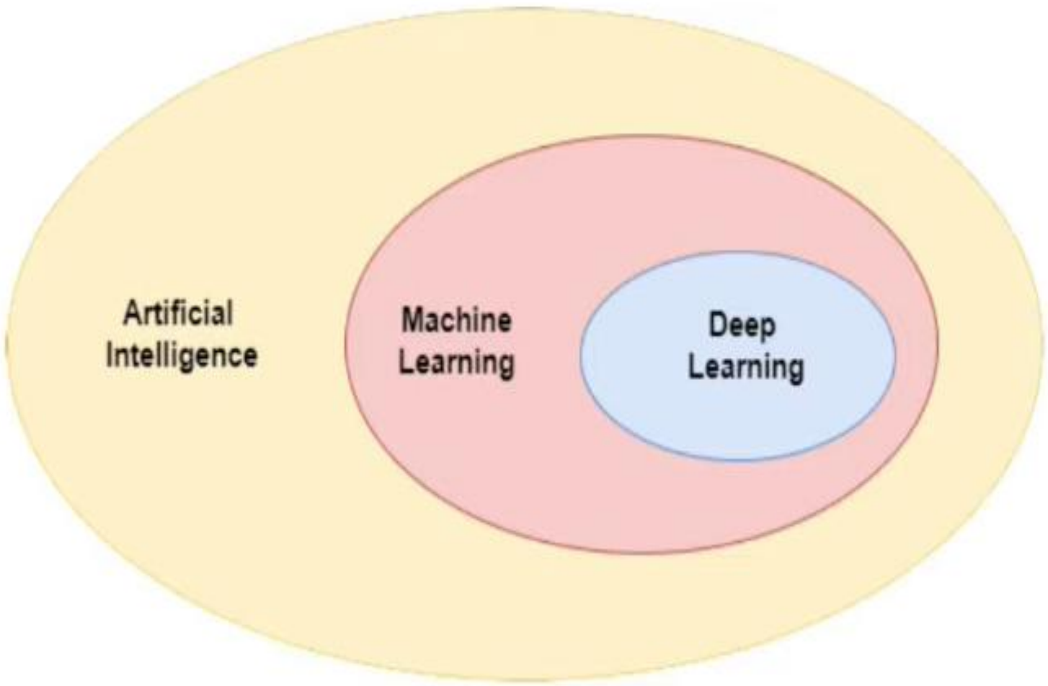
  so, we have total rows ($1254 \times 28800 =$ 36,115,200)

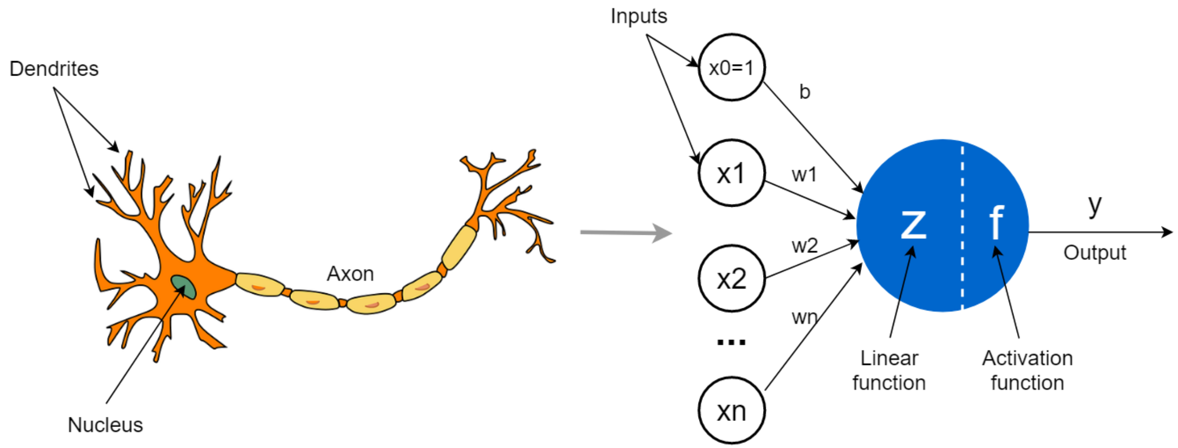
			adt
time	latitude	longitude	
2020-01-01	0.125	40.125	NaN
		40.375	NaN
		40.625	NaN
		40.875	NaN
		41.125	NaN
	...	...	...
	29.875	99.125	NaN
		99.375	NaN
		99.625	NaN
		99.875	NaN
2020-01-02	0.125	40.125	NaN

What is Deep Learning?

- Deep learning is a subfield of Artificial Intelligence and Machine Learning that is inspired by the structure of a human brain.

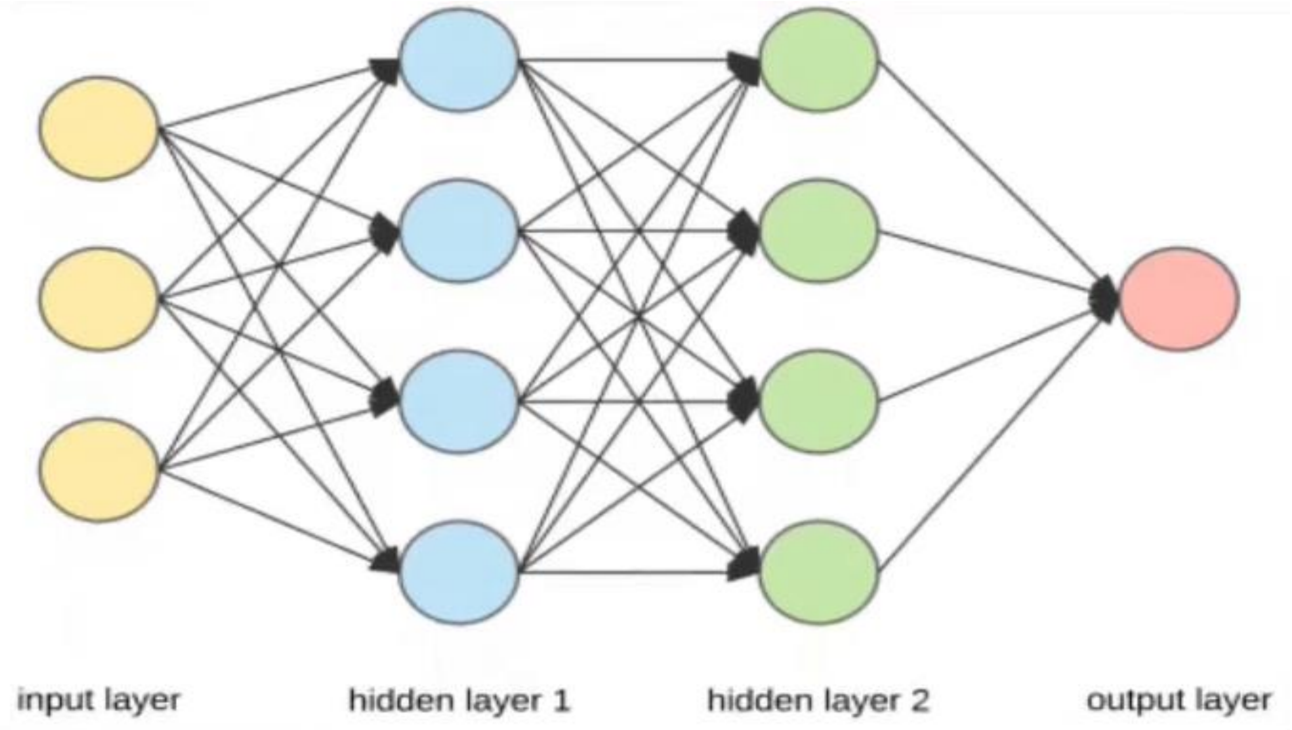
- Deep learning algorithms attempt to draw similar conclusions as humans would by continually analyzing data with a given logical structure called Neural Network.



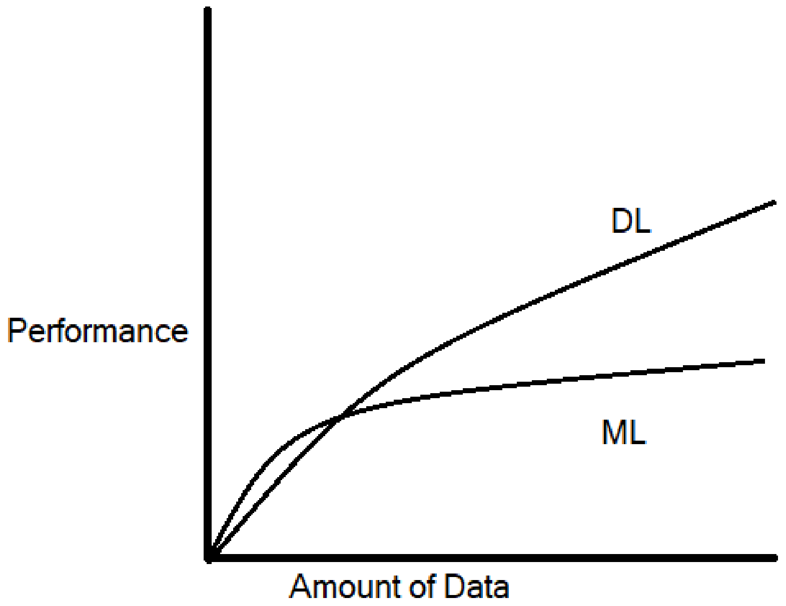


What is Deep Learning?

- Deep learning is part of a broader family of machine learning methods based on artificial neural networks with representation learning.
- Deep learning algorithms uses multiple layers to progressively extract higher-lavel features from the raw input. For example, in image processing, lower layers may identify edges, while higher layers may identify the concepts relevant to a human such as digits or letters or faces.



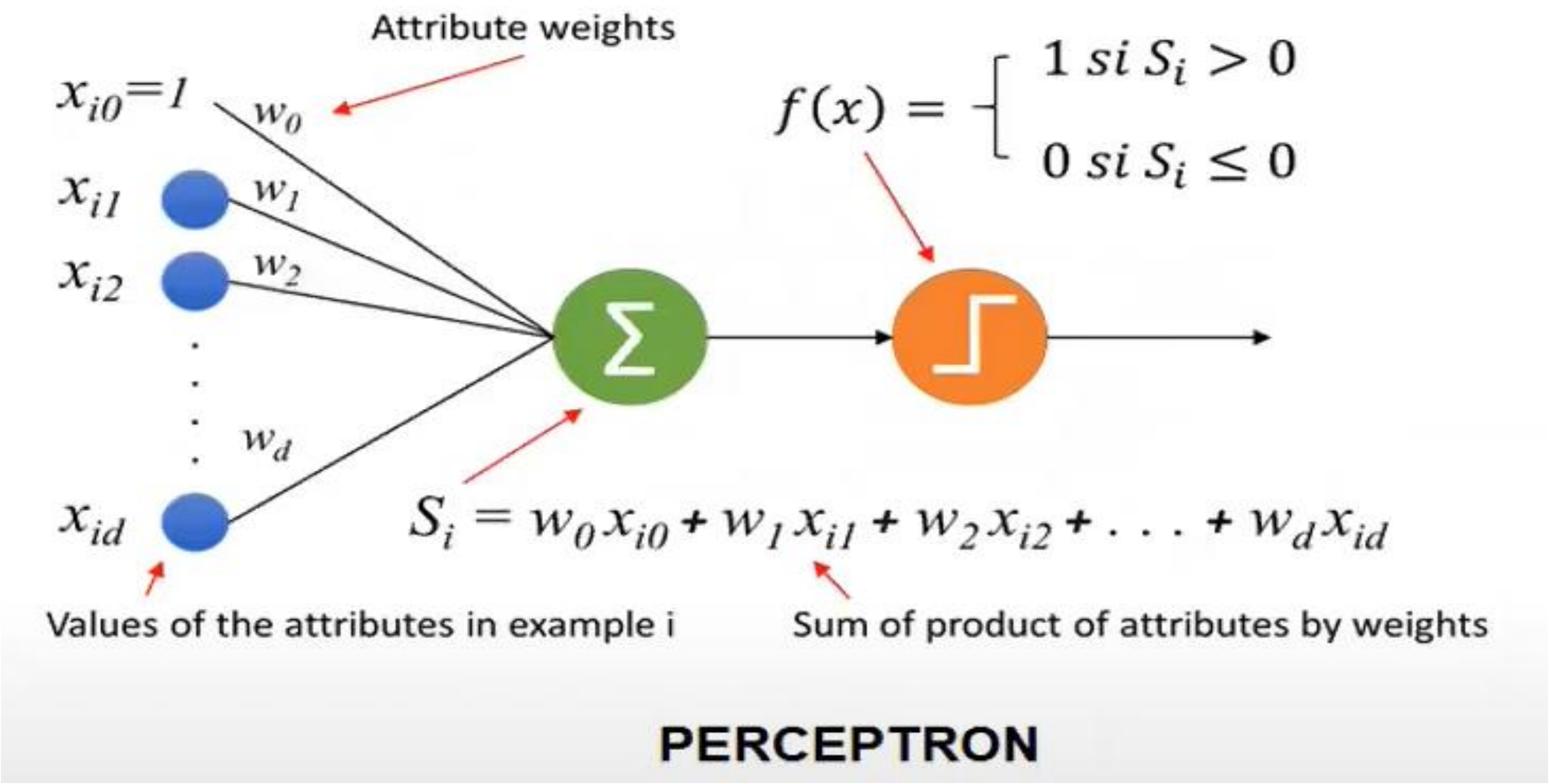
Deep Learning VS Machine Learning



1. Data Dependency
2. Hardware Dependency
3. Training Time
4. Feature Selection
5. Interpretability

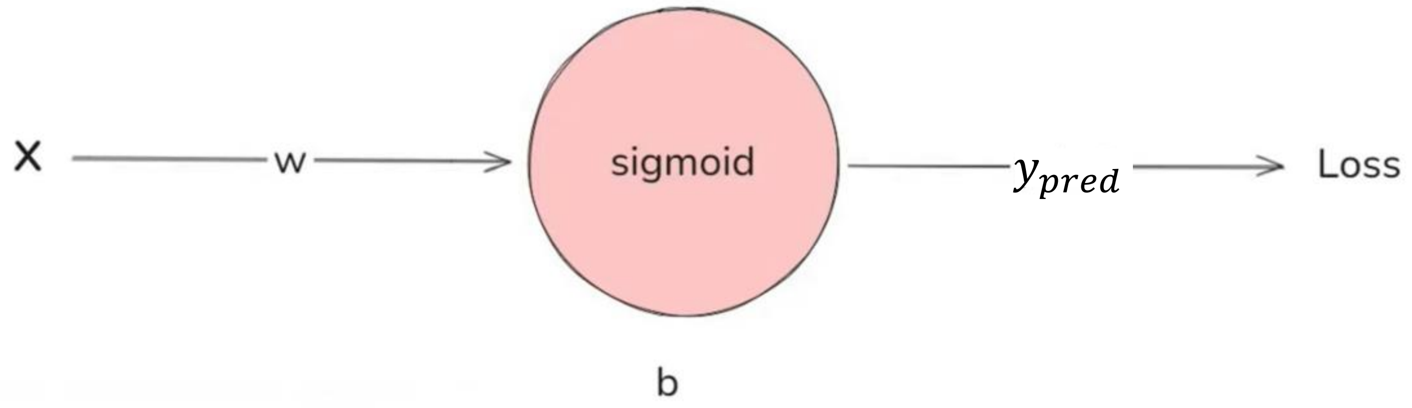
Perceptron

- Perceptron is a mathematical model or we can say it is an algorithm. It works for supervised learning.
- Because of it's design it automatically became a building block of deep learning.



Perceptron Back Propagation

CGPA	Placed
9.11	1
8.9	1
7	0
6.56	1
4.56	0



Training process

1. Forward Pass
2. Calculate Loss
3. Backward Loss
4. Update Gradients

Forward Pass Computation

1. Linear Transformation :

$$z = w.x + b$$

2. Activation(Sigmoid Function):

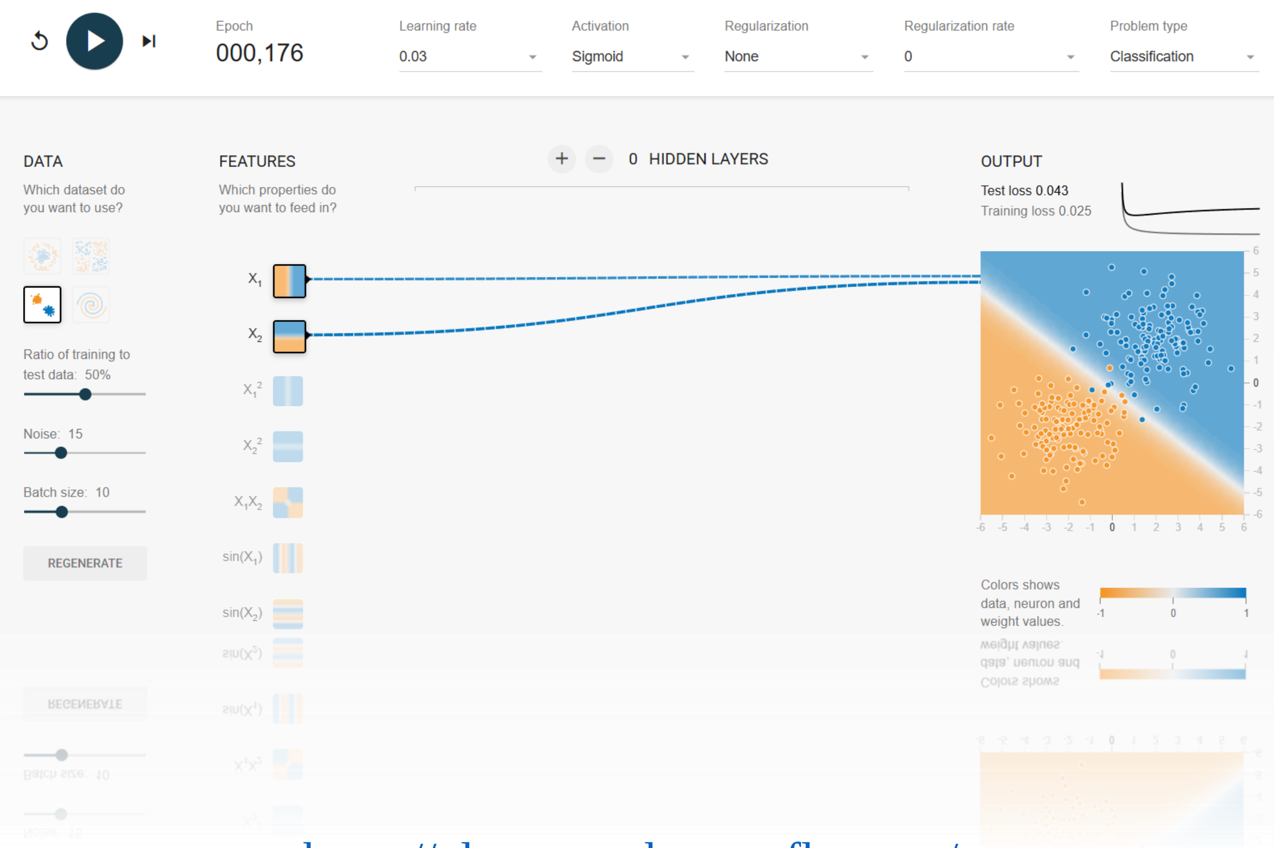
$$y_{pred} = \frac{1}{1+e^{-z}}$$

3. Loss Function (Binary Cross-Entropy Loss):

$$L = -[y_{pred}.\ln(y_{pred}) + (1 - y_{target}).\ln(1 - y_{pred})]$$

Problem with Perceptron

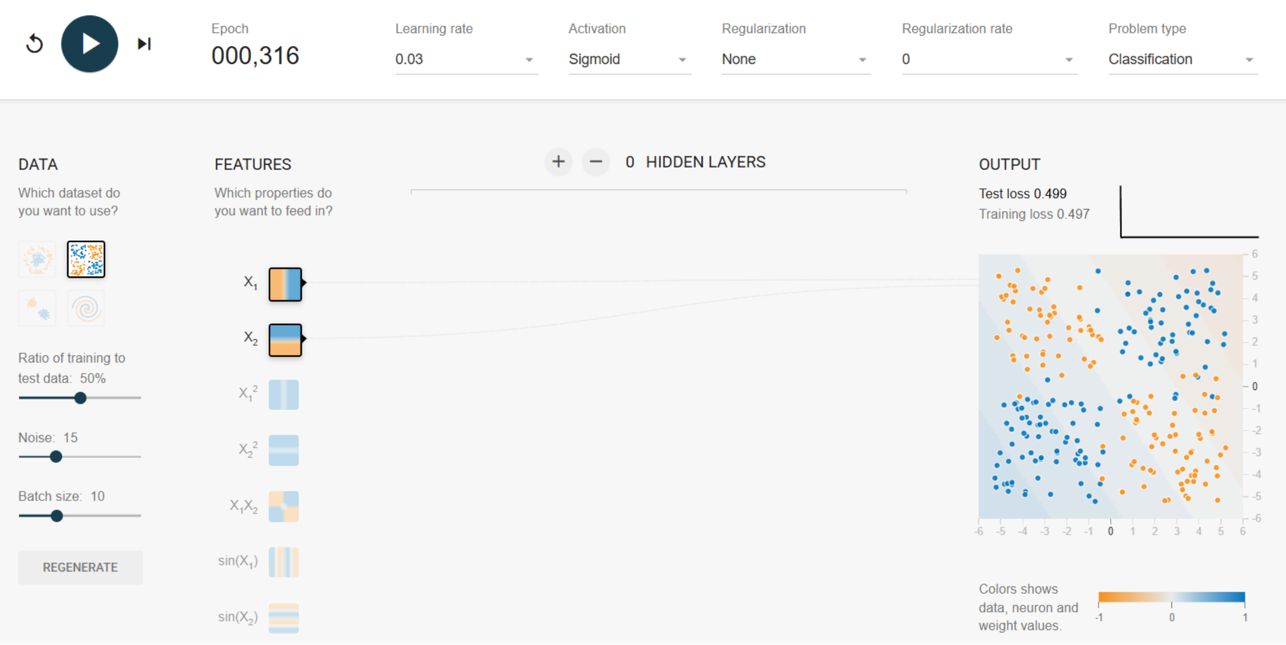
➢ The perceptron performs well with linear data, but struggles significantly with non-linear data, leading to prediction issues.



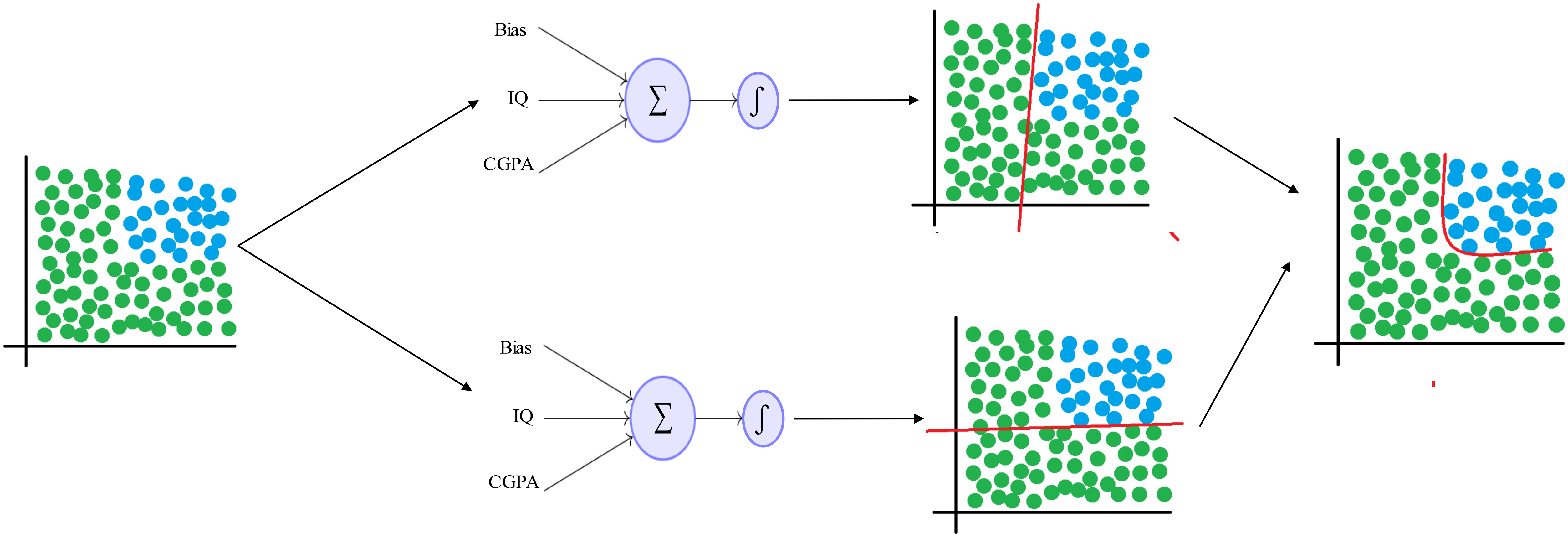
https://playground.tensorflow.org/

➢ **XOR Data:**

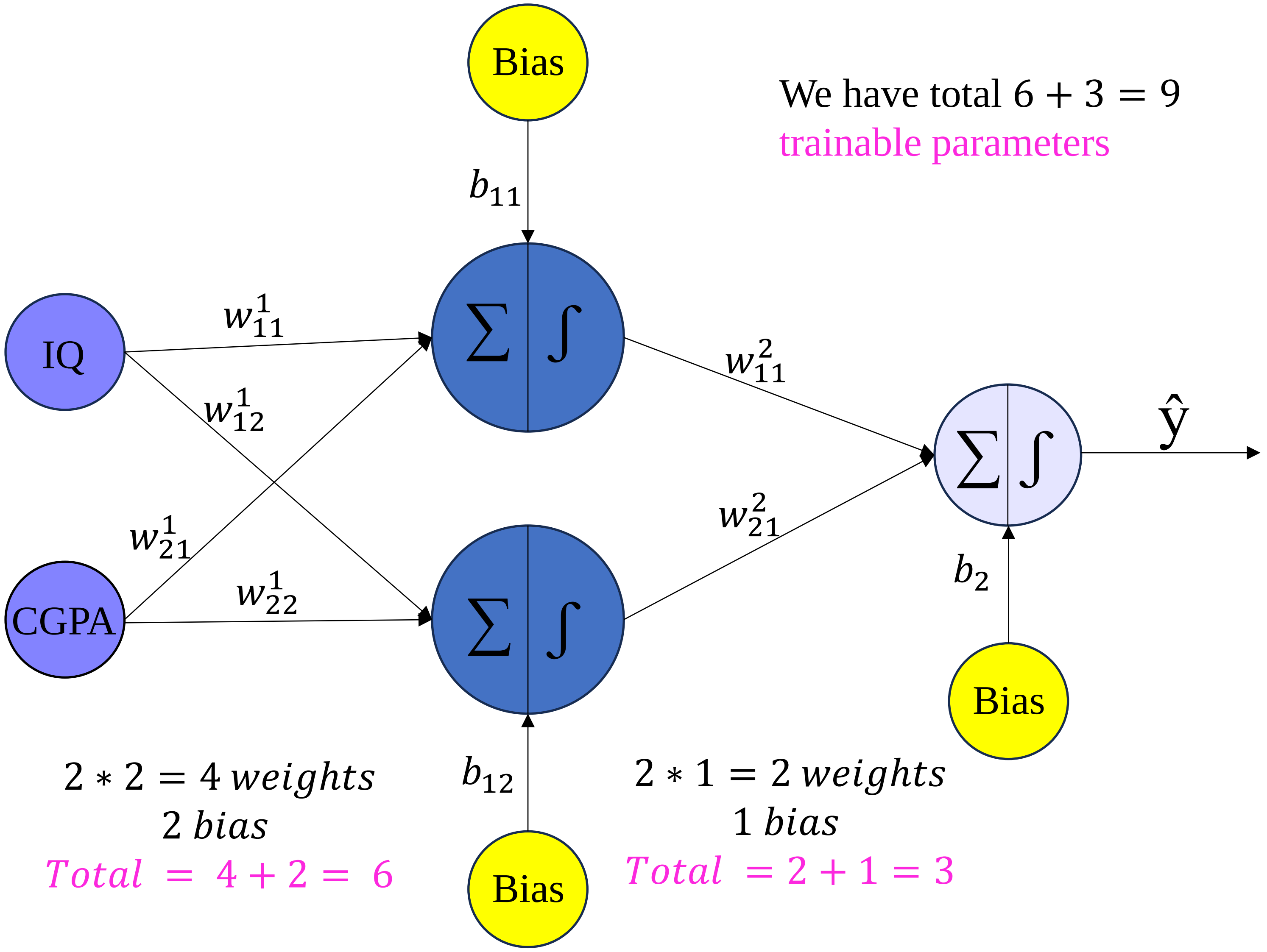
Input A	Input B	Output
0	0	0
0	1	1
1	0	1
1	1	0



ANN (Artificial Neural Network)



ANN Architecture, Forward & Backward Pass



- Forward propagation
- Layer-1

$$a^{[1]} = \begin{bmatrix} w_{11}^1 & w_{12}^1 \\ w_{21}^1 & w_{22}^1 \end{bmatrix}^T \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_{11} \\ b_{12} \end{bmatrix}$$

$$a_1^{[1]} = w_{11}^1 x_1 + w_{21}^1 x_2 + b_{11} \qquad z_1^{[1]} = \sigma\left(a_1^{[1]}\right) = o_{11}$$

$$a_2^{[1]} = w_{12}^1 x_1 + w_{22}^1 x_2 + b_{12} \qquad z_2^{[1]} = \sigma\left(a_2^{[1]}\right) = o_{22}$$

- Layer-2

$$a^{[2]} = \begin{bmatrix} w_{11}^2 \\ w_{21}^2 \end{bmatrix}^T \cdot \begin{bmatrix} o_{11} \\ o_{22} \end{bmatrix} + b_2$$

$$a^{[2]} = w_{11}^2 o_{11} + w_{21}^2 o_{22} + b_2$$

$$z^{[2]} = \sigma\left(a^{[2]}\right) = \hat{y}$$

- Loss

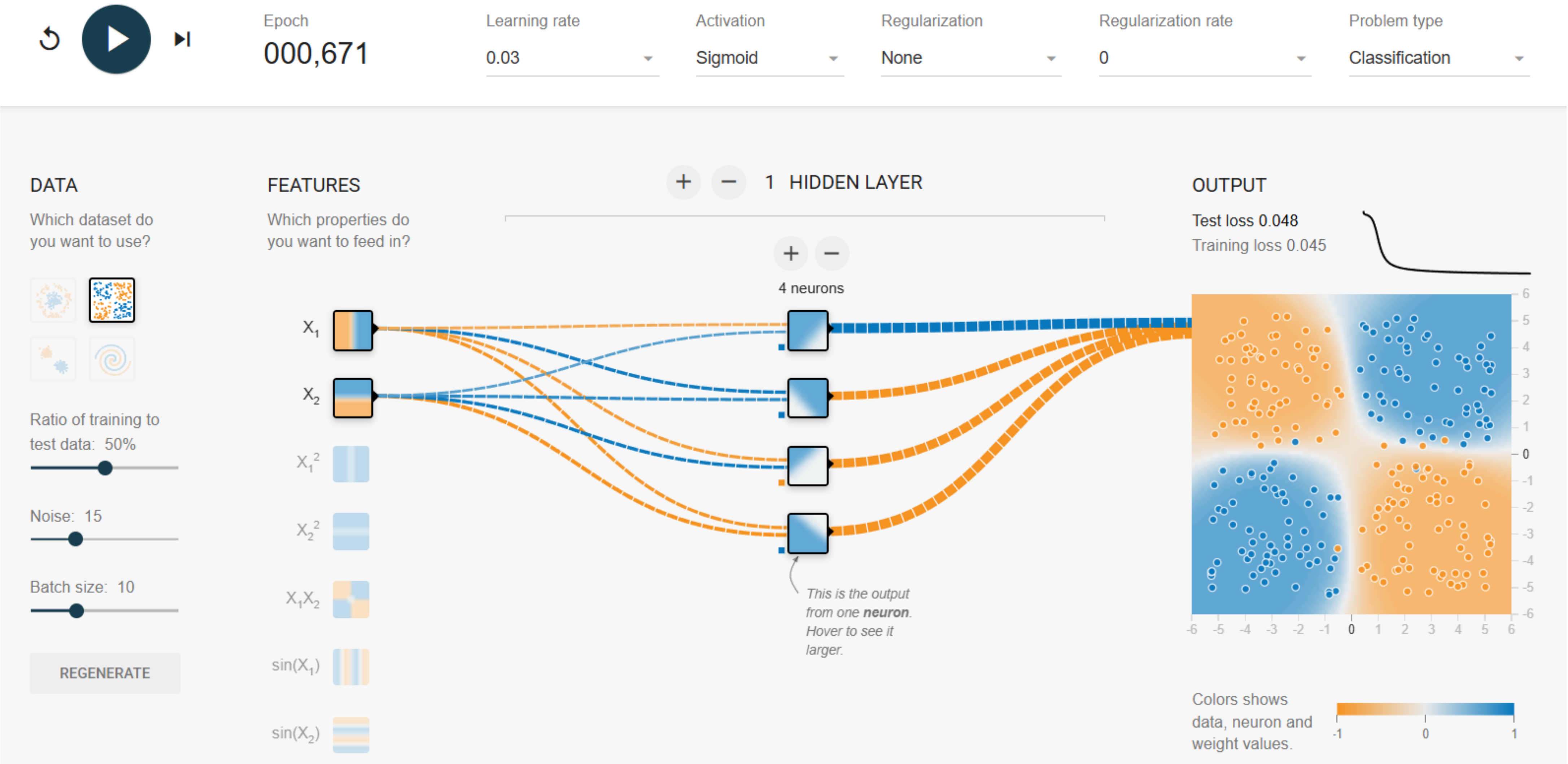
$$L = -[\hat{y} \cdot \ln(\hat{y}) + (1 - y). \ln(1 - \hat{y})]$$

- Backpropagation

$$\frac{dL}{dw_{11}^1} = ?$$

ANN Example

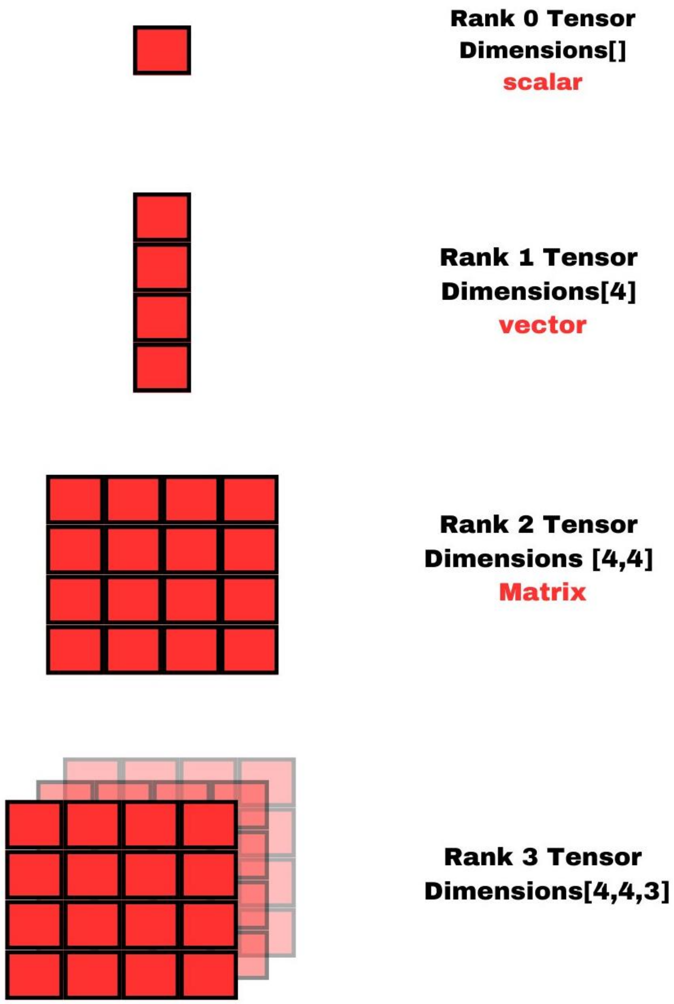
➢ XOR Dataset ANN Architecture




GOHIL
MILANSINH
AII
AND
AI
06
ROLL NO.
06
ROLL NO

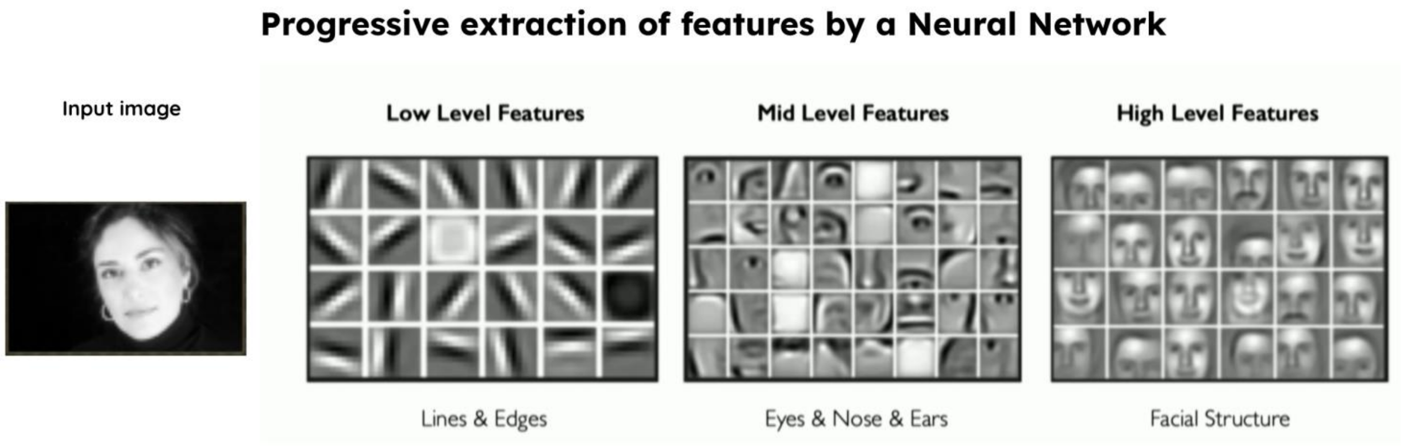
Tensors

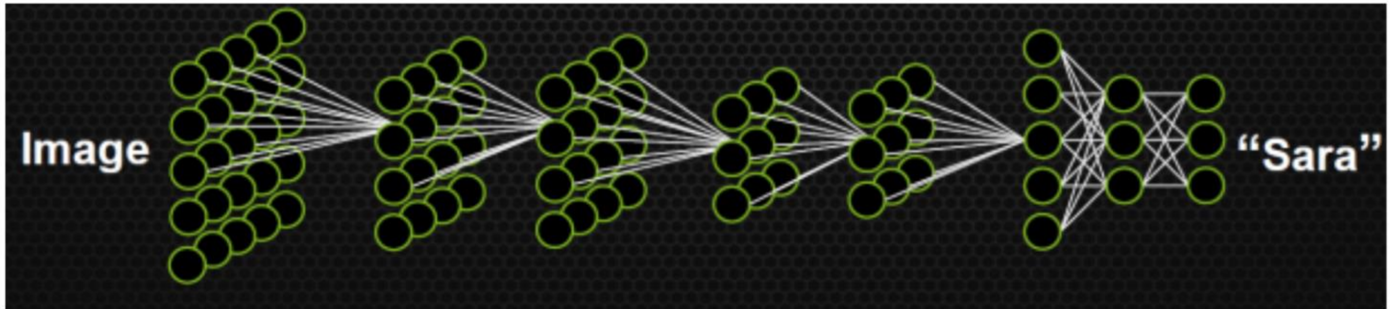
- Tensor is specialized multi-dimensional array designed for mathematical and computational efficiency.
- Types of Tensors:
  1. Scalar (0-D Tensor): Single number
  2. Vectors (1-D Tensor): A list of numbers
  3. Matrices (2-D Tensor): A 2-D grid of numbers
  4. 3D Tensor: Coloured images
  5. 4D Tensor: Batch of RGB images
  6. 5D Tensor: Video data
- Review our data and determine which data structure is most suitable for it.
- Data looks like [1254(b),120(r),240(c),1(channel)]
- For this type of data CNN is a good approach.

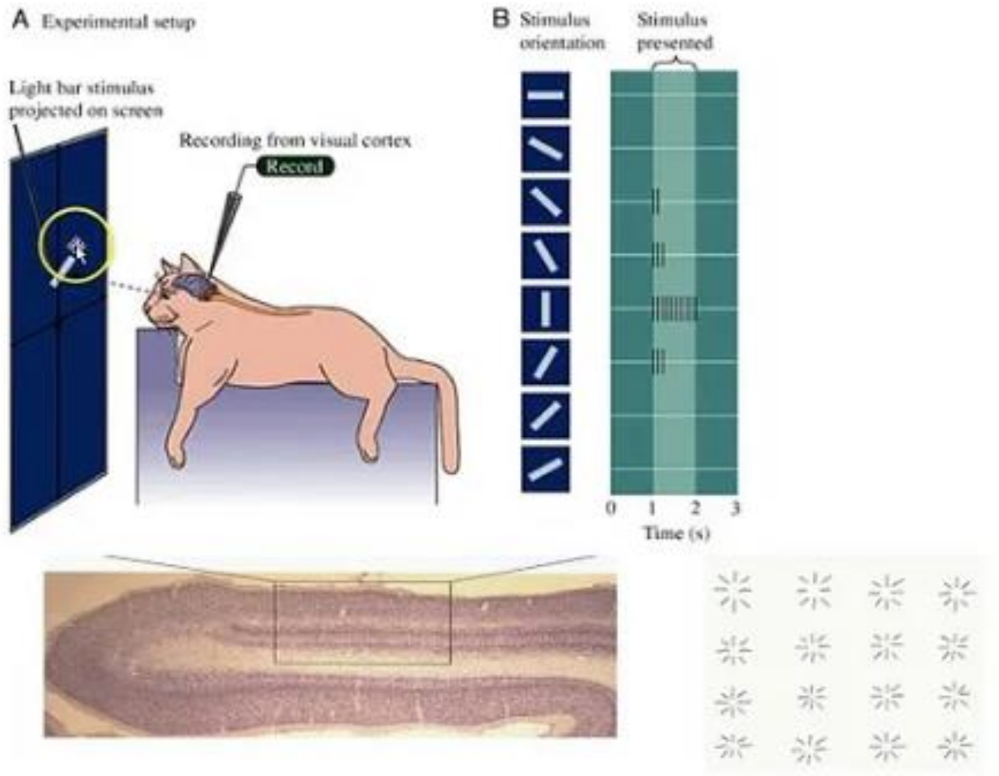


What is a CNN?

- Convolutional neural networks are a special kind of neural network for processing data that has a known grid-like topology like time series data(1D) or images(2D). It is inspired by our visual cortex.
- Hubel and Wiesel's famous cat experiment unlocked the mystery.

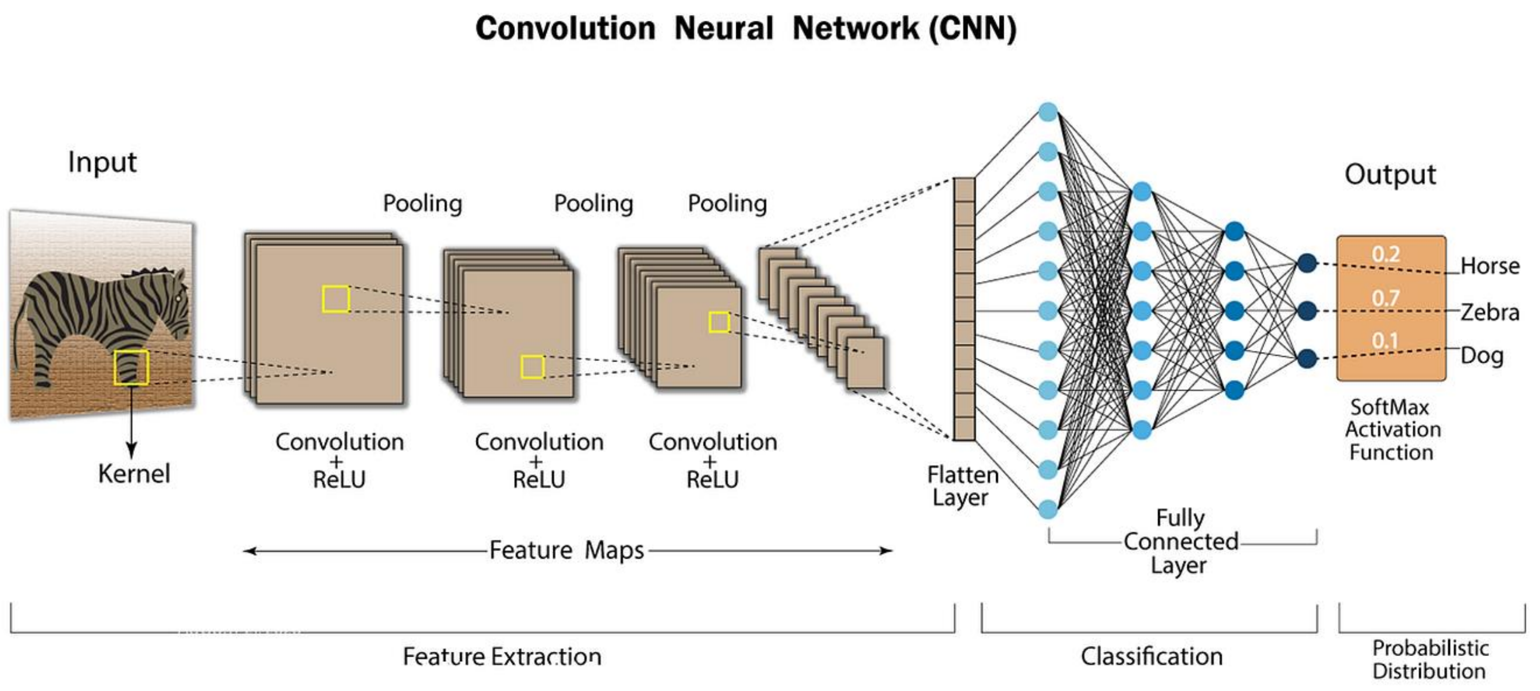






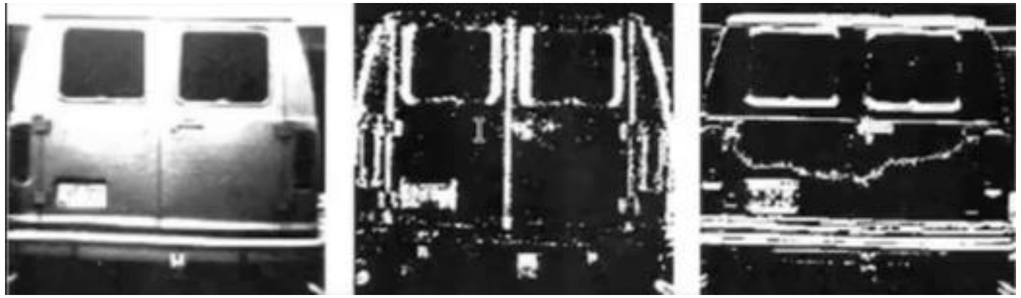
CNN Architecture

- CNNs have been split into two parts: Feature Extraction and Classification.
- CNNs are built from three basic layers:
  1. Convolution layer
  2. Pooling layer
  3. Fully Connected layer



Convolution layer

- The convolution layer detects edges and extracts primitive features.
- The convolution operation is essential for edge detection.



- This is the formula for finding the shape of the feature map $(n - f + 1) * (n - f + 1)$

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
255	255	255	255	255	255
255	255	255	255	255	255
255	255	255	255	255	255

Image $(n \times n)$

$*$

-1	-1	-1
0	0	0
1	1	1

Filter/Kernal
$(f \times f)$

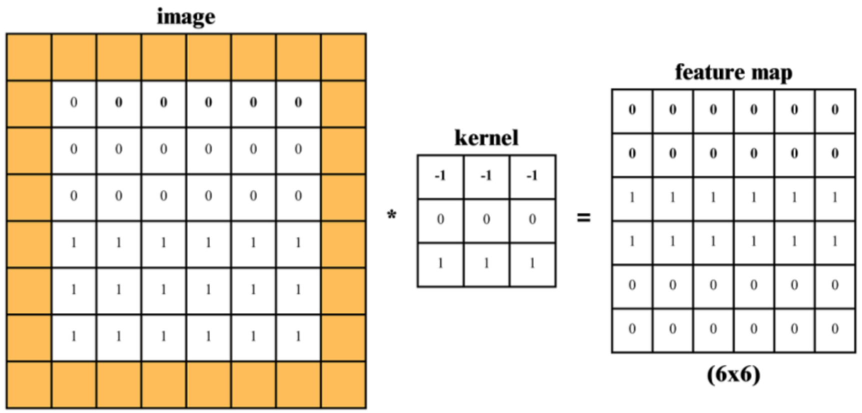
$=$

0	0	0	0
255	255	255	255
255	255	255	255
0	0	0	0

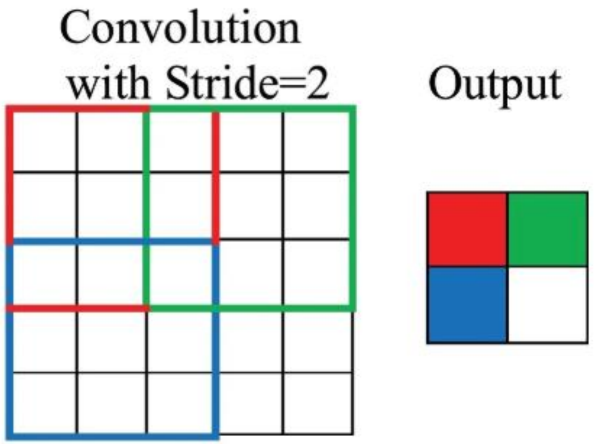
Feature map

Padding & Strides in Convolution layer

- Padding solves mainly two problems:
    1. It remains the image in the same resolution. So you don't lose the information.
    2. Now border pixels also take the same part as middle pixels do in convolution operation.



- This is the formula for finding the shape of the feature map with padding $(n + 2p - f + 1)$.
- Padding means it automatically adds pixel rows on top-bottom and adds pixel columns on left-right with zero values. That's why it's also said as zero padding.
- Strides is useful when you want high-level features.
- Strides reduce the computing power and make it efficient.
- This is the formula for finding the shape of the feature map with strides $(\frac{n-f}{s} + 1)$ & with padding and stride $(\frac{n+2p-f}{s} + 1)$.
- Stride value of (2,2) means it skips 2 pixels in right and bottom.



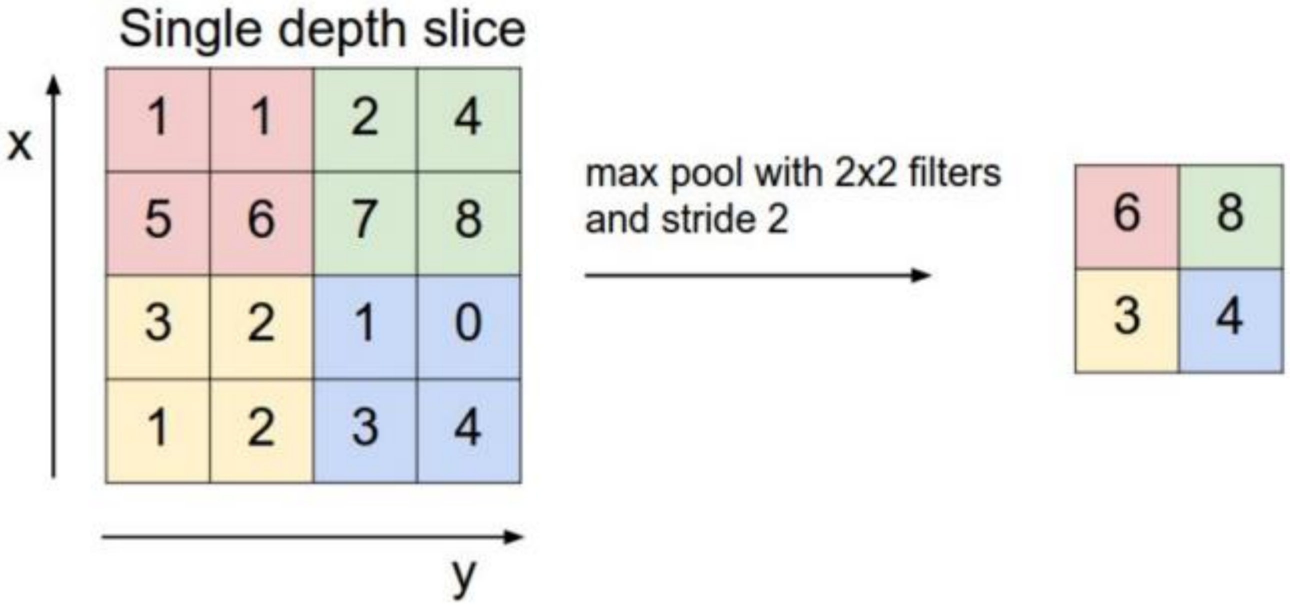
Pooling Operation

- There are two main problem that is efficiently solve by pooling operation.
  1. Memory issue
  2. Translation variance


Cat

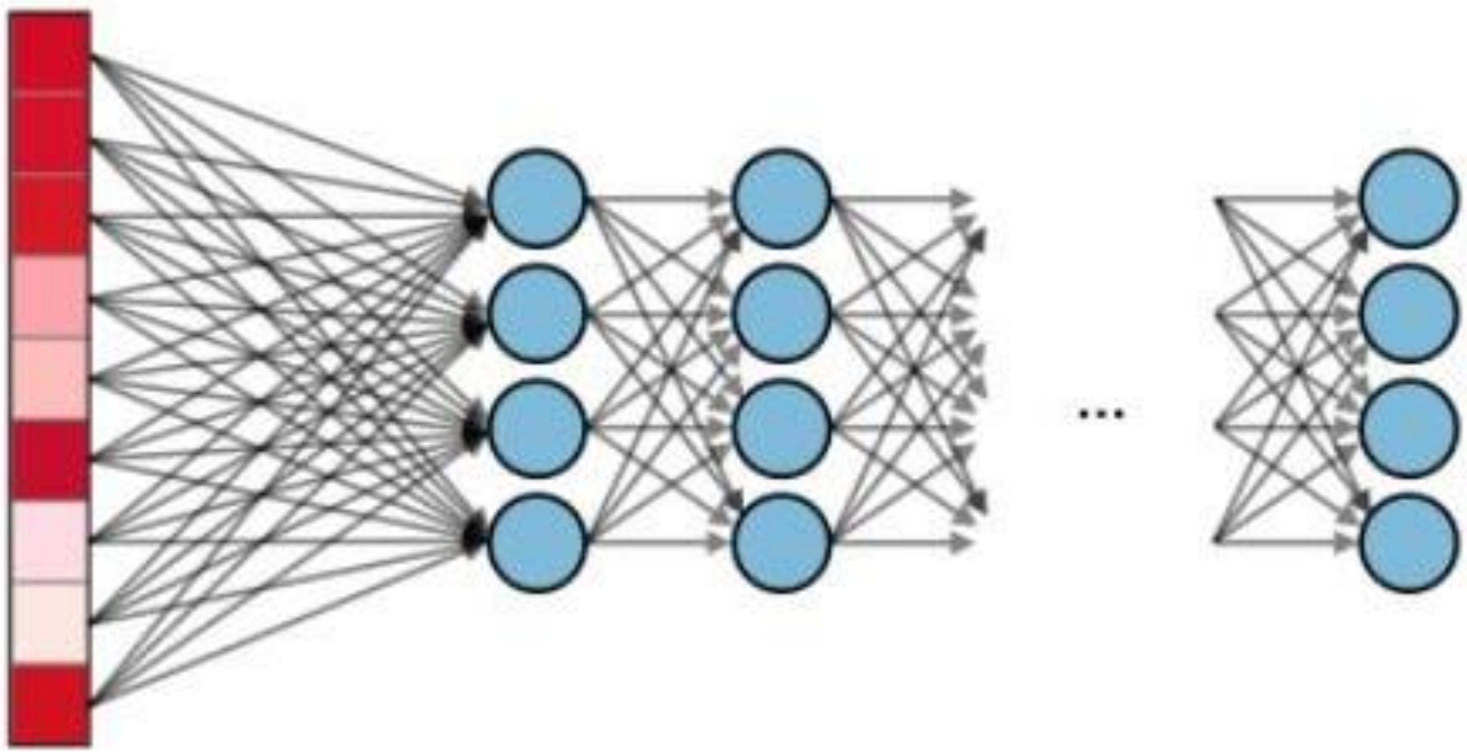

Cat

- Pooling is used to down-sample your feature map.
- There are generally 3 types of pooling layers are used.
  1. Max pooling
  2. Average pooling
  3. Global pooling
     a) Global Max pooling
     b) Global Average Pooling
- You will provide 3 parameters which are size, stride, and type.
- Used for extracting the dominant features from the feature map.
- It has a disadvantage in that it loses approximately 75% of data.



Classification Part

- Now, you flatten the last convolution layer and send it to the Fully Connected layer, which is the simple ANN structure.



Enable GPU & Preprocessing

- We enable GPU with the 'cuda' library.
- We preprocess the data.

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
device
```

```
device(type='cuda')
```

```
import datetime
import numpy as np
import torch

def datetime_to_vector(arr):
    years = arr.astype('datetime64[Y]').astype(int) + 1970
    months = (arr.astype('datetime64[M]').astype(int) % 12) + 1
    days = (arr.astype('datetime64[D]') - arr.astype('datetime64[M]')).astype('timedelta64[D]').astype(int) + 1
    hours = (arr.astype('datetime64[h]') - arr.astype('datetime64[D]')).astype('timedelta64[h]').astype(int)
    minutes = (arr.astype('datetime64[m]') - arr.astype('datetime64[h]')).astype('timedelta64[m]').astype(int)
    seconds = (arr.astype('datetime64[s]') - arr.astype('datetime64[m]')).astype('timedelta64[s]').astype(int)
    components = np.stack([years, months, days, hours, minutes, seconds], axis=1)
    # return components as torch tensor
    return torch.tensor(components,dtype=torch.float32)

date_vec = datetime_to_vector(adt_data.time.values)
date_vec = date_vec.to(device)
print(date_vec.shape)
```

```
torch.Size([1254, 6])
```

```
latitude_arr = adt_data.latitude.values
longitude_arr = adt_data.longitude.values
```

```
latitude_tensor = torch.tensor(latitude_arr, dtype=torch.float, device=device)
longitude_tensor = torch.tensor(longitude_arr, dtype=torch.float, device=device)
```

```
geo_grid_data = torch.cartesian_prod(latitude_tensor,longitude_tensor)
geo_grid_data = geo_grid_data.to(device)
```

```
geo_grid_data.shape
```

```
torch.Size([28800, 2])
```

```
# Given shapes:
# date: [1254, 6]
# geo: [28800, 2]
# Desired output shape: [1254, 28800, 8]

# Expand date to [1254, 28800, 6] (repeat geo entries for each date row)
date_expanded = date_vec.unsqueeze(1).expand(-1, 28800, -1)

# Expand geo to [1254, 28800, 2] (repeat date entries for each geo row)
geo_expanded = geo_grid_data.unsqueeze(0).expand(1254, -1, -1)

# Concatenate along the last dimension
combined = torch.cat((date_expanded, geo_expanded), dim=2)

print(combined.shape)  # Output: torch.Size([1254, 28800, 8])
```

```
torch.Size([1254, 28800, 8])
```

Data Loader, Dataset,Train-Test Split

- We define a class for loading data, which inherits from the built-in Dataset class of PyTorch.
- We also set the batch size for the training process.
- We take shuffle=False because our data is time series data.

```
class CustomDataset(Dataset):
  def __init__(self,X,y):
    self.X = X
    self.y = y

  def __len__(self):
    return self.X.shape[0]

  def __getitem__(self,index):
    return self.X[index], self.y[index]
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,shuffle=False)
```

```
train_dataset = CustomDataset(X_train,y_train)
test_dataset = CustomDataset(X_test,y_test)
```

```
train_dataloader = DataLoader(train_dataset, batch_size=1, shuffle=False)
test_dataloader = DataLoader(test_dataset, batch_size=1, shuffle=False)
```

NN Architecture & Training Loop

- We define a class that inherits to the nn.Module from Pytorch library of Python.
- Choosing Loss function as MSELoss and Adam Optimizer with 0.001 learning rate.
- We run the loop for the 50 epochs.

```python
class OceanNet(nn.Module):
    def __init__(self):
        super(OceanNet, self).__init__()

        self.encoder = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),

            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),

            nn.Flatten(),
            nn.Linear(30*60*64, 256),
            nn.ReLU(),
            nn.Dropout(0.3),

            nn.Linear(256, 120*240)
        )

    def forward(self, x):
        x = x.permute(0, 3, 1, 2)
        return self.encoder(x).view(-1, 120, 240, 1)
```

```python
model = OceanNet().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
criterion = nn.MSELoss()
```

```python
def train(model, data, epochs=50, batch_size=32):
    model.train()
    for epoch in range(epochs):
        permutation = torch.randperm(data.size(0))
        total_loss = 0

        for i in range(0, data.size(0), batch_size):
            indices = permutation[i:i+batch_size]
            batch = data[indices]

            # Forward pass
            outputs = model(batch)
            loss = criterion(outputs, batch)

            # Backward pass
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            total_loss += loss.item()

        print(f"Epoch {epoch+1}/{epochs}, Loss: {total_loss/(i+1):.4f}")

# Start training
train(model, train_tensor, epochs=50)
```

```
Epoch 1/50, Loss: 0.0309
Epoch 2/50, Loss: 0.0151
Epoch 3/50, Loss: 0.0112
Epoch 4/50, Loss: 0.0112
Epoch 5/50, Loss: 0.0104
Epoch 6/50, Loss: 0.0110
Epoch 7/50, Loss: 0.0103
Epoch 8/50, Loss: 0.0101
Epoch 9/50, Loss: 0.0102
Epoch 10/50, Loss: 0.0099
Epoch 11/50, Loss: 0.0094
Epoch 12/50, Loss: 0.0101
Epoch 13/50, Loss: 0.0096
Epoch 14/50, Loss: 0.0094
Epoch 15/50, Loss: 0.0096
Epoch 16/50, Loss: 0.0092
Epoch 17/50, Loss: 0.0088
Epoch 18/50, Loss: 0.0089
Epoch 19/50, Loss: 0.0090
Epoch 20/50, Loss: 0.0085
Epoch 21/50, Loss: 0.0083
Epoch 22/50, Loss: 0.0086
Epoch 23/50, Loss: 0.0084
Epoch 24/50, Loss: 0.0081
Epoch 25/50, Loss: 0.0079
Epoch 26/50, Loss: 0.0080
Epoch 27/50, Loss: 0.0077
Epoch 28/50, Loss: 0.0076
Epoch 29/50, Loss: 0.0076
Epoch 30/50, Loss: 0.0076
Epoch 31/50, Loss: 0.0074
Epoch 32/50, Loss: 0.0072
Epoch 33/50, Loss: 0.0072
Epoch 34/50, Loss: 0.0072
Epoch 35/50, Loss: 0.0070
Epoch 36/50, Loss: 0.0070
Epoch 37/50, Loss: 0.0069
Epoch 38/50, Loss: 0.0069
Epoch 39/50, Loss: 0.0067
Epoch 40/50, Loss: 0.0067
Epoch 41/50, Loss: 0.0066
Epoch 42/50, Loss: 0.0066
Epoch 43/50, Loss: 0.0064
Epoch 44/50, Loss: 0.0064
Epoch 45/50, Loss: 0.0065
Epoch 46/50, Loss: 0.0064
Epoch 47/50, Loss: 0.0063
Epoch 48/50, Loss: 0.0063
Epoch 49/50, Loss: 0.0063
Epoch 50/50, Loss: 0.0063
```

Evaluation

- We achieve 91.82% accuracy with our test data.
- Also we plot the graph of actual vs predicted.

```python
def calculate_accuracy(model, test_tensor, threshold=0.5):
    model.eval()
    with torch.no_grad():
        predictions = model(test_tensor)

        # Convert to class predictions (example thresholding)
        pred_classes = (predictions > threshold).float()
        true_classes = (test_tensor > threshold).float()

        correct = (pred_classes == true_classes).sum().item()
        total = true_classes.numel()

    accuracy = correct / total
    print(f"Accuracy: {accuracy:.4f}")
    return accuracy

# Usage
calculate_accuracy(model, test_tensor, threshold=0.5)
```

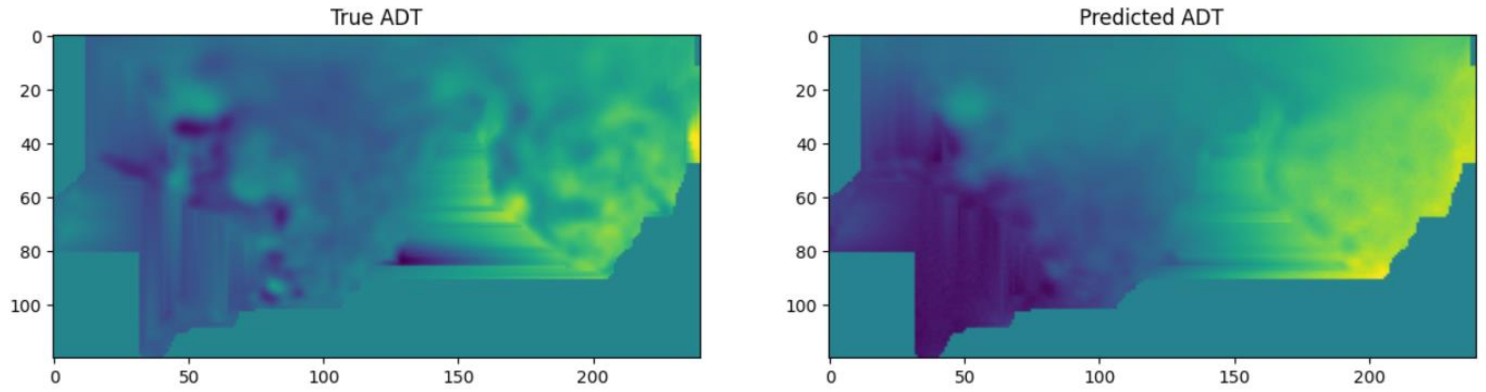
```
Accuracy: 0.9182
0.91822944333776
```

```python
import matplotlib.pyplot as plt

def plot_comparison(sample_idx=0):
    with torch.no_grad():
        pred = model(test_tensor[sample_idx].unsqueeze(0)).cpu().numpy()
        true = test_tensor[sample_idx].cpu().numpy()

    fig, ax = plt.subplots(1, 2, figsize=(15, 5))
    ax[0].imshow(true[..., 0], cmap='viridis')
    ax[0].set_title('True ADT')
    ax[1].imshow(pred[0, ..., 0], cmap='viridis')
    ax[1].set_title('Predicted ADT')
    plt.show()

plot_comparison(sample_idx=42)
```



Problem with CNN

- CNNs, while powerful for image recognition, lack the ability to retain sequential information.
- This limitation is critical in time series forecasting where models need to leverage historical patterns.
- Specialized architectures like Recurrent Neural Networks (RNNs), Long Short-Term Memory (LSTM) networks, and Gated Recurrent Units (GRUs) are used to address this need.
- These models capture temporal relationships and update with each time step, making them more effective for tasks involving sequences or time-dependent data.


deep
learning
filters
convolustion
filters
CNN
ANN
recurrion loop
THANK YOU
RNSTM