**EXPERIMENT 1** Date:

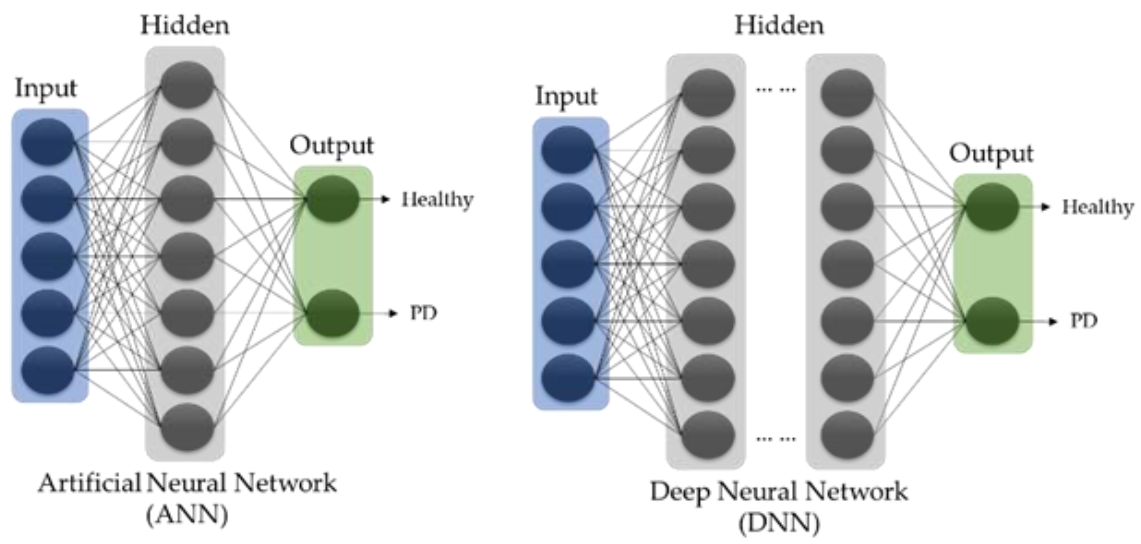
**Problem Definition:** Study of different libraries of ANN and DNN

**Packages Used:** PyTorch, matplotlib

**Dataset Used:** MNIST dataset

**Theory:**

Artificial Neural Networks (ANN) and Deep Neural Networks (DNN) are powerful models that have become essential in solving complex machine learning tasks such as image classification, language translation, and more. This experiment aims to explore PyTorch, a widely used library for building and training neural networks, and to demonstrate how to implement a simple neural network model for a classification problem using PyTorch.



**Artificial Neural Networks (ANN)** are computational models inspired by the human brain's neural networks. They consist of layers of neurons that process inputs, apply transformations, and generate outputs. ANNs have applications in classification, regression, and clustering problems.

**Deep Neural Networks (DNN)** are an extension of ANNs with many hidden layers, enabling them to learn complex patterns from vast datasets. The rise of DNNs has made significant advances in fields such as computer vision, natural language processing, and game playing.

**Key Concepts**:

* **Neurons**: Basic processing units that receive inputs, apply weights, and activate outputs.
* **Layers**: ANNs consist of input, hidden, and output layers. In DNNs, the hidden layers are significantly deeper.
* **Activation Functions**: Functions like ReLU, Sigmoid, and Tanh introduce non-linearity in the network.
* **Backpropagation**: The method used to train neural networks by minimizing the error function.

**Applications**:

* Image recognition (e.g., facial recognition)
* Speech-to-text conversion
* Autonomous driving

**Libraries for ANN and DNN**

1. **PyTorch**

PyTorch is an open-source machine learning library developed by Facebook. It’s known for its flexibility and ease of use, particularly due to its support for dynamic computation graphs. PyTorch is a preferred library in the research community because of its Pythonic nature, easy debugging, and strong support for GPU acceleration.

**Features**:

Dynamic Computation Graphs: Allows on-the-fly graph building, which is great for variable-length input data and experimentation.

GPU Acceleration: Provides seamless support for NVIDIA GPUs to speed up training.

Extensive Ecosystem: Integration with libraries like torchvision for datasets and pre-trained models.

Community Support: Large and active community, with numerous tutorials, forums, and contributions.

**Other Libraries are**

1. **TensorFlow**

* **Overview**: TensorFlow, developed by Google, is one of the most widely used libraries for building machine learning models. It supports a range of machine learning tasks and provides both high-level and low-level APIs.
* **Features**: It supports GPU acceleration, distributed computing, and has extensive support for deploying models in production.
* **Use Case**: TensorFlow is widely used in real-time applications such as image recognition, voice assistants, and recommendation systems.

1. **Keras**

* **Overview**: Keras is a high-level API built on top of TensorFlow that focuses on rapid experimentation and ease of use. It abstracts much of the complexity of TensorFlow, making it suitable for beginners.
* **Features**: Its modularity, simplicity, and user-friendly API make it an ideal choice for prototyping.
* **Use Case**: Keras is often used for academic research, educational purposes, and fast model prototyping.

1. **Theano**

* **Overview**: Theano is one of the earliest deep learning libraries, providing symbolic differentiation and efficient computations. However, it has been largely replaced by newer libraries like TensorFlow and PyTorch.
* **Features**: Though not as widely used now, Theano is strong in terms of mathematical computations and GPU acceleration.
* **Use Case**: Theano was widely used in academic research in the early stages of deep learning development.

**Experiment Setup and Implementation:**

Hardware:

* **GPU**: Recommended (NVIDIA GPU with CUDA support)

Software:

* **Python 3.x**
* **PyTorch**: Version torch==2.x
* **Other Packages**:
  + torchvision for datasets and data augmentation.
  + numpy for data manipulation.
  + matplotlib for plotting results.

Installation Instructions:

1. **Install PyTorch**: If you have a GPU, you can install PyTorch with CUDA support. Use the following command:

pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118

1. **Install Other Dependencies**:

pip install numpy matplotlib

Dataset Used

* MNIST Dataset: A popular dataset of handwritten digits (0-9), with 60,000 training samples and 10,000 test samples, each image being 28x28 pixels in grayscale.
* It can be loaded easily using torchvision.datasets.

**Implementation of a Neural Network in Pytorch:**

Import Libraries

import torch

import torch.nn as nn

import torch.optim as optim

import torchvision

import torchvision.transforms as transforms from torch.utils.data

import DataLoader import matplotlib.pyplot as plt

import matplotlib.pyplot as plt

Load and Preprocess the dataset

# Define transformations for the dataset (convert to tensor and normalize)

transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])

# Load MNIST dataset

train\_dataset = torchvision.datasets.MNIST(root='./data', train=True, download=True, transform=transform)

test\_dataset = torchvision.datasets.MNIST(root='./data', train=False, download=True, transform=transform)

# Create data loaders for batching

train\_loader = DataLoader(dataset=train\_dataset, batch\_size=64, shuffle=True)

test\_loader = DataLoader(dataset=test\_dataset, batch\_size=64, shuffle=False)

Define the neural network architecture

class SimpleANN(nn.Module):

def \_\_init\_\_(self):

super(SimpleANN, self).\_\_init\_\_()

self.fc1 = nn.Linear(28\*28, 128) # Input layer

self.fc2 = nn.Linear(128, 64) # Hidden layer

self.fc3 = nn.Linear(64, 10) # Output layer (10 classes)

def forward(self, x):

x = x.view(-1, 28\*28) # Flatten the 2D image into a 1D vector

x = torch.relu(self.fc1(x))

x = torch.relu(self.fc2(x))

x = self.fc3(x)

return x

Initialise Mode, Loss Function and Optimizer

model = SimpleANN()

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=0.001)

Training the model

# Training loop

epochs = 5

for epoch in range(epochs):

running\_loss = 0.0

for images, labels in train\_loader:

# Zero the gradients

optimizer.zero\_grad()

# Forward pass

outputs = model(images)

loss = criterion(outputs, labels)

# Backward pass and optimization

loss.backward()

optimizer.step()

running\_loss += loss.item()

print(f'Epoch [{epoch+1}/{epochs}], Loss: {running\_loss/len(train\_loader):.4f}')

 Epoch [1/5], Loss: 0.4108

Epoch [2/5], Loss: 0.2022

Epoch [3/5], Loss: 0.1452

Epoch [4/5], Loss: 0.1162

Epoch [5/5], Loss: 0.0960

Evaluating the model

correct = 0

total = 0

with torch.no\_grad(): # No need to track gradients during evaluation

for images, labels in test\_loader:

outputs = model(images)

\_, predicted = torch.max(outputs.data, 1)

total += labels.size(0)

correct += (predicted == labels).sum().item()

print(f'Accuracy of the model on the test images: {100 \* correct / total:.2f}%')

 Accuracy of the model on the test images: 96.43%

**Conclusion:**

PyTorch was successfully studied and implemented.