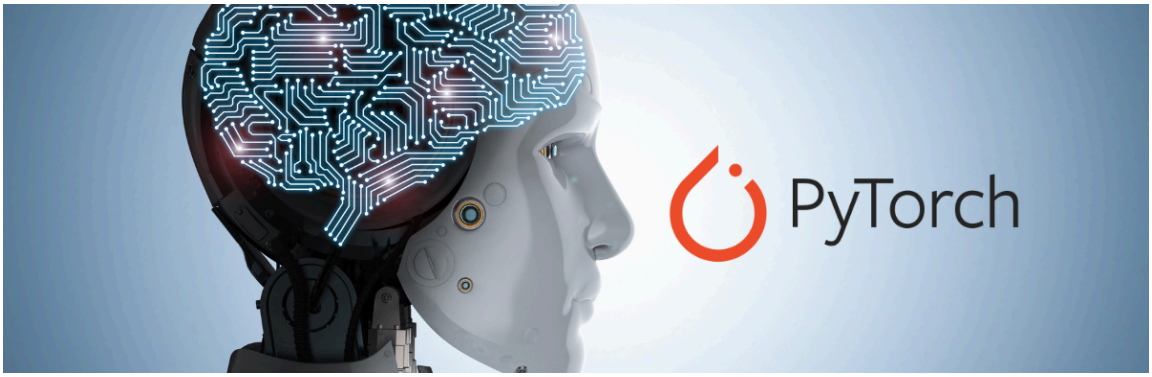


Pytorch Neural Network Modelling

Project Overview:

This project aims to provide a comprehensive guide to understanding and implementing various types of neural networks using PyTorch. It is designed to help beginners in deep learning gain a strong foundation in the key concepts and practical skills needed to build and train neural network models. The project covers linear models, recurrent neural networks (RNNs), long short-term memory networks (LSTMs), gated recurrent units (GRUs), and convolutional neural networks (CNNs).

What I Have Done:

In this project, I have created detailed Jupyter notebooks that include the following sections for each neural network type:

1. **Linear Models**: Demonstrates a simple linear regression example using synthetic data.
2. **Recurrent Neural Networks (RNNs)**: Provides a basic RNN implementation for time series prediction using synthetic sine wave data.
3. **Long Short-Term Memory Networks (LSTMs)**: Illustrates an LSTM for time series prediction using synthetic sine wave data.
4. **Gated Recurrent Units (GRUs)**: Shows a GRU implementation for time series prediction using synthetic sine wave data.
5. **Convolutional Neural Networks (CNNs)**: Features a CNN for image classification using the MNIST dataset.

Each section includes:

- **Model Definition**: Detailed description of the model architecture and components.
- **Data Generation/Preprocessing**: Steps for generating and preprocessing the data used for training and testing the models.
- **Training Loop**: The process of training the models, including loss computation and optimization.
- **Testing/Validation**: Evaluation of the models on test data to assess their performance.
- **Visualization (where applicable)**: Visual representation of the results to aid in understanding the model's performance and behavior.

Methodologies Used:

1. **Data Generation/Preprocessing**: Synthetic data generation and preprocessing techniques such as normalization and reshaping.
2. **Model Definition**: Implementation of different neural network architectures using PyTorch's `nn.Module`.
3. **Training Loop**: Standard training loops involving forward pass, loss computation, backward pass, and parameter updates using optimizers like SGD and Adam.
4. **Evaluation**: Techniques for testing and validating the models' performance, including accuracy and loss metrics.
5. **Visualization**: Use of matplotlib for visualizing data, training progress, and model predictions.

Objectives:

- **Educational**: Provide a clear and practical guide for beginners to understand and implement different types of neural networks using PyTorch.
- **Hands-on Learning**: Encourage hands-on practice through detailed code examples and explanations.
- **Foundation Building**: Help learners build a strong foundation in deep learning concepts and PyTorch programming.
- **Resource Provision**: Offer a resource that learners can refer to for future projects and studies in deep learning.

How This Project Helps Beginners:

This project serves as a comprehensive introduction to deep learning and PyTorch. By following the detailed examples and explanations, beginners can:

- Gain a clear understanding of different neural network architectures and their applications.
- Learn how to preprocess data, define models, train and evaluate neural networks.
- Develop practical skills in using PyTorch for deep learning tasks.
- Build confidence in implementing and experimenting with various neural network models.
- Have a reference guide that they can use as they advance in their deep learning journey.

This project not only teaches the theoretical aspects of neural networks but also emphasizes practical implementation, making it an invaluable resource for anyone starting out in deep learning with PyTorch.

ABOUT NOTEBOOK

This notebook covers the following sections:

1. **Linear Models**: A simple linear regression example using synthetic data.

2. **Recurrent Neural Networks (RNNs)**: A basic RNN for time series prediction using synthetic sine wave data.
3. **Long Short-Term Memory Networks (LSTMs)**: An LSTM for time series prediction using synthetic sine wave data.
4. **Gated Recurrent Units (GRUs)**: A GRU for time series prediction using synthetic sine wave data.
5. **Convolutional Neural Networks (CNNs)**: A CNN for image classification using the MNIST dataset.

Each section includes the following components:

- **Model Definition**: Detailed description of the model architecture and components.
- **Data Generation/Preprocessing**: Steps for generating and preprocessing the data used for training and testing the models.
- **Training Loop**: The process of training the models, including loss computation and optimization.
- **Testing/Validation**: Evaluation of the models on test data to assess their performance.
- **Visualization (where applicable)**: Visual representation of the results to aid in understanding the model's performance and behavior.

Introduction to PyTorch

Why PyTorch?:

PyTorch is an open-source deep learning framework developed by Facebook's AI Research lab. It has gained popularity due to its dynamic computation graph, ease of use, and strong community support.

Advantages of PyTorch:

1. **Dynamic Computation Graphs**: Unlike static computation graphs used in other frameworks (like TensorFlow), PyTorch uses dynamic computation graphs. This means the graph is built on-the-fly as operations are executed, making it easier to debug and modify.
2. **Pythonic Nature**: PyTorch integrates seamlessly with Python, making it intuitive and easy to learn, especially for Python developers.
3. **Extensive Library Support**: PyTorch has extensive support for various neural network layers, loss functions, and optimization algorithms, making it a versatile choice for building complex models.
4. **Strong Community and Ecosystem**: PyTorch has a large and active community, providing numerous resources, tutorials, and third-party libraries. This support accelerates development and troubleshooting.
5. **Performance**: PyTorch efficiently utilizes GPUs for accelerated computing, providing significant speedups in training deep learning models.

Basic Usage of PyTorch:

Here is a basic example to illustrate how to use PyTorch for building and training a simple neural network:

1. **Installation**:

```
pip install torch torchvision
```

2. **Basic Workflow**:

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms

# Define a simple neural network
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        x = torch.flatten(x, 1)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# Initialize the network, loss function, and optimizer
model = SimpleNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Load and preprocess the dataset
transform = transforms.Compose([transforms.ToTensor(),
transforms.Normalize((0.5,), (0.5,))])
trainset = torchvision.datasets.MNIST(root='./data', train=True,
download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64,
shuffle=True)

# Training loop
for epoch in range(5):  # number of epochs
    running_loss = 0.0
    for inputs, labels in trainloader:
        optimizer.zero_grad()  # zero the parameter gradients
        outputs = model(inputs)  # forward pass
        loss = criterion(outputs, labels)  # compute loss
        loss.backward()  # backward pass
        optimizer.step()  # optimize the parameters
        running_loss += loss.item()
    print(f'Epoch {epoch+1}, Loss:
{running_loss/len(trainloader):.4f}')

print('Finished Training')
```

This example demonstrates how to define a simple neural network, load a dataset, and train the model using PyTorch. PyTorch's flexibility and intuitive design make it an excellent choice

for both beginners and experienced practitioners in deep learning.

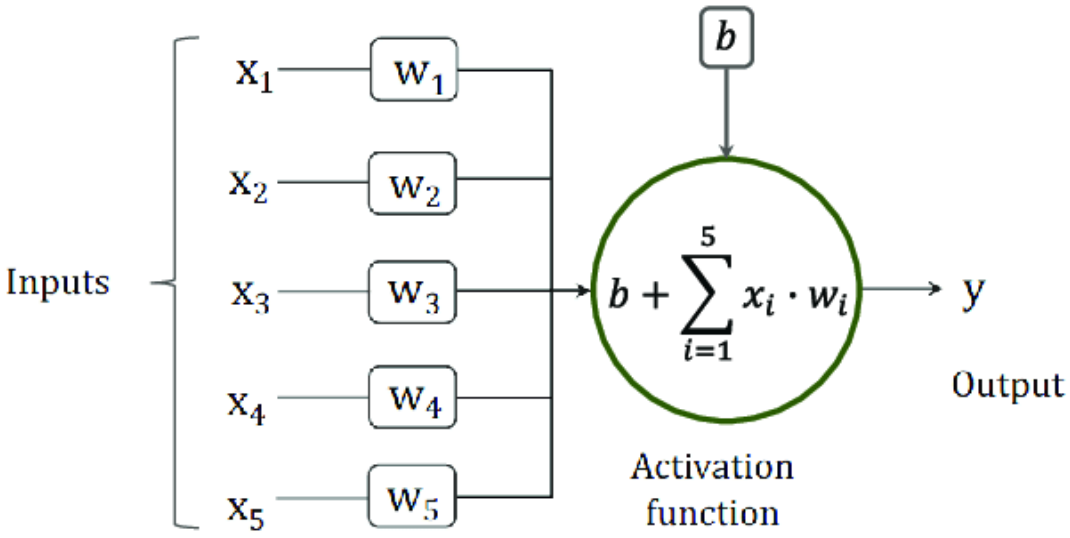
Overview of Neural Networks

Neural Networks:

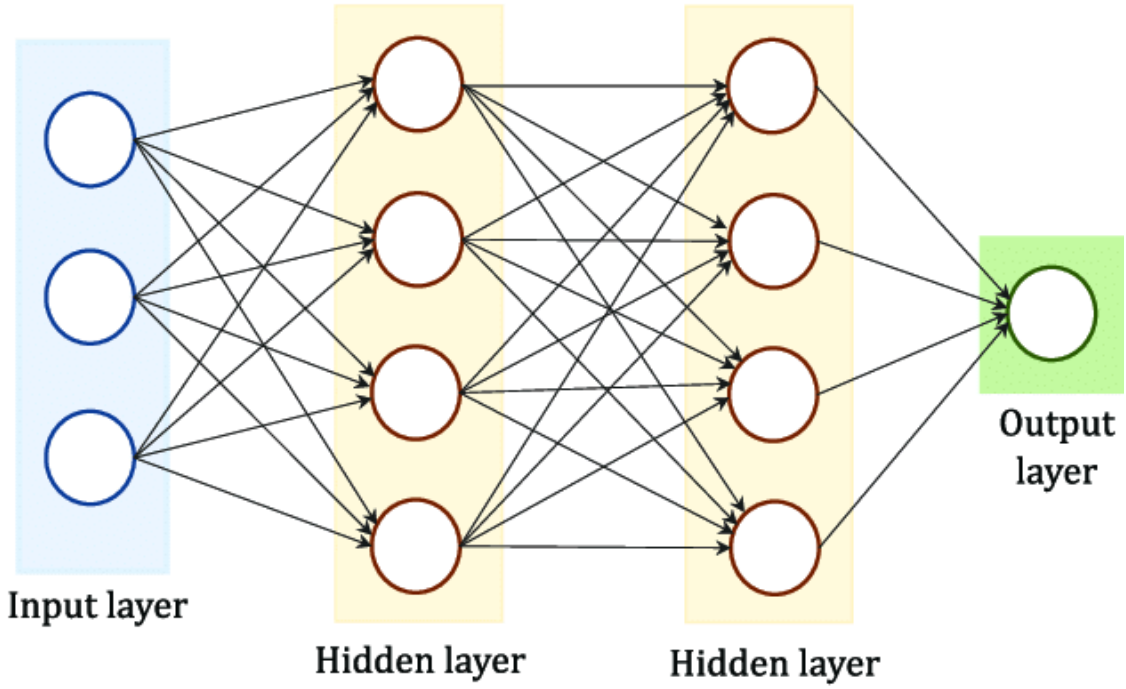
Neural networks are a subset of machine learning algorithms inspired by the structure and function of the human brain. They consist of interconnected layers of nodes (neurons) that process data in complex ways, allowing the network to learn and make predictions. Neural networks can model complex, non-linear relationships in data, making them powerful tools for various tasks.

Importance of Neural Networks:

1. **Versatility**: Neural networks can be applied to a wide range of problems, including classification, regression, and clustering.
2. **Feature Learning**: They can automatically learn features from raw data, reducing the need for manual feature engineering.
3. **High Performance**: Neural networks, especially deep learning models, have achieved state-of-the-art performance in many domains, such as image recognition, natural language processing, and game playing.
4. **Scalability**: With the advent of large datasets and powerful computational resources, neural networks can be scaled to handle massive amounts of data, leading to better generalization and accuracy.



(a) A perceptron with $N = 5$ inputs x_i and one output y.



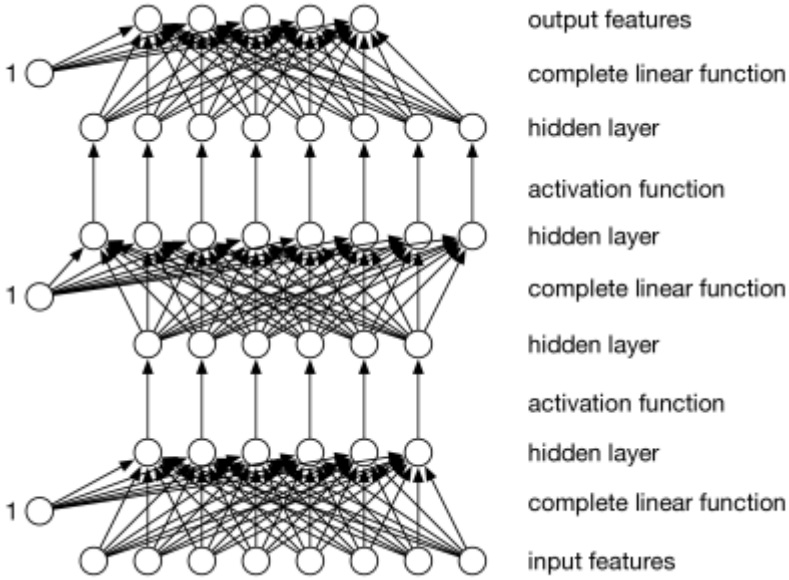
(b) Feed-forward fully connected neural network.

In [2]:

```
# Import necessary libraries
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt

# Helper function for generating synthetic data
def generate_data(seq_length):
    time_series = np.sin(np.linspace(0, 100, seq_length))
    data = []
    for i in range(len(time_series) - 1):
        data.append((time_series[i:i+1], time_series[i+1]))
    return data

# Helper function for plotting results
def plot_results(predicted, actual, title):
    plt.figure(figsize=(10, 5))
    plt.plot(predicted, label='Predicted')
```

```
    plt.plot(actual, label='Actual')
    plt.title(title)
    plt.legend()
    plt.show()
```

LINEAR MODELS



Linear Models:

Linear models are the simplest type of predictive model, where the relationship between the input variables and the output is assumed to be linear. The most common linear model is linear regression.

How Linear Models Work:

In a linear model, the output (y) is calculated as a weighted sum of the inputs (x), plus a bias term: $\[y = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b \]$ where (w_i) are the weights and (b) is the bias. The model parameters (weights and bias) are typically learned using optimization algorithms such as gradient descent, which minimizes a loss function like mean squared error (MSE).

Use Cases of Linear Models:

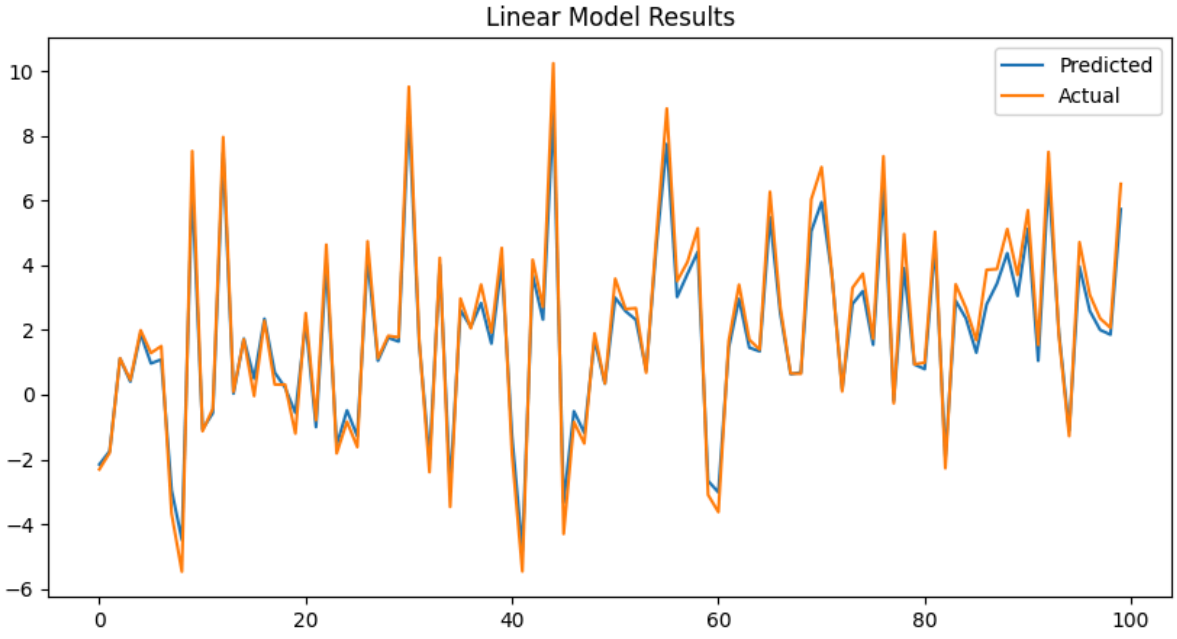
Linear models are widely used in various fields due to their simplicity and interpretability. Some common use cases include:

- **Regression Analysis**: Predicting continuous outcomes, such as house prices, sales forecasts, and medical measurements.
- **Classification**: Logistic regression, a type of linear model, is used for binary classification tasks such as spam detection and disease diagnosis.
- **Econometrics**: Analyzing economic data to understand relationships between variables, such as the impact of interest rates on investment.
- **Engineering**: Modeling physical systems where relationships between variables are approximately linear.

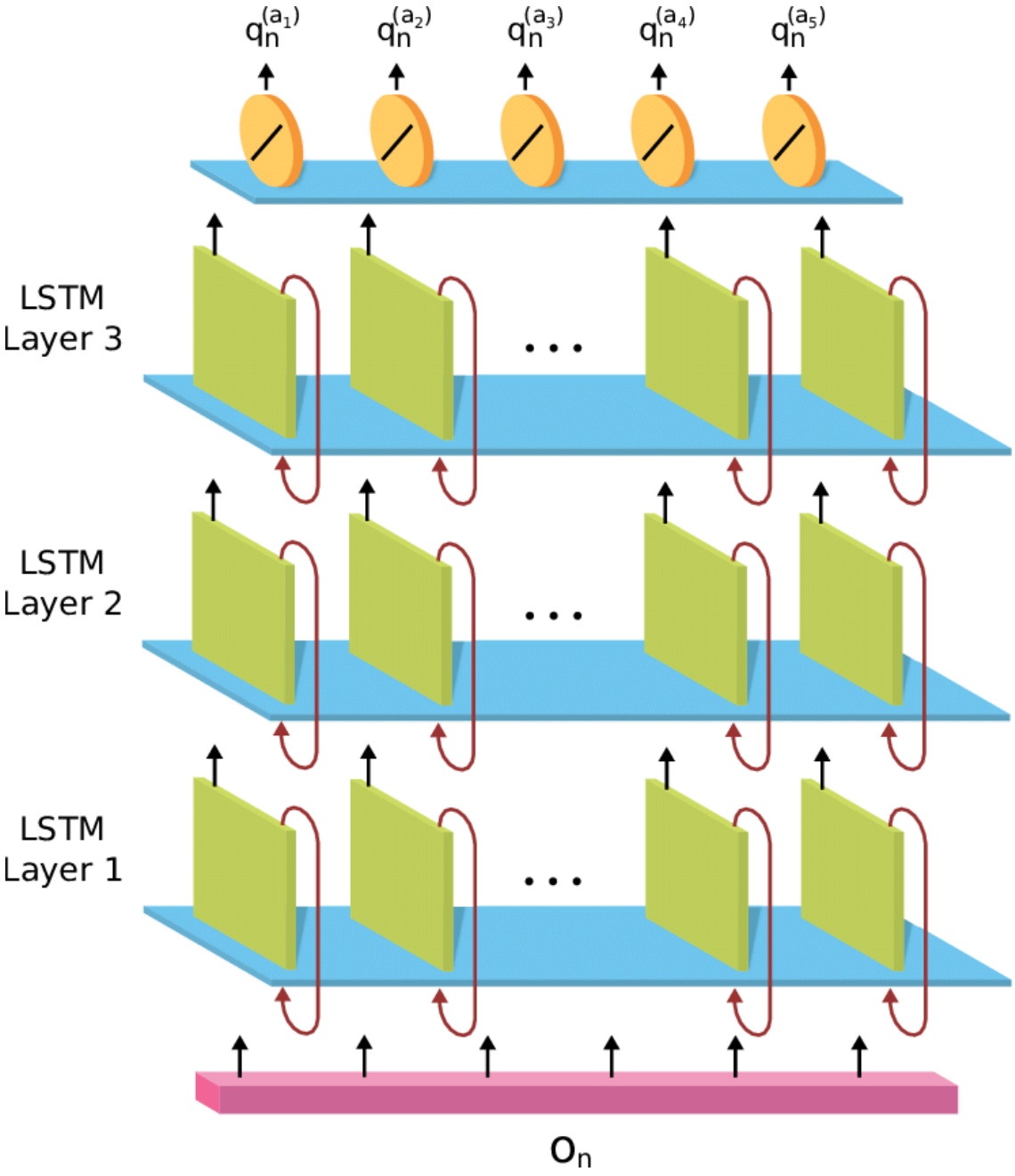
In [2]:
```
# Sample code for Linear Models
class LinearModel(nn.Module):
    def __init__(self, input_size, output_size):
        super(LinearModel, self).__init__()
        self.linear = nn.Linear(input_size, output_size)

    def forward(self, x):
        return self.linear(x)

def train_linear_model():
    # Generate synthetic data
    x_train = torch.randn(100, 1)
    y_train = 3 * x_train + 2 + 0.2 * torch.randn(100, 1)

    # Model, loss, optimizer
    model = LinearModel(1, 1)
    criterion = nn.MSELoss()
    optimizer = optim.SGD(model.parameters(), lr=0.01)

    # Training loop
    for epoch in range(100):
        model.train()
        optimizer.zero_grad()
        outputs = model(x_train)
        loss = criterion(outputs, y_train)
        loss.backward()
        optimizer.step()

    print('Linear Model Training Completed')

    # Plotting results
    with torch.no_grad():
        predicted = model(x_train).numpy()
        actual = y_train.numpy()
        plot_results(predicted, actual, 'Linear Model Results')
```

In [3]:
```
train_linear_model()
```

```
Linear Model Training Completed
```



RECURRENT NEURAL NETWORKS



Recurrent Neural Networks (RNNs): RNNs are a type of neural network designed to recognize patterns in sequences of data, such as time series, speech, and text. They use loops within the network to maintain information in what's called the "hidden state," allowing them to capture temporal dynamics.

How RNNs Work:

In an RNN, the output from the previous time step is fed back into the network along with the current input. This feedback loop allows RNNs to maintain a memory

of previous inputs, enabling them to process sequences of data. The key components of an RNN are:

1. **Input Layer**: Takes the input data at each time step.
2. **Hidden Layer**: Maintains the hidden state, which captures information from previous time steps.
3. **Output Layer**: Produces the output at each time step.

The hidden state is updated at each time step using the current input and the previous hidden state, typically using an activation function like tanh or ReLU.

Use Cases of RNNs:

RNNs are versatile and can be applied to various sequence prediction and generation tasks. Some common use cases include:

- **Time Series Prediction**: Predicting future values in a sequence, such as stock prices or weather data.
- **Natural Language Processing (NLP)**: Tasks such as language modeling, text generation, machine translation, sentiment analysis, and speech recognition.
- **Music Generation**: Composing music by predicting the next note or chord in a sequence.
- **Video Analysis**: Understanding sequences of video frames for tasks such as activity recognition and video captioning.

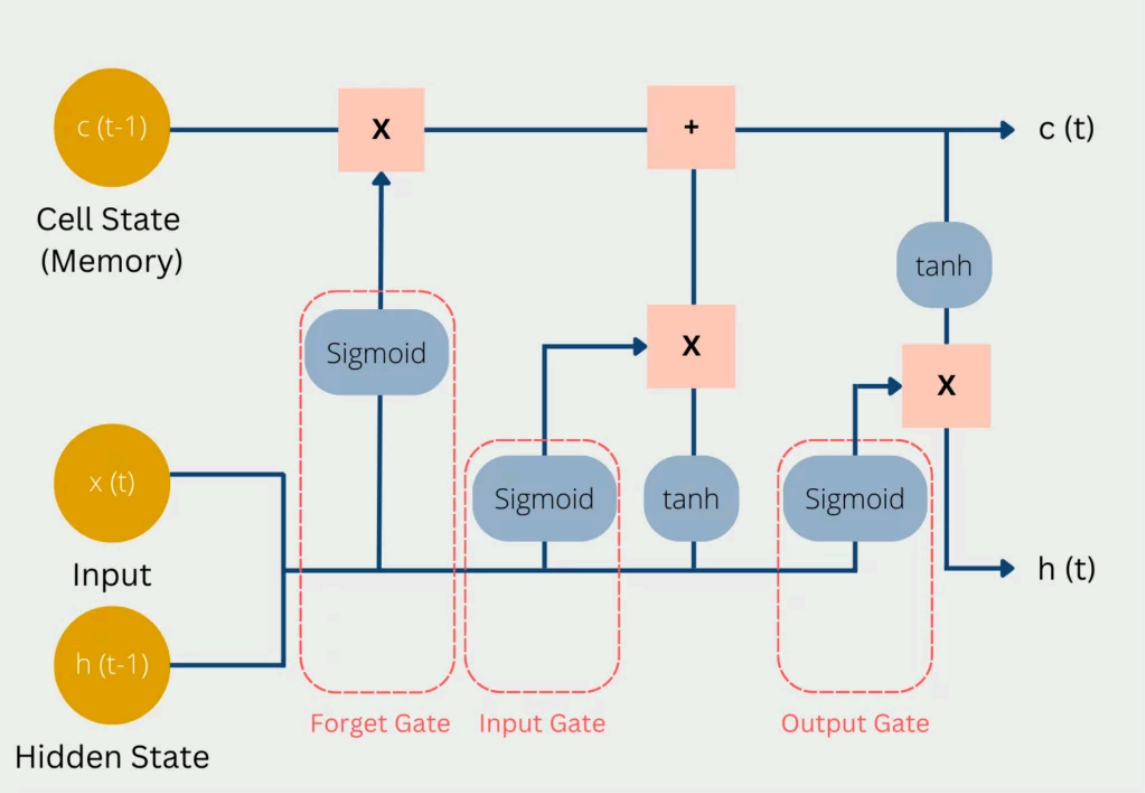
In [4]:
```
#Sample code for an RNN model
class RNNModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNNModel, self).__init__()
        self.hidden_size = hidden_size
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
        self.linear = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h0 = torch.zeros(1, x.size(0), self.hidden_size)
        out, _ = self.rnn(x, h0)
        out = self.linear(out[:, -1, :])
        return out

def train_rnn_model():
    # Generate synthetic data
    data = generate_data(100)
    data = [(torch.tensor(x, dtype=torch.float32).unsqueeze(0),
             torch.tensor(y, dtype=torch.float32).unsqueeze(0))
            for x, y in data]
    dataloader = torch.utils.data.DataLoader(data, batch_size=1, shuffle=True)

    # Model, loss, optimizer
    model = RNNModel(1, 50, 1)
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    # Training loop
    for epoch in range(100):
        for seq, target in dataloader:
            optimizer.zero_grad()
            output = model(seq)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()

    print('RNN Model Training Completed')

    # Testing the model
    test_seq = torch.tensor(np.sin(np.linspace(0, 100, 100)), dtype=torch.float32).
    with torch.no_grad():
```

```
        test_output = model(test_seq)
    print(f'RNN Predicted next value: {test_output.item():.4f}')
```

In [5]:
```
train_rnn_model()
```

```
RNN Model Training Completed
RNN Predicted next value: -0.2637
```

LONG SHORT MEMORY NETWORKS



Long Short-Term Memory Networks (LSTMs): LSTMs are a type of recurrent neural network (RNN) architecture that is specifically designed to avoid the long-term dependency problem, which standard RNNs suffer from. They were introduced by Hochreiter and Schmidhuber in 1997 and have been refined and popularized since then.

How LSTMs Work:

LSTMs work by introducing a memory cell and three types of gates (input gate, forget gate, and output gate) to control the flow of information. Here's a breakdown of these components:

1. **Memory Cell**: This cell stores values over arbitrary time intervals. The LSTM can read from, write to, and erase information from the cell, controlled by the gates.
2. **Input Gate**: Controls how much of the new information flows into the memory cell.
3. **Forget Gate**: Controls how much of the past information to forget.
4. **Output Gate**: Controls how much of the information from the memory cell is used to compute the output of the LSTM unit.

Each gate is a neural network layer with its weights, biases, and activation functions. They use the sigmoid function to output a value between 0 and 1, determining how much

information to pass through.

Use Cases of LSTMs:

LSTMs are widely used in various sequence prediction problems due to their ability to remember long-term dependencies. Some common use cases include:

- **Time Series Forecasting**: Predicting stock prices, weather conditions, and other time-dependent data.
- **Natural Language Processing (NLP)**: Language modeling, text generation, machine translation, and speech recognition.
- **Anomaly Detection**: Identifying unusual patterns in data, which is useful in fraud detection and system monitoring.
- **Video Analysis**: Understanding sequences of video frames for tasks such as activity recognition and video captioning.

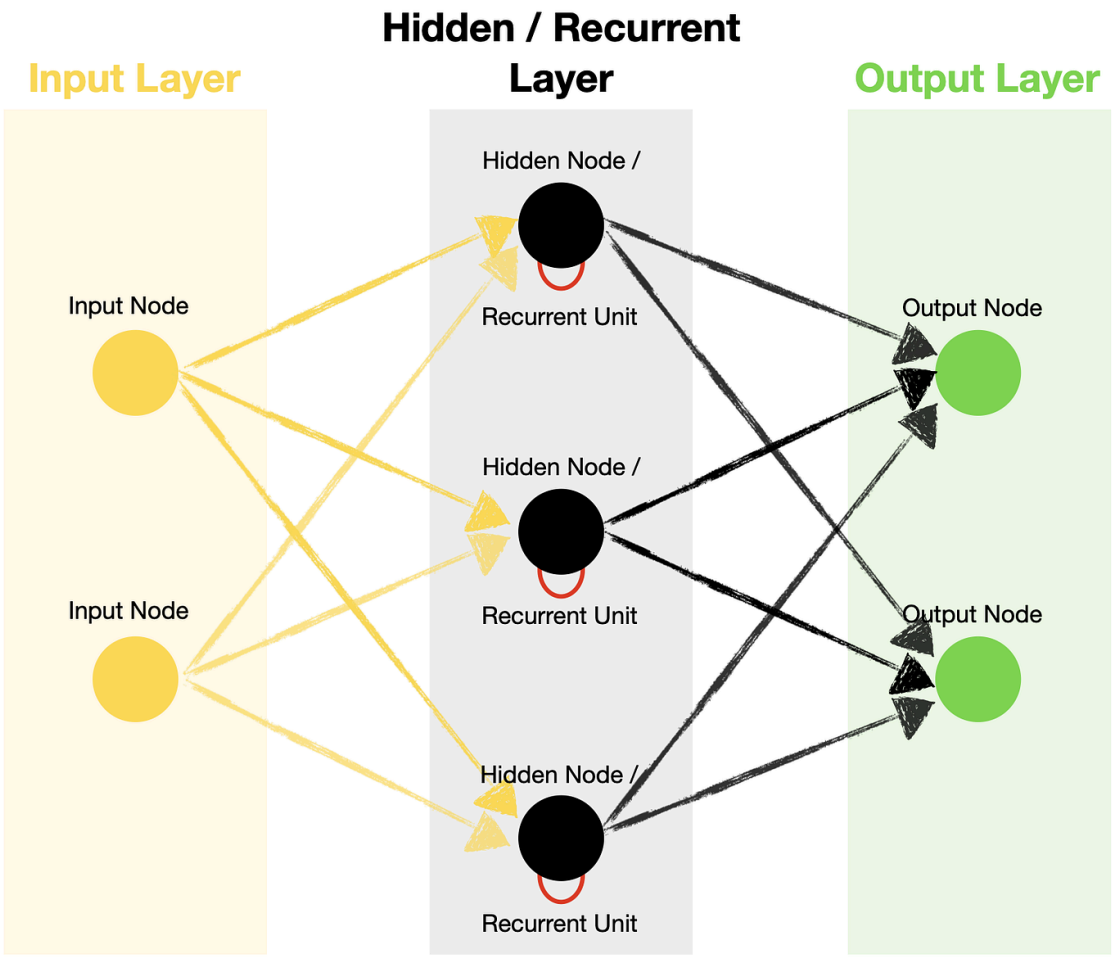
In [ ]:
```python
#Sample code for an LSTM model

class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(LSTMModel, self).__init__()
        self.hidden_size = hidden_size
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.linear = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h0 = torch.zeros(1, x.size(0), self.hidden_size)
        c0 = torch.zeros(1, x.size(0), self.hidden_size)
        out, _ = self.lstm(x, (h0, c0))
        out = self.linear(out[:, -1, :])
        return out

def train_lstm_model():
    # Generate synthetic data
    data = generate_data(100)
    data = [(torch.tensor(x, dtype=torch.float32).unsqueeze(0),
             torch.tensor(y, dtype=torch.float32).unsqueeze(0))
            for x, y in data]
    dataloader = torch.utils.data.DataLoader(data, batch_size=1, shuffle=True)

    # Model, loss, optimizer
    model = LSTMModel(1, 50, 1)
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    # Training loop
    for epoch in range(100):
        for seq, target in dataloader:
            optimizer.zero_grad()
            output = model(seq)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()

    print('LSTM Model Training Completed')

    # Testing the model
    test_seq = torch.tensor(np.sin(np.linspace(0, 100, 100)), dtype=torch.float32).
    with torch.no_grad():
```

```
        test_output = model(test_seq)
    print(f'LSTM Predicted next value: {test_output.item():.4f}')
```

In [7]:
```
train_lstm_model()
```

```
LSTM Model Training Completed
LSTM Predicted next value: -0.6323
```

GATED RECURRENT UNITS



Gated Recurrent Units (GRUs): GRUs are a type of recurrent neural network (RNN) architecture introduced by Cho et al. in 2014. They are similar to LSTMs but with a simplified structure, making them computationally more efficient.

How GRUs Work:

GRUs use two gates (reset gate and update gate) to control the flow of information:

1. **Reset Gate**: Determines how much of the previous hidden state to forget.
2. **Update Gate**: Controls how much of the new information to store in the current hidden state.

Unlike LSTMs, GRUs do not have a separate memory cell; instead, they directly operate on the hidden state. This simplicity can lead to faster training and inference times.

Use Cases of GRUs:

GRUs are used in similar applications as LSTMs, particularly when computational efficiency is important. Some common use cases include:

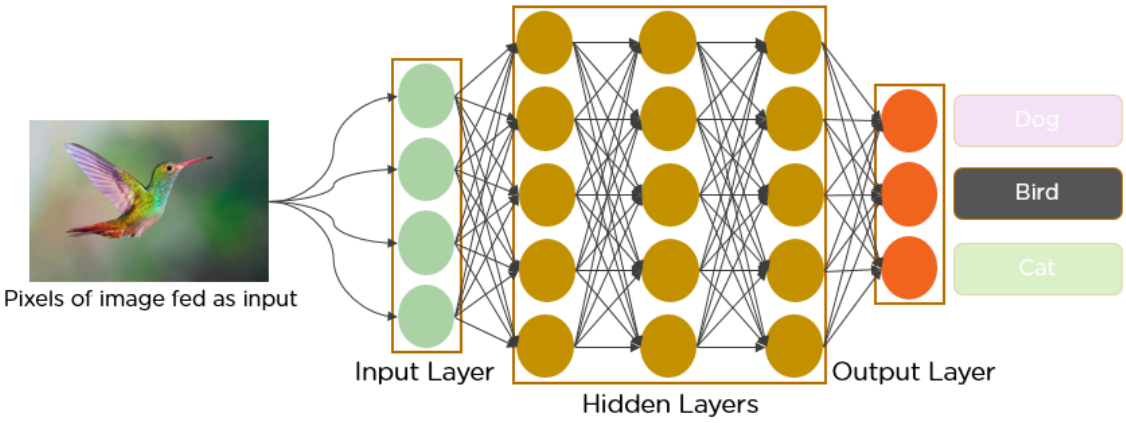
- **Time Series Prediction**: Forecasting future values in sequences such as stock prices and weather data.
- **Natural Language Processing (NLP)**: Language modeling, text generation, machine translation, and speech recognition.
- **Anomaly Detection**: Identifying unusual patterns in data for fraud detection and system monitoring.
- **Video Analysis**: Understanding sequences of video frames for tasks such as activity recognition and video captioning.

In [8]:
```python
#Sample code for an GRU  model
class GRUModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(GRUModel, self).__init__()
        self.hidden_size = hidden_size
        self.gru = nn.GRU(input_size, hidden_size, batch_first=True)
        self.linear = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h0 = torch.zeros(1, x.size(0), self.hidden_size)
        out, _ = self.gru(x, h0)
        out = self.linear(out[:, -1, :])
        return out

def train_gru_model():
    # Generate synthetic data
    data = generate_data(100)
    data = [(torch.tensor(x, dtype=torch.float32).unsqueeze(0),
             torch.tensor(y, dtype=torch.float32).unsqueeze(0))
            for x, y in data]
    dataloader = torch.utils.data.DataLoader(data, batch_size=1, shuffle=True)

    # Model, loss, optimizer
    model = GRUModel(1, 50, 1)
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    # Training loop
    for epoch in range(100):
        for seq, target in dataloader:
            optimizer.zero_grad()
            output = model(seq)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()

    print('GRU Model Training Completed')

    # Testing the model
    test_seq = torch.tensor(np.sin(np.linspace(0, 100, 100)), dtype=torch.float32).
    with torch.no_grad():
        test_output = model(test_seq)
    print(f'GRU Predicted next value: {test_output.item():.4f}')
```

In [9]:
```python
train_gru_model()
```

```
GRU Model Training Completed
GRU Predicted next value: -0.5642
```

CONVOLUTIONAL NEURAL NETWORKS



CNNs are a type of neural network architecture designed for processing structured grid data, such as images. They are particularly effective for tasks involving spatial hierarchies and local dependencies.

How CNNs Work:

CNNs consist of multiple layers, including convolutional layers, pooling layers, and fully connected layers:

1. **Conv2D Block**: A convolutional layer that applies a set of learnable filters to the input image. Each filter slides over the image, performing element-wise multiplications and summing the results to produce a feature map. This operation captures local patterns such as edges, textures, and shapes.
2. **MaxPooling**: A pooling layer that reduces the spatial dimensions of the feature maps, retaining the most important information. MaxPooling typically takes the maximum value from a small window (e.g., 2x2) and helps in reducing the computational complexity and controlling overfitting.
3. **Fully Connected Layers**: Layers where each neuron is connected to every neuron in the previous layer. These layers are typically used at the end of the network to perform classification or regression based on the features extracted by the convolutional and pooling layers.

Example of Use Cases:

CNNs are widely used in various image and video processing tasks. Some common use cases include:

- **Image Classification**: Recognizing objects and scenes in images, such as identifying animals, vehicles, or faces.
- **Object Detection**: Detecting and localizing objects within an image, used in applications like autonomous driving and surveillance.
- **Image Segmentation**: Partitioning an image into regions or objects, used in medical imaging and autonomous driving.
- **Video Analysis**: Understanding sequences of video frames for tasks such as activity recognition, video captioning, and video classification.

In [10]:
```python
#Sample code snippet using torchvision dataset
class CNNModel(nn.Module):
    def __init__(self):
        super(CNNModel, self).__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(32 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(-1, 32 * 7 * 7)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

def train_cnn_model():
    # Use MNIST dataset for image classification
    from torchvision import datasets, transforms

    transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.
    trainset = datasets.MNIST(root='./data', train=True, download=True, transform=t
    trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True

    # Model, loss, optimizer
    model = CNNModel()
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    # Training loop
    for epoch in range(5):
        running_loss = 0.0
        for images, labels in trainloader:
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        print(f'Epoch [{epoch+1}/5], Loss: {running_loss/len(trainloader):.4f}')

    print('CNN Model Training Completed')
```

REFERENCES AND ADDITIONAL RESOURCES

Daniel Bourke Youtube :https://www.youtube.com/watch?v=Z_ikDlimN6A

Zero To Mastery ; https://www.learnpytorch.io/

Pytorch Documentation ;https://pytorch.org/docs/stable/index.html

LSTM ;

https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/lstm#:~:text=LSTMs%20Long%20Short%2DTerm%20Memory,series%2C%20text%2C%2(

https://www.analyticsvidhya.com/blog/2021/03/introduction-to-long-short-term-memory-lstm/

https://www.geeksforgeeks.org/deep-learning-introduction-to-long-short-term-memory/

https://colah.github.io/posts/2015-08-Understanding-LSTMs/

RNNS ;

- https://aws.amazon.com/what-is/recurrent-neural-network/#:~:text=A%20recurrent%20neural%20network%20(RNN,a%20specific%20sequent
- https://www.geeksforgeeks.org/introduction-to-recurrent-neural-network/
- https://towardsdatascience.com/recurrent-neural-networks-rnns-3f06d7653a85

GRUS ;

- https://en.wikipedia.org/wiki/Gated_recurrent_unit
- https://towardsdatascience.com/understanding-rnns-lstms-and-grus-ed62eb584d90
- https://medium.com/@harshedabdulla/understanding-gated-recurrent-units-grus-in-deep-learning-4404599dcefb

CNNS ;

- https://www.ibm.com/topics/convolutional-neural-networks
- https://www.youtube.com/watch?v=NmLK_WQBxB4 - https://insightsimaging.springeropen.com/articles/10.1007/s13244-018-0639-9

Check out my github for some great solved problems ; https://github.com/muyale/Deep-Learning-Projects-

Check out this github for Mr Daniel Bourke's repo ; https://github.com/mrdbourke/pytorch-deep-learning

For Questions and assistance feel free to reachme out on my ;

- email : edgarmuyale@gmail.com
- Linked in ; https://www.linkedin.com/in/edgar-muyale-502a56248/
- Github : https://github.com/muyale

Some Image References; https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.geeksforgeeks.org%2Fintroduction-to-recurrent-neural-network%2F&psig=AOvVaw2Rrdgem-VVjmgyCBE1Bvwj&ust=1717135372292000&source=images&cd=vfe&opi=89978449&ved=0CE
https://www.google.com/url?sa=i&url=https%3A%2F%2Fdatabasecamp.de%2Fen%2Fml%2Flstms&psig=AOvVaw3vfCrQbQgI

Happy Coding !