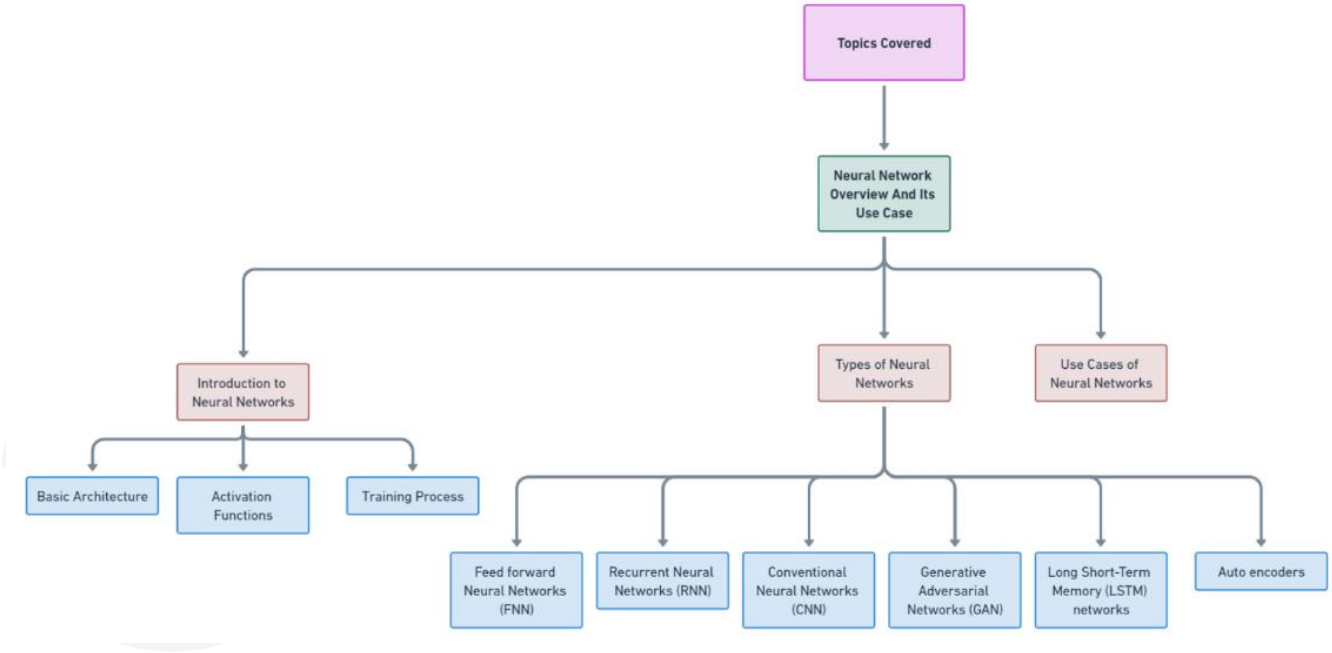
PW SKILLS

Lesson Plan

Neural Network Overview And Its Use Case

Topics Covered

- Introduction to Neural Networks
  - Basic Architecture
  - Activation Functions
  - Training Process
- Types of Neural Networks
  - Feedforward Neural Networks (FNN)
  - Recurrent Neural Networks (RNN)
  - Convolutional Neural Networks (CNN)
  - Generative Adversarial Networks (GAN)
  - Long Short-Term Memory (LSTM) networks
  - Autoencoders
- Use Cases of Neural Networks



Introduction to Neural Networks

- To understand what deep learning is, we first need to understand the relationship deep learning has with machine learning, neural networks, and artificial intelligence. The best way to think of this relationship is to visualize them as conA Neural Network is a computational model inspired by the structure and functioning of the human brain's interconnected neurons.
- It consists of interconnected nodes, commonly referred to as neurons or perceptrons, organized in layers.
- Each neuron processes input data by performing computations and propagating information forward through the network. These networks are capable of learning and recognizing patterns from data through a process of adjusting the connections (weights) between neurons.
- Neural networks are widely used in machine learning for tasks such as pattern recognition, classification, regression, and other complex data-driven problems due to their ability to learn representations from raw data.centric circles.

Basic Architecture

- Input Layer: Receives input data.
- Hidden Layers: Intermediate layers between the input and output layers where computations occur.
- Output Layer: Produces the final output.

Activation Functions

- Activation functions introduce non-linearities to the network, allowing it to learn complex patterns.

Examples: Sigmoid, ReLU, Tanh.

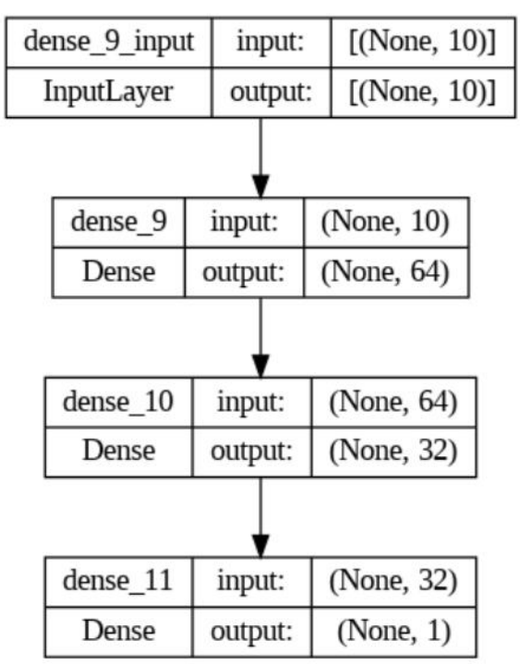
Training Process

- Forward Propagation: Input data is fed forward through the network to produce an output prediction.
- Calculate Loss: Compare the predicted output with the actual output to compute the loss.
- Backpropagation: Adjust weights and biases backward through the network to minimize the loss using optimization algorithms like Gradient Descent.
- Update Weights: Update the weights and biases to improve the network's performance.
- Iterate: Repeat the process until the model converges or reaches a satisfactory level of accuracy.

Code to see basic NN architecture

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import plot_model

# Define the model architecture
model = Sequential([
    Dense(64, activation='relu', input_shape=(10,)),  # Input
layer with 10 features
    Dense(32, activation='relu'),  # Hidden layer
    Dense(1, activation='sigmoid')  # Output layer
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

# Visualize the model architecture and save the image
plot_model(model, to_file='simple_neural_network.png',
show_shapes=True, show_layer_names=True)
```

Output:



Types Of Neural Networks

1. Feedforward Neural Network(FNN)

- Feedforward Neural Networks, also known as multilayer perceptrons (MLPs), are the simplest form of artificial neural networks.
- They are called "feedforward" because the information flows through the network in one direction, from input nodes through hidden nodes (if any) to the output nodes without any feedback loops.

Components of FNN

- Input Layer: The input layer consists of input nodes, each representing a feature or input variable. These nodes transmit the input data to the next layer.
- Hidden Layers: These layers sit between the input and output layers and perform transformations on the input data. Each hidden layer consists of numerous neurons (nodes), and the number of hidden layers and neurons per layer varies based on the network's architecture.

- Output Layer: The output layer produces the final output of the network based on the transformations carried out by the hidden layers. The number of nodes in this layer depends on the nature of the problem (e.g., regression, classification).

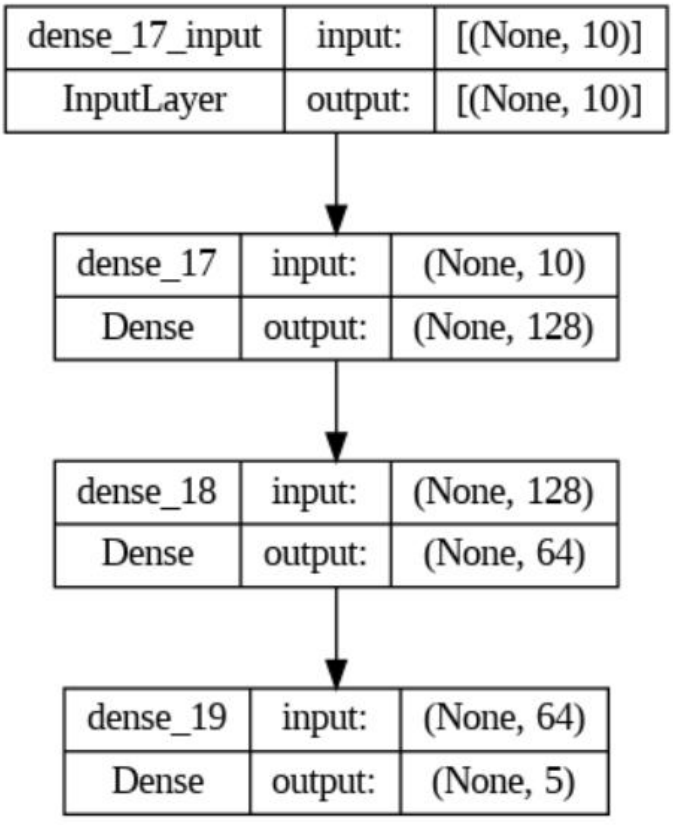
Activation Functions:

- Each neuron in an FNN applies an activation function to the weighted sum of its inputs, introducing non-linearities that allow the network to model complex relationships within the data. Common activation functions include:
  - Sigmoid: Squashes the output between 0 and 1.
  - ReLU (Rectified Linear Unit): Returns 0 for negative inputs and the input for positive inputs.
  - Tanh: Squashes the output between -1 and 1.
  - Softmax: Typically used in the output layer for multi-class classification problems, providing probabilities for each class.

Code To represent FNN

```
from keras.models import Sequential
from keras.layers import Dense
from tensorflow.keras.utils import plot_model

# Input shape and output size
input_shape = 10  # Number of input features
output_size = 5  # Example: Number of classes in a classification
problem

# Creating a simple feedforward neural network
model = Sequential()

model.add(Dense(128, input_shape=(input_shape,),
activation='relu'))  # Input shape specified
model.add(Dense(64, activation='relu'))
model.add(Dense(output_size, activation='softmax'))  # Output
size specified

# Compiling the model
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])

# Visualizing the model architecture
plot_model(model, to_file='fnn_model.png', show_shapes=True,
show_layer_names=True)
```

Output:



2. Recurrent Neural Network(RNN)

- Recurrent Neural Networks (RNNs) are a type of neural network designed to process sequential data where previous steps in the sequence contribute to the understanding of the current step.
- They are highly effective in handling sequence data due to their ability to maintain internal memory.

Sequential Data Handling:

- RNNs are tailored to handle sequential data like time series, text, audio, etc., where each input is not independent but influenced by previous inputs.
- For instance, in natural language processing (NLP), understanding the meaning of a word often requires context from previous words in a sentence.

Recurrent Connections:

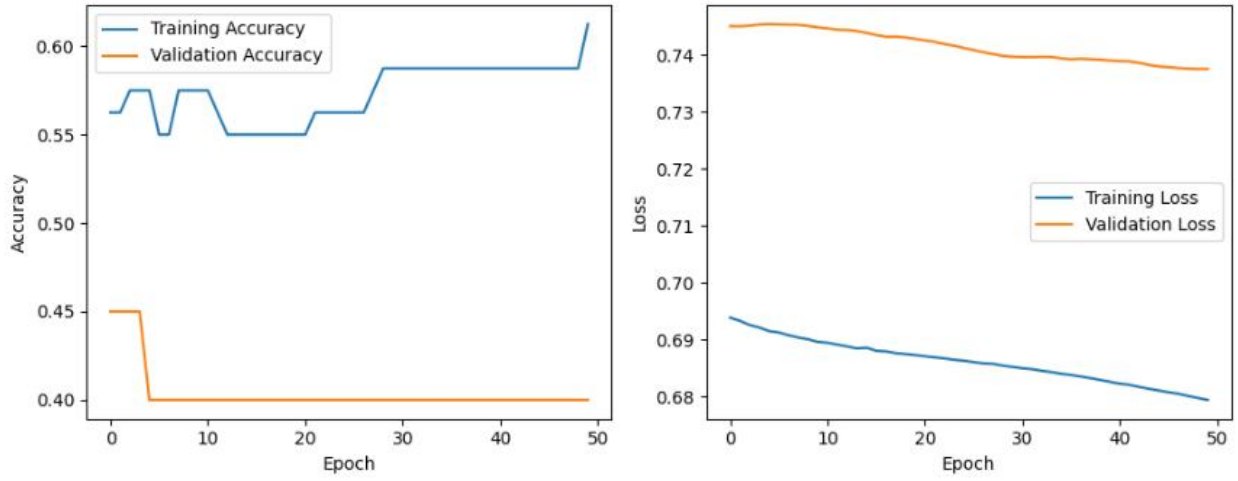
- Unlike feedforward neural networks where information flows in one direction, RNNs have connections that loop back to previous steps, allowing information persistence.
- This looping mechanism enables RNNs to maintain a form of memory or state, retaining information about previous steps in the sequence.

Code:

```
# Build RNN model
model = Sequential()
model.add(SimpleRNN(units=10, input_shape=(seq_length, 1),
activation='relu'))
model.add(Dense(units=1, activation='sigmoid'))
```

```
plot_model(model, to_file='rnn_model_plot.png', show_shapes=True,
show_layer_names=True)

# Train the RNN
history = model.fit(X, y, epochs=50, batch_size=32,
validation_split=0.2)

# Plotting accuracy and loss during training
plt.figure(figsize=(10, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation
Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()
```

Output:



3. Convolutional Neural Networks (CNN)

- Convolutional Neural Networks (CNNs) are specialized deep neural networks designed for processing structured grid-like data, primarily used for visual imagery but also for sequential data.
- CNNs are characterized by their unique architecture that employs convolutional layers, pooling layers, and fully connected layers.

Key Components of CNN

- **Convolutional Layers:**
  - Convolutional layers apply a series of filters (kernels) to the input data to extract features through convolutions.
  - Each filter captures different features of the input by sliding through the input's width and height.
  - Feature maps are generated by these convolutions, capturing spatial hierarchies of features.

- **Pooling (or Pooling Layers):**
  - Pooling layers reduce the spatial dimensions of the convolved features while retaining the most important information.
  - Common pooling operations include Max Pooling and Average Pooling, which downsample the feature maps by selecting the maximum or average value in each region, respectively.

- **Activation Functions:**
  - Activation functions like ReLU (Rectified Linear Unit) are often used after convolutional and fully connected layers to introduce non-linearity and enable the network to learn complex patterns.

- **Fully Connected Layers:**
  - Following multiple convolutional and pooling layers, fully connected layers are utilized for high-level reasoning on the extracted features.
  - These layers connect every neuron in one layer to every neuron in the next layer, capturing global relationships between features.

CNN Workflow

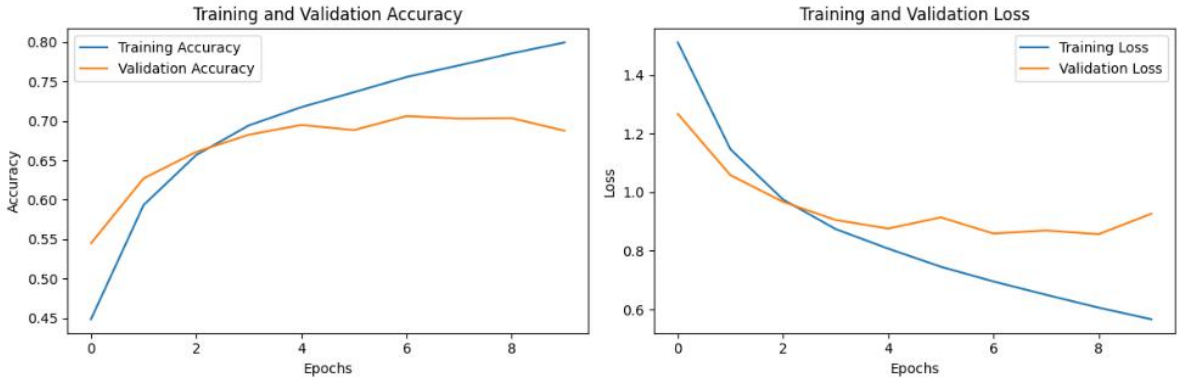
- Input Layer: Accepts input data, typically images but can also be used for sequential data like time-series.
- Convolutional Layers: Extract various features through the application of filters. Each layer can learn different levels of abstraction.
- Pooling Layers: Reduce spatial dimensions while retaining important information, making the network less sensitive to small variations.
- Flattening: Reshaping the output from the previous layers into a 1D array to be fed into fully connected layers.
- Fully Connected Layers: Perform high-level reasoning on the learned features to classify or predict based on the input.
- Output Layer: Provides the final prediction or classification.

Code:

```
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D,
Flatten, Dense
import matplotlib.pyplot as plt
import numpy as np

# Load CIFAR-10 dataset
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
'dog', 'frog', 'horse', 'ship', 'truck']

# Normalize pixel values between 0 and 1
X_train = X_train / 255.0
X_test = X_test / 255.0
```

```
# Build CNN model
model = Sequential()

model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(32,
32, 3)))
model.add(MaxPooling2D((2, 2)))

model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))

model.add(Conv2D(64, (3, 3), activation='relu'))

model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dense(10, activation='softmax'))

model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Train the model
history = model.fit(X_train, y_train, epochs=10,
validation_data=(X_test, y_test))

# Plot training and validation accuracy and loss
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation
Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')

plt.legend()

plt.tight_layout()
plt.show()

# Visualize some predictions
predictions = model.predict(X_test)
num_images = 5

plt.figure(figsize=(15, 3))
for i in range(num_images):
    plt.subplot(1, num_images, i+1)
    plt.imshow(X_test[i])
    plt.title(f'Predicted:
{class_names[np.argmax(predictions[i])]} \n Actual:
{class_names[y_test[i][0]]}')
    plt.axis('off')

plt.tight_layout()
plt.show()

# Visualize the CNN architecture
from tensorflow.keras.utils import plot_model
plot_model(model, to_file='cnn_model.png', show_shapes=True,
show_layer_names=True)
```

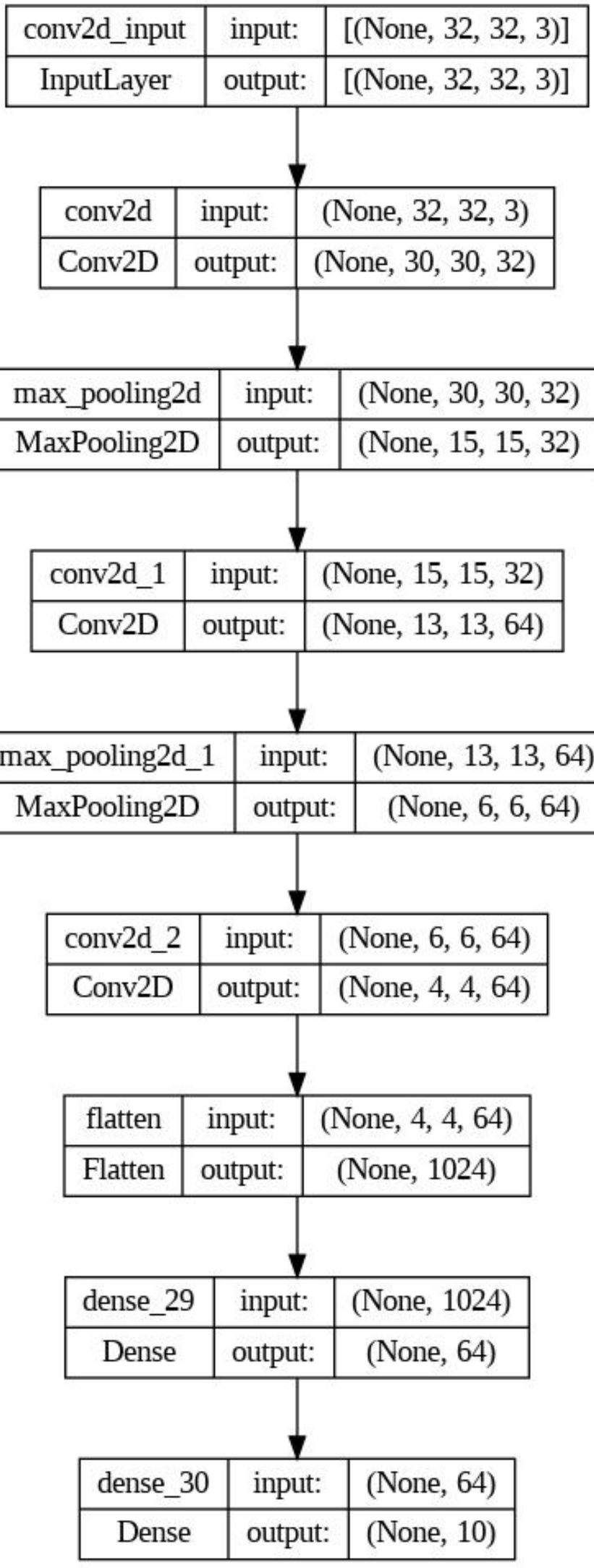
Output:



Visual Representation

```
# Visualize some predictions
predictions = model.predict(X_test)
num_images = 5

plt.figure(figsize=(15, 3))
for i in range(num_images):
    plt.subplot(1, num_images, i+1)
    plt.imshow(X_test[i])
    plt.title(f'Predicted:
{class_names[np.argmax(predictions[i])]} \n Actual:
{class_names[y_test[i][0]]}')
    plt.axis('off')

plt.tight_layout()
plt.show()

# Visualize the CNN architecture
from tensorflow.keras.utils import plot_model
plot_model(model, to_file='cnn_model.png', show_shapes=True,
show_layer_names=True)
```

Output:


conv2d_input | input: | [(None, 32, 32, 3)]
InputLayer | output: | [(None, 32, 32, 3)]
conv2d | input: | (None, 32, 32, 3)
Conv2D | output: | (None, 30, 30, 32)
max_pooling2d | input: | (None, 30, 30, 32)
MaxPooling2D | output: | (None, 15, 15, 32)
conv2d_1 | input: | (None, 15, 15, 32)
Conv2D | output: | (None, 13, 13, 64)
max_pooling2d_1 | input: | (None, 13, 13, 64)
MaxPooling2D | output: | (None, 6, 6, 64)
conv2d_2 | input: | (None, 6, 6, 64)
Conv2D | output: | (None, 4, 4, 64)
flatten | input: | (None, 4, 4, 64)
Flatten | output: | (None, 1024)
dense_29 | input: | (None, 1024)
Dense | output: | (None, 64)
dense_30 | input: | (None, 64)
Dense | output: | (None, 10)

4. Generative Adversarial Networks (GAN)

- Generative Adversarial Networks (GANs) are a class of artificial intelligence algorithms introduced by Ian Goodfellow and his colleagues in 2014.
- GANs consist of two neural networks, a generator and a discriminator, that work in tandem and compete against each other in a zero-sum game framework.

a) Generator Network:

- The generator aims to create realistic data samples by learning from a random noise vector or latent space. It generates fake data resembling the input dataset.
- It typically uses deconvolutional or upsampling layers to transform the input noise into data that resembles the original dataset.
- The generator attempts to generate data that is indistinguishable from the real data to fool the discriminator.

b) Discriminator Network:

- The discriminator evaluates the authenticity of the generated data by distinguishing between real and fake samples.
- It learns to classify whether the input data comes from the real dataset or is generated by the generator.
- The discriminator is trained on both real and generated samples to differentiate between them accurately.

Training Process:

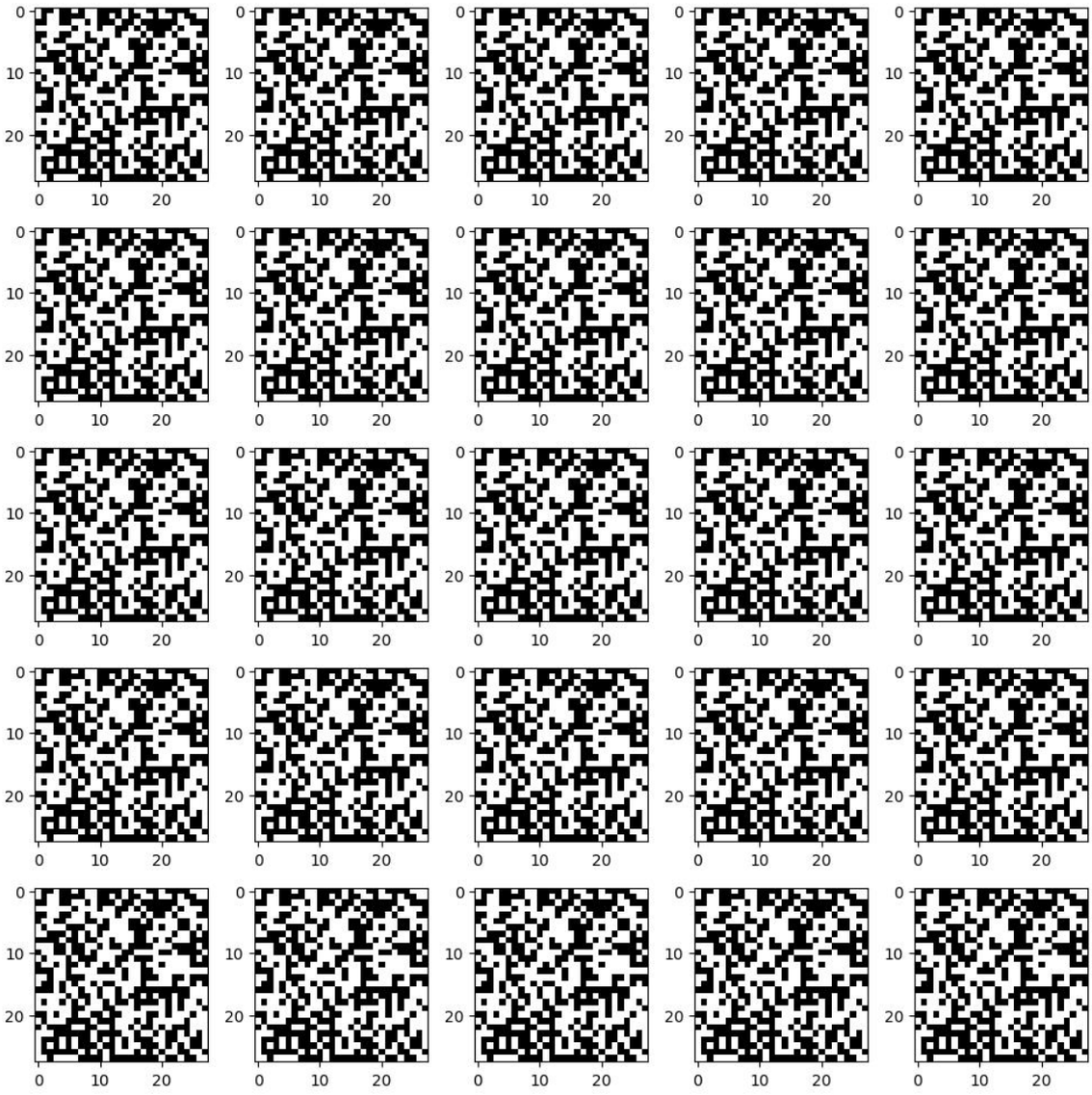
- During training, the generator and discriminator networks are updated alternately.
- The discriminator is trained on both real and fake data, optimizing its ability to differentiate between the two.
- The generator is trained to generate data that increasingly fools the discriminator, updating its weights to improve the realism of the generated samples.
- This adversarial process continues until a balance is reached, ideally producing high-quality generated data.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
from tensorflow.keras.layers import Dense, Reshape, Flatten
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam

# Load MNIST dataset
(X_train, _), (_, _) = mnist.load_data()
X_train = (X_train.astype(np.float32) - 127.5) / 127.5  #
Normalize data [-1, 1]
X_train = X_train.reshape(X_train.shape[0], 784)

# Generator
generator = Sequential()
generator.add(Dense(256, input_dim=100, activation='relu'))
generator.add(Dense(512, activation='relu'))
generator.add(Dense(1024, activation='relu'))
generator.add(Dense(784, activation='tanh'))
generator.compile(loss='binary_crossentropy',
optimizer=Adam(lr=0.0002, beta_1=0.5))

# Discriminator
discriminator = Sequential()
discriminator.add(Dense(1024, input_dim=784, activation='relu'))
discriminator.add(Dense(512, activation='relu'))
discriminator.add(Dense(256, activation='relu'))
discriminator.add(Dense(1, activation='sigmoid'))
discriminator.compile(loss='binary_crossentropy',
optimizer=Adam(lr=0.0002, beta_1=0.5), metrics=['accuracy'])
```

```
gan.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.0002,
beta_1=0.5))

# Training the GAN
batch_size = 128
epochs = 20000

for epoch in range(epochs):
    # Generate random noise as input to the generator
    noise = np.random.normal(0, 1, size=(batch_size, 100))

    # Generate fake images from the noise
    fake_images = generator.predict(noise)

    # Get a random set of real images
    idx = np.random.randint(0, X_train.shape[0], batch_size)
    real_images = X_train[idx]

    # Concatenate real and fake images into a single array
    combined_images = np.concatenate([real_images, fake_images])

    # Labels for generated and real data
    labels = np.concatenate([np.ones((batch_size, 1)),
np.zeros((batch_size, 1))])
    labels += 0.05 * np.random.random(labels.shape)  # Add random
noise to labels

    # Train discriminator
    d_loss = discriminator.train_on_batch(combined_images,
labels)

    # Generate new random noise for the generator
    noise = np.random.normal(0, 1, size=(batch_size, 100))

    # Labels for generated data
    misleading_labels = np.zeros((batch_size, 1))

    # Train generator (via the combined model, with frozen
discriminator weights)
    g_loss = gan.train_on_batch(noise, misleading_labels)

# Print progress
    if epoch % 1000 == 0:
        print(f"Epoch: {epoch} - Discriminator Loss: {d_loss[0]},
Generator Loss: {g_loss}")

# Generating and plotting generated images
noise = np.random.normal(0, 1, size=(25, 100))
generated_images = generator.predict(noise)

plt.figure(figsize=(10, 10))
for i in range(25):
    plt.subplot(5, 5, i+1)
    plt.imshow(generated_images[i].reshape(28, 28), cmap='gray')
    plt.axis('off')
plt.tight_layout()
plt.show()
```

Output:



Note: This is an overview of the GAN. if interested Explore or more detail understanding.

5. Long Short-Term Memory (LSTM) networks

- Long Short-Term Memory (LSTM) networks are a type of recurrent neural network (RNN) architecture specifically designed to overcome the limitations of traditional RNNs in capturing long-term dependencies in sequential data.
- LSTMs have gained immense popularity in various fields, especially in natural language processing, time series prediction, and speech recognition, due to their ability to handle sequences and remember information over long periods.

Advantages of LSTMs

- Long-Term Dependencies: LSTMs excel in capturing long-range dependencies in sequential data due to their ability to retain and propagate information through time steps.
- Avoiding Vanishing Gradient: The architecture's design helps in mitigating the vanishing gradient problem, allowing for more stable and effective training.

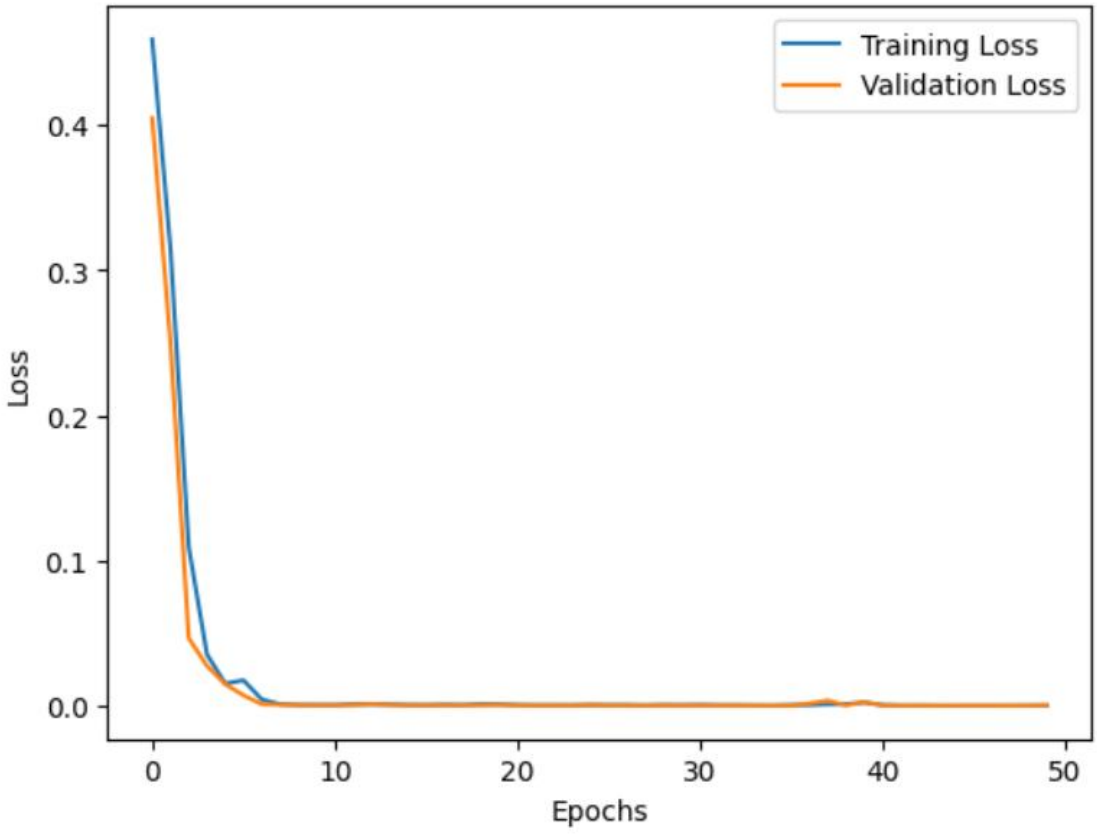
Applications of LSTMs:

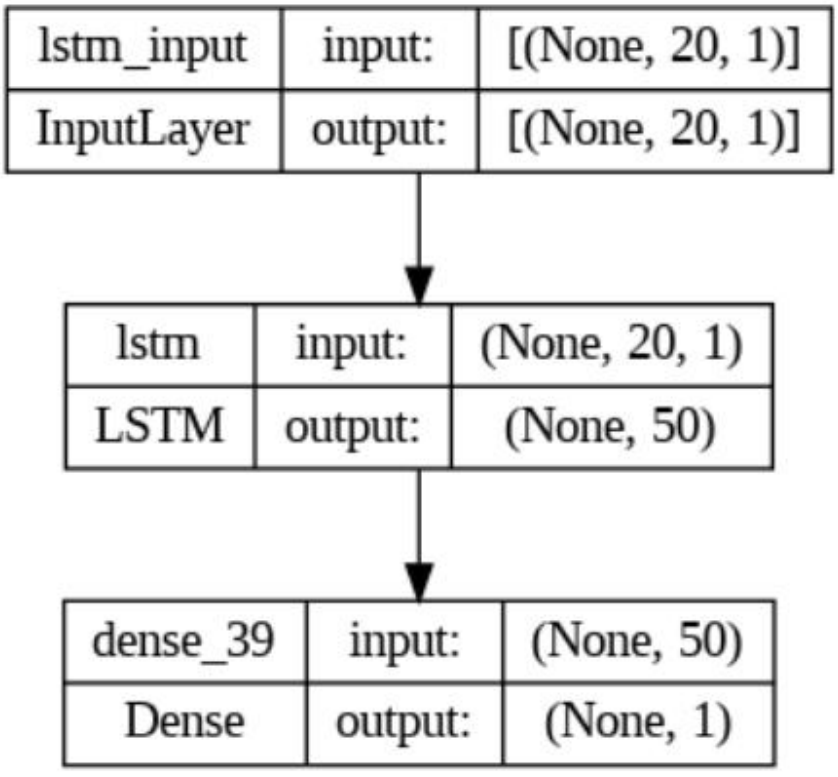
- Natural Language Processing: Language modeling, text generation, machine translation.
- Time Series Prediction: Stock market forecasting, weather prediction.
- Speech Recognition: Phoneme recognition, speech-to-text systems.
- Healthcare: ECG signal analysis, patient monitoring.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from tensorflow.keras.utils import plot_model

# Generate some sample sequential data (sine wave)
def generate_sequence(length, period):
    return [np.sin(2 * np.pi * i / period) for i in
range(length)]

# Generate input and output sequences for LSTM
def generate_data(n_samples=1000, n_time_steps=20):
    data = []
    for _ in range(n_samples):
        period = np.random.randint(10, 20)
        sequence = generate_sequence(n_time_steps + 1, period)
        data.append(sequence)
    data = np.array(data)
    X, y = data[:, :-1], data[:, -1]
    X = X.reshape(X.shape[0], X.shape[1], 1)  # Reshape for LSTM
input
    return X, y
# Create LSTM model
model = Sequential()
model.add(LSTM(50, activation='relu', input_shape=(20, 1)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')

# Generate data
X, y = generate_data()

# Train LSTM model
history = model.fit(X, y, epochs=50, batch_size=32,
validation_split=0.2)

# Plot training loss and validation loss
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Visualize LSTM model architecture
plot_model(model, to_file='lstm_model.png', show_shapes=True,
show_layer_names=True)
```

Output:


Training Loss
Validation Loss
Loss
Epochs
0.0
0.1
0.2
0.3
0.4
0
10
20
30
40
50


lstm_input
input:
[(None, 20, 1)]
InputLayer
output:
[(None, 20, 1)]
lstm
input:
(None, 20, 1)
LSTM
output:
(None, 50)
dense_39
input:
(None, 50)
Dense
output:
(None, 1)

6. Autoencoders

- Autoencoders are neural networks designed to copy their input to the output, typically through a bottleneck layer that represents a compressed representation of the input data.
- They aim to learn efficient representations of data, capturing the most important aspects to reconstruct the input accurately.

Structure

- **Encoder:** Takes the input and maps it into a latent space representation (often of lower dimensionality than the input). This compression layer captures the essential features of the input data.
- **Decoder:** Reconstructs the data from the compressed representation back to the original input dimensions.
- **Bottleneck/Hidden Layer:** The layer between the encoder and decoder, containing the compressed representation.

Types of Autoencoders

- **Vanilla Autoencoder:** Basic architecture with symmetric encoder and decoder.
- **Denoising Autoencoder:** Trained to reconstruct undistorted data from corrupted input, forcing the network to learn robust representations.
- **Sparse Autoencoder:** Introduces sparsity constraints to the hidden layer to learn more useful and compact representations.
- **Variational Autoencoder (VAE):** A probabilistic approach that generates new data points by learning the underlying probability distribution of the data in the latent space.
- **Convolutional Autoencoder:** Utilizes convolutional layers for applications in image data, maintaining spatial relationships.

Applications

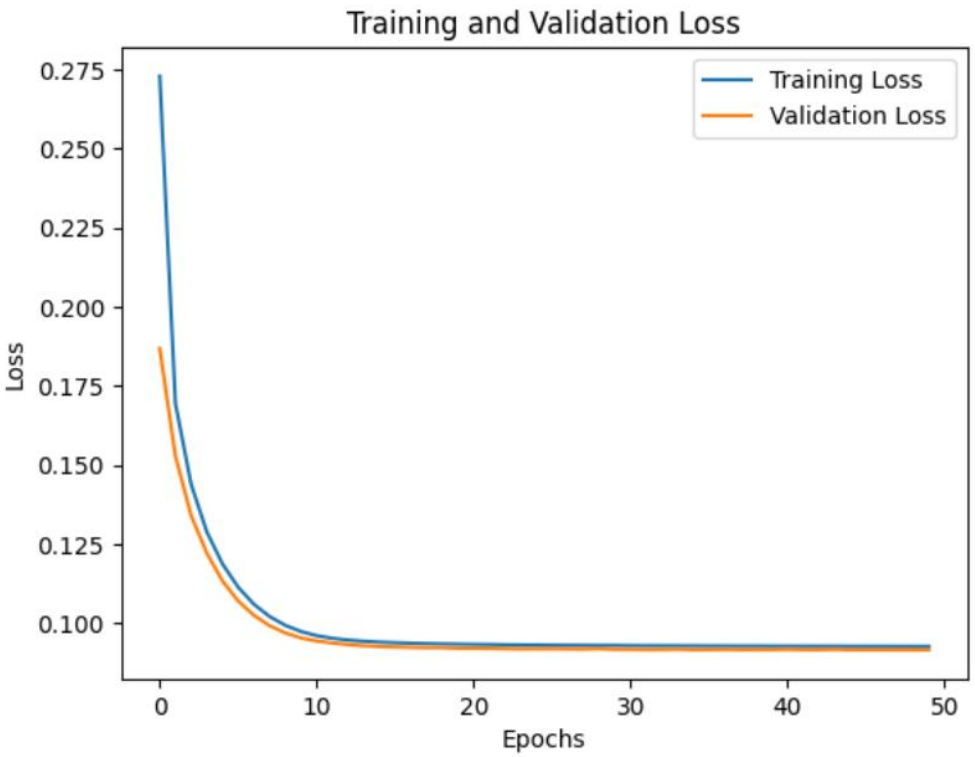
- Dimensionality Reduction: Creating compact representations of high-dimensional data.
- Feature Learning: Extracting meaningful features from data for subsequent tasks like classification.
- Data Denoising: Removing noise from corrupted data.
- Anomaly Detection: Identifying outliers by high reconstruction error.

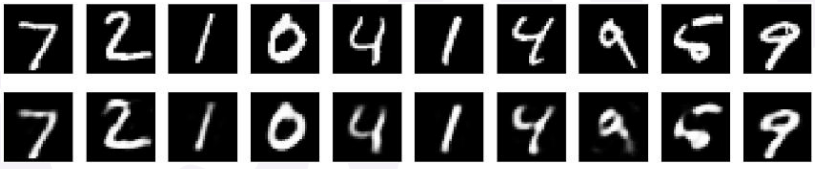
Code:

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model

# Load MNIST dataset
(X_train, _), (X_test, _) = mnist.load_data()

# Normalize and reshape data
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0
X_train = X_train.reshape((len(X_train),
np.prod(X_train.shape[1:])))
X_test = X_test.reshape((len(X_test), np.prod(X_test.shape[1:])))
```

```
epochs = 50
batch_size = 256
history = autoencoder.fit(X_train, X_train, epochs=epochs,
batch_size=batch_size, shuffle=True, validation_data=(X_test,
X_test))

# Visualize model structure
from tensorflow.keras.utils import plot_model
plot_model(autoencoder, to_file='autoencoder.png',
show_shapes=True, show_layer_names=True)

# Plot training/validation loss
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs_range = range(epochs)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Training and Validation Loss')
plt.savefig('loss_plot.png')
plt.show()

# Reconstructed images visualization
decoded_imgs = autoencoder.predict(X_test)

n = 10  # How many digits to display
plt.figure(figsize=(20, 4))
for i in range(n):
    # Display original images
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(X_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Display reconstructed images
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.savefig('reconstructed_images.png')
plt.show()
```

Output:





Use Cases of Neural Networks

1. **Image Recognition and Classification**
   - Application: Identifying objects or patterns within images.
   - Example: ImageNet competition winners using CNN for image recognition.

2. **Natural Language Processing (NLP)**
   - Application: Processing and understanding human language data.
   - Example: Sentiment analysis, machine translation, chatbots powered by RNNs or transformers.

3. **Recommender Systems**
   - Application: Suggesting items or content to users based on their preferences.
   - Example: Netflix's recommendation engine using collaborative filtering and deep learning.

4. **Time Series Prediction**
   - Application: Forecasting future events or trends based on historical data.
   - Example: Stock market prediction, weather forecasting using RNNs or LSTM networks.

5. **Anomaly Detection**
   - Application: Identifying unusual patterns or outliers in data.
   - Example: Fraud detection in financial transactions, network intrusion detection.

6. **Robotics and Control Systems**
   - Application: Enabling robots to perceive and interact with their environment.
   - Example: Autonomous vehicles, robotic arms controlled by neural networks.

7. **Medical Diagnosis and Healthcare**
   - Application: Assisting in disease diagnosis and healthcare management.
   - Example: Diagnosing diseases from medical images (X-rays, MRIs), personalized medicine.

8. **Audio and Speech Recognition**
   - Application: Transcribing spoken words into text or recognizing voice commands.
   - Example: Speech-to-text systems, virtual assistants like Siri or Alexa.

9. **Financial Forecasting and Trading**
   - Application: Predicting market trends and making financial decisions.
   - Example: Algorithmic trading, predicting stock prices using neural networks.

10. **Predictive Maintenance in Manufacturing**
    - Application: Anticipating equipment failures and optimizing maintenance schedules.
    - Example: Predicting machinery breakdowns based on sensor data in manufacturing plants.

11. **Biometrics and Security**
    - Application: Authentication and identification based on unique biological characteristics.
    - Examples: Facial recognition, and fingerprint recognition systems.

12. **Environmental Analysis and Conservation**
    - Application: Monitoring and managing environmental data for conservation efforts.
    - Example: Species identification from images or audio recordings, climate modeling.