# Neural network a simple perception

## Theoritical questions :-

Ques.1 :- What is deep learning and how is it connected with artificial intelligence ?

Ans :- Deep Learning: A Powerful Subset of Artificial Intelligence

Deep learning is a subset of machine learning that utilizes artificial neural networks to mimic the human brain's learning process. It's a powerful technique that allows computers to learn complex patterns from large datasets.

How it's Connected to Artificial Intelligence:

Artificial intelligence (AI) is a broader field that encompasses various techniques to make machines intelligent. Deep learning is one of the key techniques driving advancements in AI. Here's how they're interconnected:

Neural Networks: Both deep learning and AI utilize neural networks, but deep learning takes it a step further by employing multiple layers of these networks. These layers allow for more complex pattern recognition and decision-making. Learning from Data: Both deep learning and AI rely on learning from data. However, deep learning models can learn directly from raw data, such as images or text, without explicit feature engineering.

Ques.2 :- What is a neural network,and what are the different types of neural networks ?

Ans :- Neural Networks: Mimicking the Human Brain

A neural network is a computational system inspired by the biological structure of the human brain. It's composed of interconnected nodes, or neurons, that process information. These neurons are organized into layers, and information flows from the input layer to the output layer through hidden layers.

Types of Neural Networks

There are various types of neural networks, each designed for specific tasks. Here are some of the most common ones:

Feedforward Neural Networks:

Simplest type of neural network. Information flows in one direction, from input to output, without forming cycles. Used for tasks like classification and regression. Recurrent Neural Networks (RNNs):

Designed to process sequential data, like time series or natural language. Utilize feedback loops, allowing information to persist over time. Examples: Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) networks. Convolutional Neural Networks (CNNs):

Primarily used for image and video analysis. Employ convolutional layers to extract features from input data. Excellent for tasks like image classification, object detection, and image segmentation. Generative Adversarial Networks (GANs):

Composed of two neural networks, a generator and a discriminator. The generator creates new data instances, while the discriminator evaluates their authenticity. Used for tasks like image generation, style transfer, and data augmentation.

Ques.3 :- What is the mathmatical structure of a neural network ?

Ans :- Mathematical Structure of a Neural Network

At the core of a neural network lies a mathematical function that maps input data to output predictions. This function is composed of several layers of interconnected nodes, each performing a simple calculation.

Basic Unit: The Neuron

A single neuron, or node, in a neural network can be represented mathematically as follows:

output = activation_function(weighted_sum + bias) Where:

weighted_sum: This is the dot product of the input vector (x) and the weight vector (w), often denoted as w^T * x. bias: A constant value added to the weighted sum to shift the activation function. activation_function: A non-linear function that introduces non-linearity to the network, allowing it to learn complex patterns. Common activation functions include ReLU, sigmoid, and tanh. Layers in a Neural Network

A neural network typically consists of multiple layers:

Input Layer: Receives the raw input data. Hidden Layers: Process the input data through multiple layers of neurons, learning complex patterns. Output Layer: Produces the final output, such as a classification or regression prediction. Forward Propagation

During forward propagation, the input data is fed through the network, layer by layer. The output of one layer becomes the input to the next. The mathematical operation at each layer can be summarized as:

output_layer_i = activation_function(W_i * output_layer_(i-1) + b_i) Where:

W_i: Weight matrix for the ith layer b_i: Bias vector for the ith layer Backpropagation

Backpropagation is an algorithm used to train neural networks. It involves calculating the error between the predicted output and the actual target output, and then adjusting the weights and biases in each layer to minimize this error.

The core idea is to compute the gradient of the loss function with respect to the weights and biases, and then update the weights and biases using gradient descent:

weight_new = weight_old - learning_rate * gradient

Ques.4 :- what is an activation function,and why is it essential in neural?

Ans :- Activation Functions: Introducing Non-Linearity

An activation function is a mathematical function that introduces non-linearity into a neural network. Without activation functions, a neural network would be equivalent to a simple linear regression model, limiting its ability to learn complex patterns.

Why are Activation Functions Essential?

Non-Linearity: Real-world data is inherently non-linear. Activation functions enable neural networks to model complex relationships between inputs and outputs that linear models cannot capture. Decision-Making: Activation functions introduce decision-making capabilities. They help determine whether a neuron should "fire" or not based on the input it receives. Feature Learning: By introducing non-linear transformations, activation functions allow neural networks to learn hierarchical representations of data, extracting meaningful features from raw input. Common Activation Functions:

Sigmoid: Squashes input values between 0 and 1. Used in older neural networks but less common nowadays due to vanishing gradient problems. Tanh: Similar to sigmoid but outputs values between -1 and 1. Can be more effective than sigmoid in certain scenarios. ReLU (Rectified Linear Unit): Outputs the input directly if it's positive, otherwise, it outputs 0. Widely used due to its simplicity and efficiency. Leaky ReLU: A variant of ReLU that outputs a small positive value for negative inputs. Helps mitigate the "dying ReLU" problem. ELU (Exponential Linear Unit): Combines the advantages of ReLU and Leaky ReLU. Can be more effective in deeper networks. Choosing the Right Activation Function:

The choice of activation function depends on the specific task and network architecture. Some factors to consider include:

Vanishing Gradient Problem: Sigmoid and tanh can suffer from this issue, especially in deep networks. Sparsity: ReLU and its variants can introduce sparsity, which can be beneficial for feature learning. Derivative: The derivative of the activation function is used in backpropagation. A smooth derivative can help with optimization.

Ques.5 Could you list some common activation functions used in neural networks?

Ans :- Here are some common activation functions used in neural networks:

Classic Activation Functions:

Sigmoid: Squashes input values between 0 and 1. Tanh (Hyperbolic Tangent): Squashes input values between -1 and 1. Modern Activation Functions:

ReLU (Rectified Linear Unit): Outputs the input directly if it's positive, otherwise, it outputs 0. Leaky ReLU: A variation of ReLU that outputs a small positive value for negative inputs. ELU (Exponential Linear Unit): Combines the advantages of ReLU and Leaky ReLU. Softmax: Often used in the output layer for multi-class classification, producing a probability distribution over the classes.

Ques.6 What is multilayer neural network?

Ans :- A multilayer neural network, also known as a Multilayer Perceptron (MLP), is a type of artificial neural network with multiple layers of interconnected nodes. It's a fundamental building block in deep learning and is used for various tasks like classification, regression, and pattern recognition.

Key Components of a Multilayer Neural Network:

Input Layer: Receives the raw input data. Hidden Layers: Process the input data through multiple layers of neurons, learning complex patterns. Output Layer: Produces the final output, such as a classification or regression prediction. How it Works:

Forward Propagation:

Input data is fed into the input layer. Each neuron in a layer calculates a weighted sum of its inputs and passes it through an activation function. The output of one layer becomes the input to the next layer. This process continues until the output layer produces the final prediction. Backpropagation:

The network's output is compared to the desired output, and an error is calculated. The error is propagated backward through the network, layer by layer. The weights and biases of each neuron are adjusted to minimize the error. This process is repeated iteratively until the network learns to make accurate predictions. Advantages of Multilayer Neural Networks:

Ability to learn complex patterns: By using multiple layers and non-linear activation functions, MLPs can model highly complex relationships in data. Versatility: They can be applied to a wide range of tasks, from image and speech recognition to natural language processing. Scalability: They can be scaled to handle large datasets and complex problems.

Ques.7 what is a loss function,and why is it crucial for neural network training?

Ans :- A loss function is a mathematical function that measures the discrepancy between the predicted output of a neural network and the actual target value. It quantifies the error made by the model.

Why is it Crucial?

The primary goal of training a neural network is to minimize this loss function. By doing so, the network learns to make more accurate predictions.

Common Loss Functions:

Mean Squared Error (MSE):

Commonly used for regression tasks. Calculates the average squared difference between predicted and actual values. Suitable when large errors should be penalized more heavily. Mean Absolute Error (MAE):

Also used for regression tasks. Calculates the average absolute difference between predicted and actual values. Less sensitive to outliers compared to MSE. Binary Cross-Entropy Loss:

Used for binary classification tasks (e.g., predicting whether an email is spam or not). Measures the dissimilarity between two probability distributions: the predicted probability and the true label. Categorical Cross-Entropy Loss:

Used for multi-class classification tasks (e.g., classifying images into different categories). Measures the dissimilarity between the predicted probability distribution and the true one-hot encoded label. The Training Process:

Forward Pass: Input data is fed into the network, and the output is calculated. Loss Calculation: The loss function is used to calculate the error between the predicted output and the true label. Backpropagation: The error is propagated backward through the network, and the weights and biases are adjusted to minimize the loss. Optimization: An optimization algorithm, like gradient descent, is used to update the weights and biases. By iteratively minimizing the loss function, the neural network learns to make more accurate predictions.

Ques.8 What are some common types of loss functions?

Ans :- Here are some common types of loss functions used in neural networks:

Regression Loss Functions:

Mean Squared Error (MSE): Calculates the average squared difference between predicted and actual values. Mean Absolute Error (MAE): Calculates the average absolute difference between predicted and actual values. Huber Loss: Combines the best of both MSE and MAE, being less sensitive to outliers than MSE while still providing a differentiable loss function. Classification Loss Functions:

Binary Cross-Entropy Loss: Used for binary classification problems (e.g., spam detection). Categorical Cross-Entropy Loss: Used for multi-class classification problems (e.g., image classification). Sparse Categorical Cross-Entropy Loss: A variant of categorical cross-entropy loss used when dealing with integer class labels.

Ques.9 How does a neural network learn?

Ans :- Neural networks learn through a process called backpropagation. Here's a simplified explanation:

Forward Pass:

Input data is fed into the network. Each neuron in a layer calculates a weighted sum of its inputs and passes it through an activation function. The output of one layer becomes the input to the next layer. This process continues until the output layer produces a prediction. Loss Calculation:

The network's output is compared to the desired output, and a loss function is used to calculate the error. Backpropagation:

The error is propagated backward through the network, layer by layer. The weights and biases of each neuron are adjusted to minimize the error. This process is repeated iteratively until the network learns to make accurate predictions

Ques.10 What is an optimizer in neural networks,and why is it necessary?

Ans :- An optimizer is an algorithm or method used to adjust the parameters (weights and biases) of a neural network during the training process. Its primary goal is to minimize the loss function, which measures the difference between the network's predicted output and the actual target output.

Why is it Necessary?

Efficient Learning: Optimizers guide the learning process by determining the optimal step size (learning rate) and direction for adjusting the parameters. Convergence: They help the network converge to a minimum of the loss function, ensuring that the model learns to make accurate

predictions. Avoiding Local Minima: Effective optimizers can help the network escape local minima, which are suboptimal solutions that can hinder learning. Common Optimizers:

Stochastic Gradient Descent (SGD):

Updates parameters using the gradient of the loss function calculated on a single training example. Simple but can be noisy and slow to converge. Mini-Batch Gradient Descent:

Updates parameters using the gradient of the loss function calculated on a small batch of training examples. Balances the efficiency of batch gradient descent and the robustness of stochastic gradient descent. Momentum:

Accumulates gradients over multiple iterations, helping to accelerate convergence and smooth out oscillations. Adam (Adaptive Moment Estimation):

Combines the advantages of momentum and RMSprop, adapting the learning rate for each parameter. A popular and effective optimizer for many deep learning tasks. RMSprop (Root Mean Square Propagation):

Adapts the learning rate for each parameter based on the running average of the squared gradients. Helps to handle noisy gradients and accelerate convergence. The choice of optimizer depends on factors such as the complexity of the model, the size of the dataset, and the desired level of accuracy. Experimentation is often necessary to find the best optimizer for a specific task.

Sources and related content

Ques.11 Could you breifly describe some common optimizers?

Ans :- Here are some common optimizers used in neural networks:

1. Stochastic Gradient Descent (SGD):

Updates parameters using the gradient of the loss function calculated on a single training example. Simple but can be noisy and slow to converge.

2. Mini-Batch Gradient Descent:

Updates parameters using the gradient of the loss function calculated on a small batch of training examples. Balances the efficiency of batch gradient descent and the robustness of stochastic gradient descent.

3. Momentum:

Accumulates gradients over multiple iterations, helping to accelerate convergence and smooth out oscillations.

4. Adam (Adaptive Moment Estimation):

Combines the advantages of momentum and RMSprop, adapting the learning rate for each parameter. A popular and effective optimizer for many deep learning tasks.

5. RMSprop (Root Mean Square Propagation):

Adapts the learning rate for each parameter based on the running average of the squared gradients. Helps to handle noisy gradients and accelerate convergence. The choice of optimizer depends on factors such as the complexity of the model, the size of the dataset, and the desired level of accuracy. Experimentation is often necessary to find the best optimizer for a specific task.

Ques.12 Can you explained forward and backword propagation in neural networks ?

Ans :-Forward and Backward Propagation in Neural Networks Forward Propagation In forward propagation, information moves from the input layer through the hidden layers to the output layer. Here's a breakdown:

Input Layer: Receives the raw input data. Hidden Layers: Each neuron receives weighted inputs from the previous layer. The weighted sum is passed through an activation function, introducing non-linearity. The output of each neuron becomes the input to the next layer. Output Layer: The final layer produces the output, which is compared to the actual target value. Backward Propagation Backward propagation is the process of adjusting the weights and biases in the network to minimize the error between the predicted output and the actual target. Here's how it works:

Error Calculation: The error between the predicted output and the actual target is calculated using a loss function. Gradient Calculation: The gradient of the loss function with respect to the weights and biases is calculated using the chain rule. Weight and Bias1 Update: The weights and biases are updated using an optimization algorithm like gradient descent. The updated weights and biases reduce the error in the next forward pass.

Ques.13 What is weight initialization,and how does it impact training?

Ans :- Weight initialization refers to the process of assigning initial values to the weights of a neural network before training begins. This initial state significantly impacts the network's ability to learn and converge to an optimal solution.

Why is Weight Initialization Important?

Vanishing Gradient Problem: If weights are too small, gradients can become very small during backpropagation, making it difficult for the network to learn. Exploding Gradient Problem: If weights are too large, gradients can become very large, leading to unstable training and divergence. Convergence Speed: Good initialization can accelerate the training process by providing a better starting point for optimization. Common Weight Initialization Techniques:

Random Initialization:

Weights are assigned random values from a specific distribution, such as uniform or normal distribution. While simple, it can lead to unstable training if not carefully tuned. Xavier Initialization:

Designed to keep the variance of activations constant across layers. Suitable for networks with sigmoid or tanh activation functions. He Initialization:

Specifically designed for ReLU activation functions. Aims to keep the variance of activations constant across layers. Orthogonal Initialization:

Initializes weights to orthogonal matrices. Can be useful for deep networks and recurrent neural networks. The Impact of Weight Initialization on Training:

Convergence Speed: Good initialization can lead to faster convergence and better performance. Stability: Proper initialization helps prevent vanishing or exploding gradients, ensuring stable training. Generalization: Well-initialized networks are more likely to generalize well to unseen data.

Ques.14 What is vanishing gradient problem in deep learning?

Ans :- The vanishing gradient problem is a challenge that arises during the training of deep neural networks. It occurs when the gradients used to update the network's weights become extremely small or "vanish" as they are backpropagated from the output layers to the earlier layers.

Why does it happen?

Chain Rule in Backpropagation: The gradients are calculated using the chain rule, which involves multiplying many small numbers (derivatives of activation functions). As the number of layers increases, this product becomes exponentially smaller, leading to vanishing gradients. Saturation of Activation Functions: Activation functions like sigmoid and tanh have a limited output range, and their derivatives tend to be small, especially near the saturation points. This further contributes to the vanishing gradient problem. Consequences of Vanishing Gradients:

Slow Learning: The weights of earlier layers receive very small updates, making the network learn slowly or not at all. Suboptimal Performance: The network may get stuck in a local minimum, unable to reach the global minimum of the loss function. Solutions to the Vanishing Gradient Problem:

ReLU Activation Function: ReLU is a popular choice as it introduces non-linearity without suffering from the saturation problem. Leaky ReLU: A variation of ReLU that allows a small gradient for negative inputs, helping to alleviate the vanishing gradient problem. ELU (Exponential Linear Unit): Combines the advantages of ReLU and Leaky ReLU. Batch Normalization: Normalizes the input to each layer, helping to stabilize the training process and reduce the vanishing gradient problem. Gradient Clipping: Limits the magnitude of gradients to prevent them from exploding. Careful Weight Initialization: Techniques like Xavier and He initialization can help mitigate the problem. Residual Networks: Skip connections allow gradients to flow directly to earlier layers, bypassing the vanishing gradient issue.

Ques.15 What is exploding gradient problem?

Ans :- The exploding gradient problem is another challenge that can arise during the training of deep neural networks. It occurs when the gradients used to update the network's weights become excessively large.

Why does it happen?

Chain Rule in Backpropagation: Similar to the vanishing gradient problem, the chain rule is used to calculate gradients. However, in this case, the product of many large numbers can lead to exploding gradients. Large Weight Initializations: If the initial weights are too large, the gradients can quickly grow exponentially during backpropagation. Consequences of Exploding Gradients:

Instability: Large weight updates can make the training process unstable, leading to divergence. NaN Values: In extreme cases, the weights can become so large that they result in NaN (Not a Number) values, halting the training process. Solutions to the Exploding Gradient Problem:

Gradient Clipping: A common technique to limit the magnitude of gradients. By clipping the gradients to a certain threshold, it prevents them from becoming too large. Careful Weight Initialization: Using appropriate initialization techniques like Xavier or He initialization can help mitigate the problem. Normalization Techniques: Techniques like batch normalization can help stabilize the training process and reduce the impact of exploding gradients. Choosing the Right Optimizer: Optimizers like Adam and RMSprop can help to adaptively adjust the learning rate, which can help to prevent exploding gradients.

## Practical questions :-

```
# Q.1 How do you create a simple perceptron for basic binary classification?

# Ans :-
import numpy as np

def perceptron(X, weights, bias):
  """
  Perceptron function.

  Args:
    X: Input features (numpy array).
    weights: Weights for each feature (numpy array).
    bias: Bias term.

  Returns:
    Output of the perceptron (0 or 1).
  """

  z = np.dot(X, weights) + bias
  return 1 if z > 0 else 0

# Example usage:
X = np.array([[1, 1], [0, 0], [1, 0], [0, 1]])  # Input features
y = np.array([1, 0, 0, 0])  # Target labels
weights = np.array([0.2, -0.1])
bias = 0.1

# Train the perceptron (simplified example)
learning_rate = 0.1
for epoch in range(10):
  for i in range(len(X)):
    prediction = perceptron(X[i], weights, bias)
    error = y[i] - prediction
    weights += learning_rate * error * X[i]
    bias += learning_rate * error


# Ques.2 How can you build a neural network with one hidden layer using keras?
# Ans :-
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(16, activation='relu', input_dim=784))  # Hidden layer with 16 neurons
model.add(Dense(10, activation='softmax'))  # Output layer for 10-class classification

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test, y_test))

test_loss, test_acc = model.evaluate(X_test, y_test)
print('Test accuracy:', test_acc)
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning:
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-5-f91befd0250b> in <cell line: 13>()
     11 # Compile the model
     12 model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=
['accuracy'])
---> 13 model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test,
y_test))
     14
     15 test_loss, test_acc = model.evaluate(X_test, y_test)

NameError: name 'X_train' is not defined
```

Next steps: Explain error

```
# Q.3 How do you initialize the weights using the Xavier(Glorot)initalization method in keras?
# Ans :-
from tensorflow.keras.layers import Dense
from tensorflow.keras.initializers import GlorotUniform

model = Sequential()
model.add(Dense(64, activation='relu', kernel_initializer='glorot_uniform', input_dim=784))
model.add(Dense(10, activation='softmax', kernel_initializer='glorot_uniform'))



# Q.4 How can you apply different activation functions in a neural network in keras?
# Ans :-
from tensorflow.keras.layers import Dense
from tensorflow.keras.activations import relu, sigmoid, tanh

model = Sequential()
model.add(Dense(64, activation='relu', input_dim=784))  # Using ReLU activation
model.add(Dense(32, activation=sigmoid))  # Using Sigmoid activation
model.add(Dense(10, activation='softmax'))  # Using Softmax activation


# Q.5  How do you add dropout  to a neural network model to prevent overfitting?
# Ans :-
from tensorflow.keras.layers import Dropout

model = Sequential()
model.add(Dense(64, activation='relu', input_dim=784))
model.add(Dropout(0.2))  # 20% dropout rate
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.2))  # 20% dropout rate
model.add(Dense(10, activation='softmax'))
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argumer
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
# Q.6 How do you manually implement forward propagation in a simple neural networks
# Ans :-
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Sample input data
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

# Initialize weights and biases randomly
weights1 = np.random.rand(2, 2)
biases1 = np.random.rand(2)
```

```
weights2 = np.random.rand(2, 1)
biases2 = np.random.rand(1)

# Forward propagation
def forward_propagation(X):
    hidden_layer_input = np.dot(X, weights1) + biases1
    hidden_layer_activation = sigmoid(hidden_layer_input)

    output_layer_input = np.dot(hidden_layer_activation, weights2) + biases2
    output = sigmoid(output_layer_input)

    return output

# Example usage
output = forward_propagation(X)
```

```
[[0.85554914]
 [0.87309669]
 [0.88338482]
 [0.89199008]]
```

```
# Q.7 How do you add batch normalization to a neural network model in keras?
# Ans :-
from tensorflow.keras.layers import BatchNormalization

model = Sequential()
model.add(Dense(64, activation='relu', input_dim=784))
model.add(BatchNormalization())
model.add(Dense(64, activation='relu'))
model.add(BatchNormalization())
model.add(Dense(10, activation='softmax'))
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argumer
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
# Q.8 How can you visualize the training process with accuracy and loss curves?
# Ans :-
history.history['accuracy']
history.history['val_accuracy']
history.history['loss']


import matplotlib.pyplot as plt

# Assuming you've trained your model and have a 'history' object
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.show()

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['train', 'val'], loc='upper right')
plt.show()
```

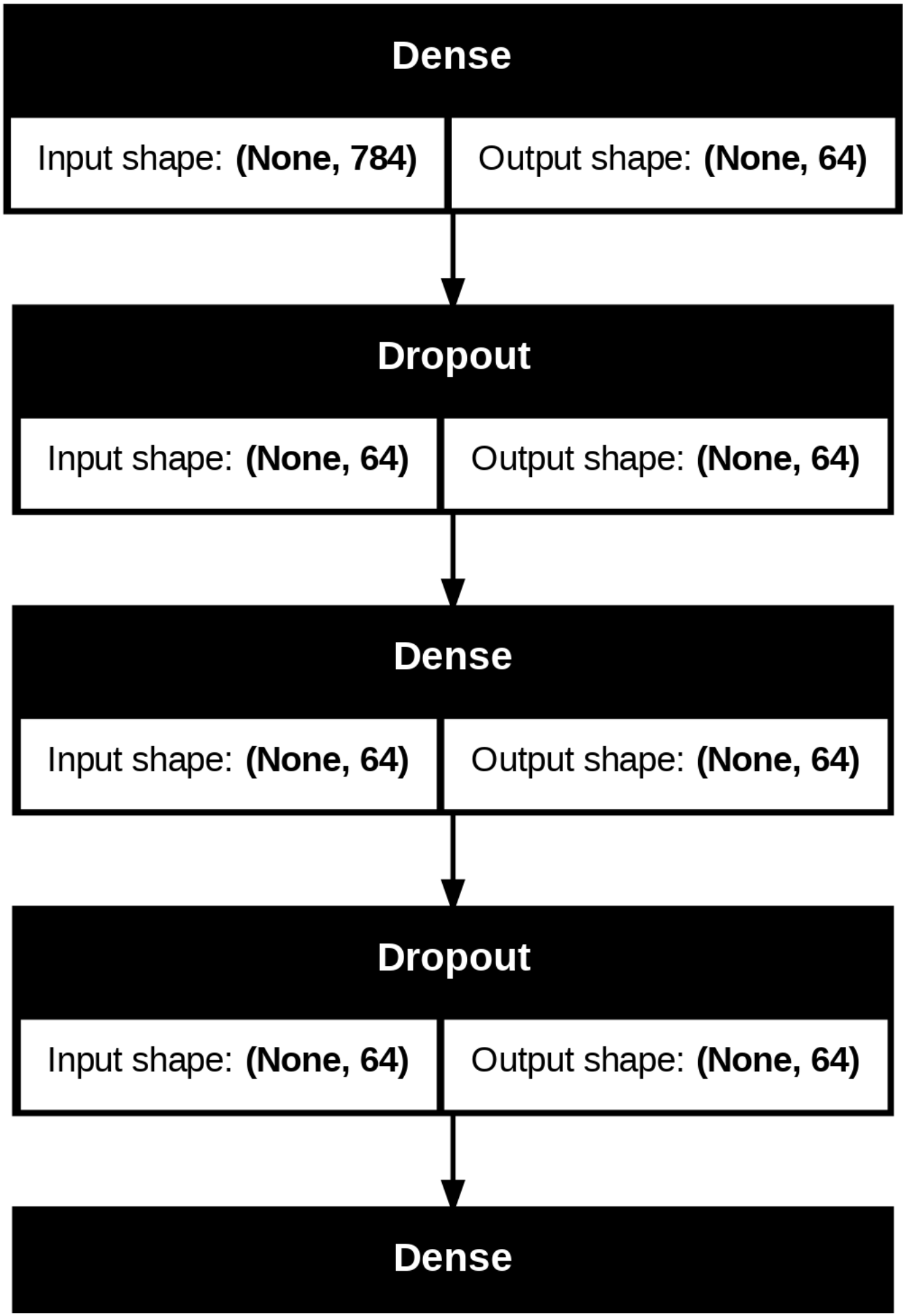
```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-17-858ac3e4b166> in <cell line: 3>()
      1 # Q.8 How can you visualize the training process with accuracy and loss curves?
      2 # Ans :-
----> 3 history.history['accuracy']
      4 history.history['val_accuracy']
      5 history.history['loss']

NameError: name 'history' is not defined
```

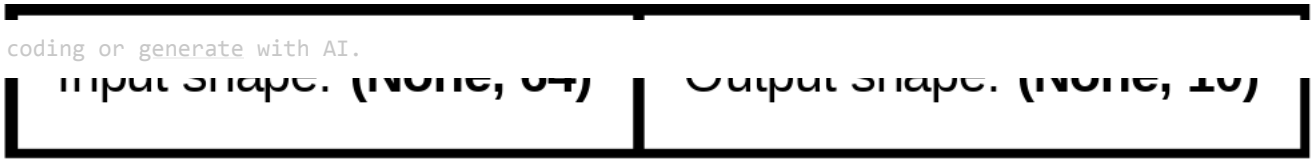
Next steps: Explain error

```
# Q.9 How can you use gradient clipping in keras to control the gradient size and prevent exploding gradients?
# Ans :-
from tensorflow.keras.optimizers import Adam

# Create an optimizer with gradient clipping
optimizer = Adam(learning_rate=0.001, clipvalue=0.5)

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

from tensorflow.keras.optimizers import Adam

# Create an optimizer with gradient clipping
optimizer = Adam(learning_rate=0.001, clipnorm=1.0)

# Compile the model with the optimizer
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])
```

```
# Q.10 How can you create a custom loss function in keras?
# Ans :-
import tensorflow as tf
from tensorflow.keras.losses import Loss

class CustomLoss(Loss):
    def __init__(self, name='custom_loss'):
        super().__init__(name=name)

    def call(self, y_true, y_pred):
        # Calculate the loss based on your desired formula
        loss = ...
        return loss

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

class MeanSquaredError(Loss):
    def call(self, y_true, y_pred):
        return tf.reduce_mean(tf.square(y_true - y_pred))
```

```
# Q.11 How can you visualize the structure of a neural networks model in keras?
# Ans :-
from tensorflow.keras.utils import plot_model

# Assuming you have a model 'model'
plot_model(model, to_file='model.png', show_shapes=True)
```


Dense
Input shape: (None, 784)
Output shape: (None, 64)
Dropout
Input shape: (None, 64)
Output shape: (None, 64)
Dense
Input shape: (None, 64)
Output shape: (None, 64)
Dropout
Input shape: (None, 64)
Output shape: (None, 64)
Dense




Start coding or generate with AI.