1.Implement a Single Layer Neural Network for OR gate, also verify with three inputs

```
import numpy as np
# Sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
# Derivative of the sigmoid function
def sigmoid_derivative(x):
    return x * (1 - x)
# Hyperparameters
learning_rate = 0.1
epochs = 1000
# OR Gate - Training Data (Three Inputs)
inputs= np.array([[0, 0, 0],
                     [0, 0, 1],
                     [0, 1, 0],
                     [0, 1, 1],
                     [1, 0, 0],
                     [1, 0, 1],
                     [1, 1, 0],
                     [1, 1, 1]])

outputs = np.array([[0], [1], [1], [1], [1], [1], [1], [1]])  # Expected OR gate output
# Initialize weights and bias for 3 inputs
weights = np.random.rand(3, 1)  # 3 inputs to 1 output
bias = np.random.rand(1)
# Training the network for 3-input OR gate
for epoch in range(epochs):
    # Forward pass
    weighted_sum = np.dot(inputs, weights) + bias
    predictions = sigmoid(weighted_sum)
    # Compute error
    error = outputs - predictions
    # Backpropagation
    adjustments = error * sigmoid_derivative(predictions)
    weights += np.dot(inputs.T, adjustments) * learning_rate
    bias += np.sum(adjustments) * learning_rate
# Print final weights and bias after training
print("Final weights:")
print(weights)
print("\nFinal bias:")
print(bias)
# Testing the trained model on OR gate with 3 inputs
```

```
print("\nTesting OR Gate (3 inputs):")
for input_data in inputs:
    result = sigmoid(np.dot(input_data, weights) + bias)
    print(f"Input: {input_data}, Output: {round(result[0])}")
```

Show hidden output

2.Implement a Single Layer Neural Network for AND gate, also verify with three inputs just give me the code that is enough

```
import numpy as np
# Sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
# Derivative of the sigmoid function
def sigmoid_derivative(x):
    return x * (1 - x)
# Initialize weights and bias
np.random.seed(42)
learning_rate = 0.1
# Training the network
epochs = 10000
# AND Gate - Training Data (Three Inputs)
inputs = np.array([[0, 0, 0],
                   [0, 0, 1],
                   [0, 1, 0],
                   [0, 1, 1],
                   [1, 0, 0],
                   [1, 0, 1],
                   [1, 1, 0],
                   [1, 1, 1]])
outputs = np.array([[0], [0], [0], [0], [0], [0], [0], [1]])  # Expected AND gate output
# Initialize weights and bias for 3 inputs
weights = np.random.rand(3, 1)
bias = np.random.rand(1)
# Training the network for 3-input OR gate
for epoch in range(epochs):
    # Forward pass
    weighted_sum = np.dot(inputs, weights) + bias
    predictions = sigmoid(weighted_sum)
    # Compute error
    error = outputs - predictions
    # Backpropagation
```

```
        adjustments = error * sigmoid_derivative(predictions)
        weights += np.dot(inputs.T, adjustments) * learning_rate
        bias += np.sum(adjustments) * learning_rate
# Print final weights and bias after training
print("Final weights:")
print(weights)
print("\nFinal bias:")
print(bias)
# Testing the trained model on AND gate with 3 inputs
print("\nTesting AND Gate (3 inputs):")
for input_data in inputs:
    result = sigmoid(np.dot(input_data, weights) + bias)
    print(f"Input: {input_data}, Output: {round(result[0])}")
```

Show hidden output

3.Implement a Neural Network with Hidden Layer for XOR gate, also verify with three inputs

```
import numpy as np
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def sigmoid_derivative(x):
    return x * (1 - x)
np.random.seed(42)
learning_rate = 0.1
epochs = 1000
inputs_3 = np.array([[0, 0, 0],
                     [0, 0, 1],
                     [0, 1, 0],
                     [0, 1, 1],
                     [1, 0, 0],
                     [1, 0, 1],
                     [1, 1, 0],
                     [1, 1, 1]])
outputs_3 = np.array([[0], [1], [1], [0], [1], [0], [0], [1]])  # Expected XOR gate output
weights_input_hidden = np.random.rand(3, 3)
weights_hidden_output = np.random.rand(3, 1)
bias_hidden = np.random.rand(3)
bias_output = np.random.rand(1)
# Training the network for 3-input XOR gate
for epoch in range(epochs):
    # Forward pass
```

```
        hidden_layer_input = np.dot(inputs_3, weights_input_hidden) + bias_hidden
        hidden_layer_output = sigmoid(hidden_layer_input)
        final_input = np.dot(hidden_layer_output, weights_hidden_output) + bias_output
        final_output = sigmoid(final_input)
        # Compute error
        error = outputs_3 - final_output
        # Backpropagation
        output_adjustments = error * sigmoid_derivative(final_output)
        hidden_error = np.dot(output_adjustments, weights_hidden_output.T)
        hidden_adjustments = hidden_error * sigmoid_derivative(hidden_layer_output)
        # Update weights and biases
        weights_hidden_output += np.dot(hidden_layer_output.T, output_adjustments) * learning_rate
        bias_output += np.sum(output_adjustments, axis=0) * learning_rate
        weights_input_hidden += np.dot(inputs_3.T, hidden_adjustments) * learning_rate
        bias_hidden += np.sum(hidden_adjustments, axis=0) * learning_rate
    # Print final weights and bias after training
    print("Final weights (Input to Hidden):")
    print(weights_input_hidden)
    print("\nFinal weights (Hidden to Output):")
    print(weights_hidden_output)
    print("\nFinal bias (Hidden Layer):")
    print(bias_hidden)
    print("\nFinal bias (Output Layer):")
    print(bias_output)
    # Testing the trained model on XOR gate with 3 inputs
    print("\nTesting XOR Gate (3 inputs):")
    for input_data in inputs_3:
        hidden_layer_input = np.dot(input_data, weights_input_hidden) + bias_hidden
        hidden_layer_output = sigmoid(hidden_layer_input)
        final_input = np.dot(hidden_layer_output, weights_hidden_output) + bias_output
        result = sigmoid(final_input)
        print(f"Input: {input_data}, Output: {round(result[0])}")
```

Show hidden output

4.Implement a Neural Network with Hidden Layer for NAND gate, with three inputs

```
import numpy as np
# Sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
# Derivative of the sigmoid function
def sigmoid_derivative(x):
    return x * (1 - x)
```

```
    # Initialize weights and bias
    np.random.seed(42)
    learning_rate = 0.1
    # Training the network
    epochs = 1000
    # NAND Gate - Training Data (Three Inputs)
    inputs = np.array([[0, 0, 0],
                       [0, 0, 1],
                       [0, 1, 0],
                       [0, 1, 1],
                       [1, 0, 0],
                       [1, 0, 1],
                       [1, 1, 0],
                       [1, 1, 1]])
    outputs = np.array([[1], [1], [1], [1], [1], [1], [1], [0]])  # Expected NAND gate output
    # Initialize weights and bias for 3-input NAND gate
    input_size = 3
    hidden_size = 3
    output_size = 1
    weights_input_hidden = np.random.rand(input_size, hidden_size)
    weights_hidden_output = np.random.rand(hidden_size, output_size)
    bias_hidden = np.random.rand(hidden_size)
    bias_output = np.random.rand(output_size)
    # Training the network for 3-input NAND gate
    for epoch in range(epochs):
        # Forward pass
        hidden_layer_input = np.dot(inputs, weights_input_hidden) + bias_hidden
        hidden_layer_output = sigmoid(hidden_layer_input)
        final_input = np.dot(hidden_layer_output, weights_hidden_output) + bias_output
        final_output = sigmoid(final_input)
        # Compute error
        error = outputs - final_output
        # Backpropagation
        output_adjustments = error * sigmoid_derivative(final_output)
        hidden_error = np.dot(output_adjustments, weights_hidden_output.T)
        hidden_adjustments = hidden_error * sigmoid_derivative(hidden_layer_output)
        # Update weights and biases
        weights_hidden_output += np.dot(hidden_layer_output.T, output_adjustments) * learning_rate
        bias_output += np.sum(output_adjustments, axis=0) * learning_rate
        weights_input_hidden += np.dot(inputs.T, hidden_adjustments) * learning_rate
        bias_hidden += np.sum(hidden_adjustments, axis=0) * learning_rate
    # Print final weights and bias after training
    print("Final weights (Input to Hidden):")
    print(weights_input_hidden)
    print("\nFinal weights (Hidden to Output):")
    print(weights_hidden_output)
    print("\nFinal bias (Hidden Layer):")
```

```
print(bias_hidden)
print("\nFinal bias (Output Layer):")
print(bias_output)
# Testing the trained model on NAND gate with 3 inputs
print("\nTesting NAND Gate (3 inputs):")
for input_data in inputs:
    hidden_layer_input = np.dot(input_data, weights_input_hidden) + bias_hidden
    hidden_layer_output = sigmoid(hidden_layer_input)
    final_input = np.dot(hidden_layer_output, weights_hidden_output) + bias_output
    result = sigmoid(final_input)
    print(f"Input: {input_data}, Output: {round(result[0])}")
```

Show hidden output

5. Implement Train a Neural Network for Multiclass Models for the data set available at MNIST data (available at https://keras.io/2.17/api/datasets/mnist/)

```
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt

# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Normalize pixel values to the range [0, 1]
x_train, x_test = x_train / 255.0, x_test / 255.0

# Flatten the 28x28 images into 1D vectors of size 784
x_train = x_train.reshape(-1, 784)
x_test = x_test.reshape(-1, 784)

# Convert labels to one-hot encoding
y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)

# Define the neural network model
model = keras.Sequential([
    keras.layers.Dense(128, activation='relu', input_shape=(784,)),  # Hidden layer 1
    keras.layers.Dense(64, activation='relu'),  # Hidden layer 2
    keras.layers.Dense(10, activation='softmax')  # Output layer (10 classes)
])
```

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

# Train the model
history = model.fit(x_train, y_train, epochs=10, batch_size=32, validation_data=(x_test, y_test))

# Evaluate the model on the test set
test_loss, test_accuracy = model.evaluate(x_test, y_test, verbose=0)
print(f"\nTest Accuracy: {test_accuracy:.4f}")

# Predict a few test images
predictions = model.predict(x_test[:5])

# Display results
plt.figure(figsize=(10, 5))
for i in range(5):
    plt.subplot(1, 5, i + 1)
    plt.imshow(x_test[i].reshape(28, 28), cmap='gray')
    plt.title(f"Pred: {np.argmax(predictions[i])}\nTrue: {np.argmax(y_test[i])}")
    plt.axis('off')
plt.tight_layout()
plt.show()
```

Show hidden output

6.Implement Neural network architecture for Multiclass Models for the data set available at Fashion MNIST data (available at https://keras.io/2.17/api/datasets/fashion_mnist/)

```
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt
# Load the Fashion MNIST dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.fashion_mnist.load_data()
# Normalize pixel values to the range [0, 1]
x_train, x_test = x_train / 255.0, x_test / 255.0
# Flatten the 28x28 images into 1D vectors of size 784
x_train = x_train.reshape(-1, 784)
x_test = x_test.reshape(-1, 784)
# Convert labels to one-hot encoding
y_train = keras.utils.to_categorical(y_train, 10)
```

```
y_test = keras.utils.to_categorical(y_test, 10)
# Define the neural network model
model = keras.Sequential([
    keras.layers.Dense(256, activation='relu', input_shape=(784,)),  # Hidden layer 1
    keras.layers.Dense(128, activation='relu'),  # Hidden layer 2
    keras.layers.Dense(64, activation='relu'),  # Hidden layer 3
    keras.layers.Dense(10, activation='softmax')  # Output layer (10 classes)
])
# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
# Train the model
model.fit(x_train, y_train, epochs=15, batch_size=32, validation_data=(x_test, y_test))
# Evaluate the model on the test set
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print(f"\nTest Accuracy: {test_accuracy:.4f}")
# Define class names for Fashion MNIST
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
# Predict a few test images
predictions = model.predict(x_test[:5])
# Display results
for i in range(5):
    plt.imshow(x_test[i].reshape(28, 28), cmap='gray')
    plt.title(f"Predicted: {class_names[np.argmax(predictions[i])]}, Actual: {class_names[np.argmax(y_test[i])]}")
    plt.axis('off')
    plt.show()
```

Show hidden output

7.Implement Neural network architecture for Multiclass Models for the data set available at CIFAR10 data (available at https://keras.io/2.17/api/datasets/cifar10/)

```
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt
# Load the CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
# Normalize pixel values to the range [0, 1]
x_train, x_test = x_train / 255.0, x_test / 255.0
# Convert labels to one-hot encoding
y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)
```

```
# Define the neural network model
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(32, 32, 3)),  # Flatten input images
    keras.layers.Dense(512, activation='relu'),  # Hidden layer 1
    keras.layers.Dense(256, activation='relu'),  # Hidden layer 2
    keras.layers.Dense(128, activation='relu'),  # Hidden layer 3
    keras.layers.Dense(10, activation='softmax')  # Output layer (10 classes)
])
# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
# Train the model
model.fit(x_train, y_train, epochs=20, batch_size=64, validation_data=(x_test, y_test))
# Evaluate the model on the test set
test_loss, test_accuracy = model.evaluate(x_test, y_test,verbose=0)
print(f"\nTest Accuracy: {test_accuracy:.4f}")
# Define class names for CIFAR-10
class_names = ["Airplane", "Automobile", "Bird", "Cat", "Deer",
               "Dog", "Frog", "Horse", "Ship", "Truck"]
# Predict a few test images
predictions = model.predict(x_test[:5])
# Display results
for i in range(5):
    plt.imshow(x_test[i])
    plt.title(f"Predicted: {class_names[np.argmax(predictions[i])]}, Actual: {class_names[np.argmax(y_test[i])]}")
    plt.axis('off')
    plt.show()
```

Show hidden output

WASTE LIST

```
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Dense, Flatten
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.utils import to_categorical
# Step 1: Prepare the corpus
corpus = ['The cat sat on the mat', 'The dog ran in the park', 'The bird sang in the tree']
# Step 2: Tokenize the text
tokenizer = Tokenizer()
tokenizer.fit_on_texts(corpus)
vocab_size = len(tokenizer.word_index) + 1  # Plus one for padding
word_index = tokenizer.word_index
print("Word Index:", word_index)
```

```
# Step 3: Create context-target pairs for CBOW (context is 2 words before and after the target word)
def generate_context_target_pairs(corpus, window_size=2):
    context = []
    target = []
    for sentence in corpus:
        words = sentence.split()
        for i in range(window_size, len(words) - window_size):
            context_words = words[i-window_size:i] + words[i+1:i+window_size+1]
            target_word = words[i]
            context.append([word_index[word] for word in context_words])
            target.append(word_index[target_word])
    return np.array(context), np.array(target)
context_data, target_data = generate_context_target_pairs(corpus)
# Step 4: Convert target labels to one-hot encoding
target_data = to_categorical(target_data, num_classes=vocab_size)
# Step 5: Build the CBOW model
embedding_dim = 5  # Dimensionality of the embedding layer
model = Sequential()
model.add(Embedding(input_dim=vocab_size, output_dim=embedding_dim, input_length=4))
model.add(Flatten())
model.add(Dense(vocab_size, activation='softmax'))
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
# Step 6: Train the model
model.fit(context_data, target_data, epochs=100)
# Step 7: Get the word embeddings (weights from the Embedding layer)
word_embeddings = model.layers[0].get_weights()[0]
print("Word Embeddings:\n", word_embeddings)
# Optionally, get the vector for a specific word
word_vector = word_embeddings[word_index['dog']]  # Get embedding for the word 'dog'
print("Embedding for 'dog':", word_vector)
```

```
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Dense, Flatten
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.utils import to_categorical
# Step 1: Prepare the corpus
corpus = ["the quick brown fox jumps", "over the lazy dog", "hello world"]
# Step 2: Tokenize the text
tokenizer = Tokenizer()
tokenizer.fit_on_texts(corpus)
vocab_size = len(tokenizer.word_index) + 1  # Plus one for padding
word_index = tokenizer.word_index
print("Word Index:", word_index)
# Step 3: Create context-target pairs for CBOW (context is 2 words before and after the target word)
def generate_context_target_pairs(corpus, window_size=2):
```

```
    context = []
    target = []
    for sentence in corpus:
        words = sentence.split()
        for i in range(window_size, len(words) - window_size):
            context_words = words[i-window_size:i] + words[i+1:i+window_size+1]
            target_word = words[i]
            context.append([word_index[word] for word in context_words])
            target.append(word_index[target_word])
    return np.array(context), np.array(target)
context_data, target_data = generate_context_target_pairs(corpus)
# Step 4: Convert target labels to one-hot encoding
target_data = to_categorical(target_data, num_classes=vocab_size)
# Step 5: Build the CBOW model
embedding_dim = 5  # Dimensionality of the embedding layer
model = Sequential()
model.add(Embedding(input_dim=vocab_size, output_dim=embedding_dim, input_length=4))
model.add(Flatten())
model.add(Dense(vocab_size, activation='softmax'))
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
# Step 6: Train the model
model.fit(context_data, target_data, epochs=100)
# Step 7: Get the word embeddings (weights from the Embedding layer)
word_embeddings = model.layers[0].get_weights()[0]
print("Word Embeddings:\n", word_embeddings)
# Optionally, get the vector for a specific word
word_vector = word_embeddings[word_index['quick']]  # Get embedding for the word 'quick'
print("Embedding for 'quick':", word_vector)
```

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

8. Train a Neural Network for Binary Classification (Digit 5 vs Non-5) - MNIST

```
import tensorflow as tf
from tensorflow import keras
import numpy as np

# Load MNIST dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
```

```
# Convert to binary classification (1 if 5, else 0)
y_train = (y_train == 5).astype(np.int32)
y_test = (y_test == 5).astype(np.int32)

# Normalize the images
x_train, x_test = x_train / 255.0, x_test / 255.0

# Flatten the images (28x28 → 784)
x_train = x_train.reshape(-1, 784)
x_test = x_test.reshape(-1, 784)

# Build the model
model = keras.Sequential([
    keras.layers.Dense(128, activation="relu", input_shape=(784,)),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dense(1, activation="sigmoid")  # Binary output
])

model.compile(optimizer="adam", loss="binary_crossentropy", metrics=["accuracy"])

# Train the model
model.fit(x_train, y_train, epochs=10, batch_size=32, validation_data=(x_test, y_test))

# Evaluate the model
loss, acc = model.evaluate(x_test, y_test)
print(f"Test Accuracy: {acc:.4f}")
```

Show hidden output

9. Train a Binary Classifier (Ankle Boot vs Non-Ankle Boot) - Fashion MNIST

```
# Load Fashion MNIST dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.fashion_mnist.load_data()

# Convert to binary classification (1 if Ankle Boot, else 0)
y_train = (y_train == 9).astype(np.int32)
y_test = (y_test == 9).astype(np.int32)

# Normalize and reshape
x_train, x_test = x_train / 255.0, x_test / 255.0
x_train, x_test = x_train.reshape(-1, 784), x_test.reshape(-1, 784)

# Build the model
model = keras.Sequential([
    keras.layers.Dense(128, activation="relu", input_shape=(784,)),
```

```
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dense(1, activation="sigmoid")
])

model.compile(optimizer="adam", loss="binary_crossentropy", metrics=["accuracy"])
model.fit(x_train, y_train, epochs=10, batch_size=32, validation_data=(x_test, y_test))
```

10. Train a Binary Classifier (Automobile vs Non-Automobile) - CIFAR-10

```
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()

# Convert to binary classification (1 if Automobile, else 0)
y_train = (y_train == 1).astype(np.int32).flatten()
y_test = (y_test == 1).astype(np.int32).flatten()

# Normalize the images
x_train, x_test = x_train / 255.0, x_test / 255.0

# Build CNN model
model = keras.Sequential([
    keras.layers.Conv2D(32, (3,3), activation="relu", input_shape=(32,32,3)),
    keras.layers.MaxPooling2D(2,2),
    keras.layers.Conv2D(64, (3,3), activation="relu"),
    keras.layers.MaxPooling2D(2,2),
    keras.layers.Flatten(),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dense(1, activation="sigmoid")
])

model.compile(optimizer="adam", loss="binary_crossentropy", metrics=["accuracy"])
model.fit(x_train, y_train, epochs=10, batch_size=32, validation_data=(x_test, y_test))
```

11. & 12. Effect of Gradient Descent Strategies on Multiclass Models

```
optimizers = {
    "SGD": keras.optimizers.SGD(),
    "Batch Gradient": keras.optimizers.SGD(momentum=0.9),
    "Mini-Batch": keras.optimizers.Adam()
}

for name, optimizer in optimizers.items():
```

```
for name, optimizer in optimizers.items():
    print(f"\nTraining with {name}...\n")

    # Load MNIST dataset
    (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

    # Normalize and reshape
    x_train, x_test = x_train / 255.0, x_test / 255.0
    x_train, x_test = x_train.reshape(-1, 784), x_test.reshape(-1, 784)

    # Build Model
    model = keras.Sequential([
        keras.layers.Dense(128, activation="relu", input_shape=(784,)),
        keras.layers.Dense(64, activation="relu"),
        keras.layers.Dense(10, activation="softmax")
    ])

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

    # Train
    model.fit(x_train, y_train, epochs=5, batch_size=32 if name == "Mini-Batch" else 60000, validation_data=(x_test, y_test))
```

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

```
import numpy as np
weights = np.random.rand(3, 1)
weights
```

```
array([[0.78856163],
       [0.09490114],
       [0.09623598]])
```

Start coding or generate with AI.

Start coding or generate with AI.