classifiers_step_by_step

September 22, 2024

1 Linear Classifier with Gradient Descent

In this lab, we will explore the implementation of a linear classifier from scratch. The topics covered include:

- Initialization of weights and bias
- Matrix multiplication of inputs (X) and weights (theta) with bias
- Loss (cost) function calculation
- Gradient Descent (both batch and stochastic)
- Weight update
- Use case for binomial and multinomial classification using sigmoid and softmax

---

1.1 0. Import necessary libraries

[1]:
```python
import os
import time
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import torch
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms

import torch.nn as nn
# import torchvision.transforms as transforms
# import torchvision.datasets as datasets
from torchvision import models
```

1.2 1. Initialization of Weights and Bias

Before training, we need to initialize our weights (`theta`) and bias (`b`). This can be done randomly or using a small constant value. In most cases, initializing weights with small random values works best to break symmetry, while bias can be initialized to zero.

1.3 2. Matrix Multiplication of X and θ with Bias

In linear models, the prediction is computed as the dot product between the input features X and the weight vector θ, plus the bias b. Mathematically, this is expressed as:

$$y = X\theta + b$$

To incorporate the bias term into the matrix multiplication, we can augment the input matrix X and the weight vector θ.

1.3.1 i. Augmenting X

Add a column of ones to the input matrix X to account for the bias term. Let X have dimensions $m \times n$ (where m is the number of samples and n is the number of features). The augmented matrix X_{bias} will have dimensions $m \times (n+1)$:

$$X_{\text{bias}} = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1n} \\ 1 & x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix}$$

1.3.2 ii. Augmenting θ

Extend θ to include the bias term. The extended vector θ_{bias} will have dimensions $(n+1) \times c$ (where c is the no. of classes):

$$\theta_{\text{bias}} = \begin{bmatrix} b \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix}$$

1.3.3 iii. Matrix Multiplication

With the augmented matrix X_{bias} and the extended vector θ_{bias}, the prediction y can be computed as:

$$y = X_{\text{bias}}\theta_{\text{bias}}$$

This approach simplifies the computation by integrating the bias term directly into the matrix multiplication, which can be more efficient and straightforward in practice, especially when using matrix operations libraries.

[2]:
```python
# Add bias term (column of 1s) to X
def add_bias_term(X):
    return np.c_[np.ones((X.shape[0], 1)), X]
```

[3]:
```
# Initialize weights with X updated to handle bias
def initialize_parameters(X, y, multiclass):
    n_features = X.shape[1]  # Number of features from the input X with bias␣
 ↪term
    if multiclass:
        n_classes = y.shape[1]
        theta = np.random.randn(n_features, n_classes) * 0.01  # Small random␣
 ↪weights
    else:
        theta = np.random.randn(n_features, 1) * 0.01  # Small random weights
    return theta
```

[4]:
```
# Linear prediction
def linear_prediction(X, theta):
    return np.dot(X, theta)
```

1.4 3. Loss Functions for Classification

For classification tasks, different loss functions are used depending on whether the task is binary classification or multinomial classification.

1.4.1 Binary Classification (Sigmoid/Logistic Regression)

The loss function used is binary cross-entropy, also known as log loss. It is defined as:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right]$$

where: - m is the number of samples. - y_i is the true label for the i-th sample. - $\hat{y}_i$ is the predicted probability for the i-th sample.

1.4.2 Multinomial Classification (Softmax)

For multinomial classification, especially after one-hot encoding the labels, the loss function is categorical cross-entropy. It is defined as:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_{ik}\log(\hat{y}_{ik})$$

where: - m is the number of samples. - K is the number of classes. - y_{ik} is the one-hot encoded label for the i-th sample and k-th class (binary indicator: 0 or 1). - $\hat{y}_{ik}$ is the predicted probability of class k for the i-th sample.

In one-hot encoding, y_{ik} is 1 if the i-th sample belongs to class k, and 0 otherwise. This loss function measures how well the predicted probabilities match the one-hot encoded true labels.

[5]:
```
# Binary Cross Entropy Loss
def binary_cross_entropy_loss(y_true, y_pred):
    m = y_true.shape[0]
    epsilon = 1e-15  # To avoid log(0) this is a very small value
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    return -np.sum(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))␣
 ↪/ m

# Categorical Cross Entropy Loss
def categorical_cross_entropy_loss(y_true, y_pred):
    m = y_true.shape[0]
    epsilon = 1e-15  # To avoid log(0)
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    return -np.sum(y_true * np.log(y_pred)) / m
```

1.5 4. Gradient Descent for Minimizing the Loss Function with weight updates

Gradient descent is used to minimize the loss function by iteratively updating the weights based on the gradient of the loss function.

1.5.1 Batch Gradient Descent

In Batch Gradient Descent, the gradient is computed using all examples in the dataset:

$$\theta = \theta - \alpha \frac{\partial J(\theta)}{\partial \theta}$$

where: - α is the learning rate. - $\frac{\partial J(\theta)}{\partial \theta}$ is the gradient of the loss function with respect to the weights.

1.5.2 Stochastic Gradient Descent (SGD)

In Stochastic Gradient Descent, the gradient is computed using only one example at a time:

$$\theta = \theta - \alpha \frac{\partial J(\theta)}{\partial \theta}$$

where: - α is the learning rate. - $\frac{\partial J(\theta)}{\partial \theta}$ is the gradient of the loss function with respect to the weights, computed for a single example.

In both cases, the update rule for weights is the same, but the difference lies in how the gradient is computed: either over the entire dataset (Batch Gradient Descent) or a single example (Stochastic Gradient Descent).

1.6 5. Weight Update

After calculating the gradient, we update the weights using the formula mentioned above. Depending on whether we are using batch gradient descent or stochastic gradient descent, the weight update happens differently.

[6]:
```
def gradient_descent_step(X, y, predictions, theta, learning_rate, multiclass):
    m = X.shape[0]
    # predictions = linear_prediction(X, theta)
    if multiclass:
        # predictions = softmax(predictions)
        gradients = (1/m) * np.dot(X.T, (predictions - y))
    else:
        # predictions = sigmoid(predictions)
        gradients = (1/m) * np.dot(X.T, (predictions - y))
    theta = theta - learning_rate * gradients
    return theta
```

[7]:
```
# Stochastic Gradient Descent (SGD) Step
def stochastic_gradient_descent_step(X, y, theta, learning_rate, multiclass):
    m = X.shape[0]
    for i in range(m):
        xi = X[i:i+1]
        yi = y[i:i+1]
        # print (theta.shape)
        prediction = linear_prediction(xi, theta)
        if multiclass:
            prediction = softmax(prediction)
        else:
            prediction = sigmoid(prediction)
        # print(prediction.shape)
        # print(xi.T.shape)
        # print(yi.shape)
        gradients = np.dot(xi.T, (prediction - yi))
        theta = theta - learning_rate * gradients
    return theta
```

1.7 Activation: Binomial Classification (Sigmoid)

In binary classification, the sigmoid function is applied to the linear output to obtain the predicted probability. The sigmoid function $\sigma(z)$ is defined as:

$$\hat{y} = \sigma(X\theta) = \frac{1}{1 + e^{-X\theta}}$$

where: - $\hat{y}$ is the predicted probability. - $X\theta$ represents the linear combination of the input features X and the weights θ. - e is the base of the natural logarithm.

The sigmoid function maps the linear output to a probability value between 0 and 1, which can then be used to make a classification decision.

1.8 Activation: Multinomial Classification (Softmax)

In multinomial classification, the softmax function is used to compute probabilities across multiple classes. The softmax function softmax(z_i) for class k is defined as:

$$\hat{y}_{ik} = \frac{e^{(X\theta_k)}}{\sum_{j=1}^{K} e^{(X\theta_j)}}$$

where: - $\hat{y}_{ik}$ is the predicted probability of the i-th sample belonging to class k. - $X\theta_k$ is the linear combination of the input features X and the weights θ_k for class k. - K is the total number of classes. - The denominator is the sum of the exponentials of the linear combinations for all classes, ensuring that the probabilities sum up to 1.

The softmax function converts the linear outputs into a probability distribution over multiple classes, which is useful for making predictions in multiclass classification problems.

[8]:
```
# Sigmoid function for binary classification
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Softmax function for multi-class
def softmax(z):
    exp_scores = np.exp(z - np.max(z, axis=1, keepdims=True))  # For numerical␣
↪stability
    return exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
```

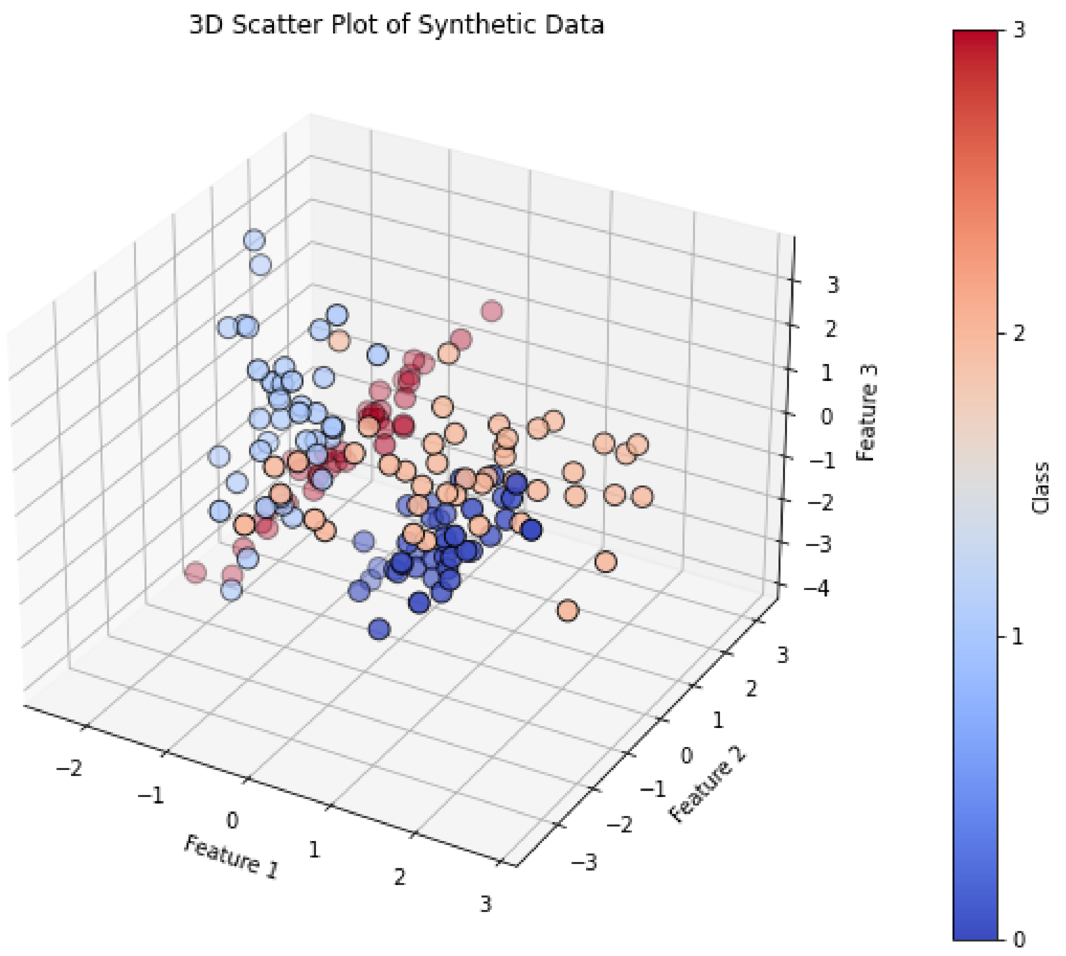
1.8.1 OKAY LETS GO A HEAD AND TRAIN OUR MODEL !!

[9]:
```
# Training function
def train_model(X, y, learning_rate=0.01, iterations=1000, batch=True,␣
↪multiclass=False):
    # Add bias term to X
    X = add_bias_term(X)

    # Initialize theta
    theta = initialize_parameters(X, y, multiclass)

    # Track loss over iterations
    losses = []

    for i in range(iterations):
        # Compute predictions and loss
        if multiclass:
            predictions = softmax(linear_prediction(X, theta))
            loss = categorical_cross_entropy_loss(y, predictions)
        else:
            predictions = sigmoid(linear_prediction(X, theta))
            loss = binary_cross_entropy_loss(y, predictions)
        losses.append(loss)

        # Update weights
        if batch:
            theta = gradient_descent_step(X, y, predictions, theta,␣
 ↪learning_rate, multiclass)
        else:
            # Use stochastic gradient descent
            theta = stochastic_gradient_descent_step(X, y, theta,␣
 ↪learning_rate, multiclass)

        # Print loss every 100 iterations
        if i % 100 == 0:
            print(f"Iteration {i}/{iterations}, Loss: {loss:.4f}")

    return theta, losses
```

1.8.2 Lets generate some data!!

[10]:
```
# Set multiclass to True or False
multiclass = True  # Set to True for multiclass classification

if multiclass:
    # For multiclass classification
    X_syn, y_syn = make_classification(n_samples=200, n_features=3,␣
 ↪n_informative=3,
                                   n_redundant=0, n_clusters_per_class=1,␣
 ↪n_classes=4, random_state=100)
    # Convert y to one-hot encoding
    y_syn = np.eye(np.max(y_syn) + 1)[y_syn]
else:
    # For binary classification
    X_syn, y_syn = make_classification(n_samples=200, n_features=2,␣
 ↪n_classes=2, n_informative=2, n_redundant=0, random_state=100)
    y_syn = y_syn.reshape(-1, 1)  # Reshape y to be a column vector

# Split into training and test sets
X_train_syn, X_test_syn, y_train_syn, y_test_syn = train_test_split(X_syn,␣
 ↪y_syn, test_size=0.2, random_state=100)
```

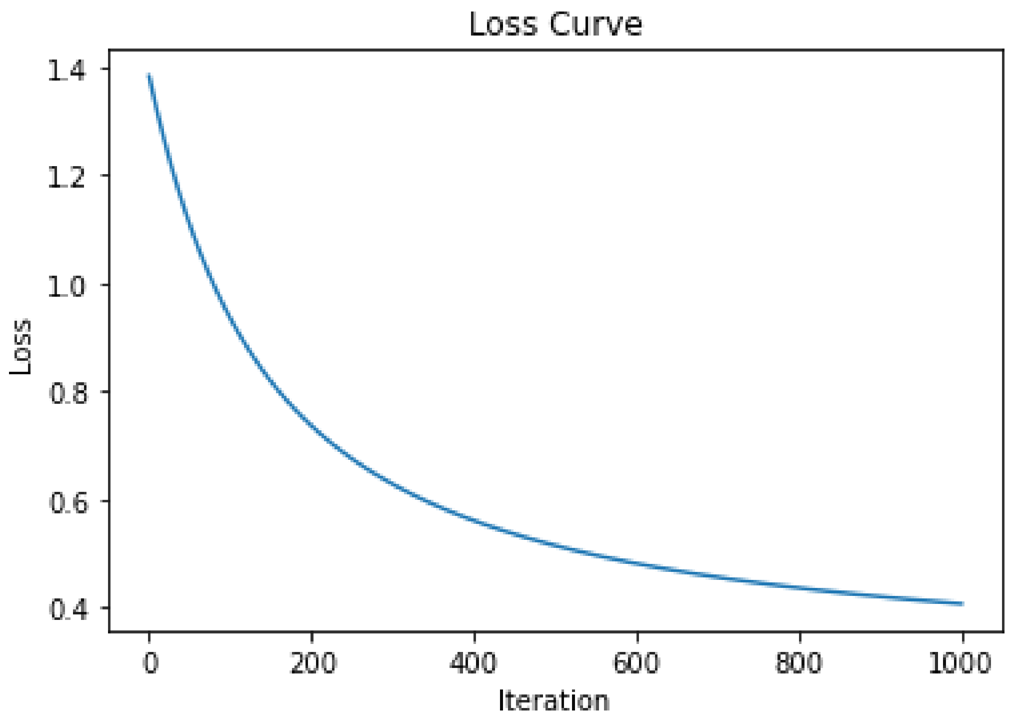
[11]:
```
if multiclass:
    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection='3d')
    # Convert one-hot encoding to class labels for plotting
    y_train_labels = np.argmax(y_train_syn, axis=1)
    # Use the 3 features for the scatter plot
    scatter = ax.scatter(X_train_syn[:, 0], X_train_syn[:, 1], X_train_syn[:,␣
↪2],
                         c=y_train_labels, cmap='coolwarm', edgecolor='k',␣
↪s=100)

    # Add labels
    ax.set_title("3D Scatter Plot of Synthetic Data")
    ax.set_xlabel("Feature 1")
    ax.set_ylabel("Feature 2")
    ax.set_zlabel("Feature 3")

    # Add color bar to represent class labels
    cbar = fig.colorbar(scatter, ax=ax, pad=0.1)
    cbar.set_label('Class')

    # Set ticks to be integers corresponding to class labels
    cbar.set_ticks(np.arange(np.min(y_train_labels), np.max(y_train_labels) +␣
↪1))

    plt.show()
else:
    plt.figure(figsize=(10, 8))
    plt.scatter(X_train_syn[:, 0], X_train_syn[:, 1], c=y_train_syn,␣
↪cmap='coolwarm', edgecolor='k', s=100)
    plt.title("Scatter Plot of Training Data")
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
    plt.show()
```



```
[12]: start_time = time.time()
      theta, losses = train_model(X_train_syn, y_train_syn, learning_rate=0.01,␣
       ↪iterations=1000, batch=True, multiclass=multiclass)
      print(f"Time Taken Using Batch gradient descent :{time.time() - start_time}")
```

```
Iteration 0/1000, Loss: 1.3832
Iteration 100/1000, Loss: 0.9343
Iteration 200/1000, Loss: 0.7356
Iteration 300/1000, Loss: 0.6275
Iteration 400/1000, Loss: 0.5600
Iteration 500/1000, Loss: 0.5140
Iteration 600/1000, Loss: 0.4806
Iteration 700/1000, Loss: 0.4553
Iteration 800/1000, Loss: 0.4354
Iteration 900/1000, Loss: 0.4194
Time Taken Using Batch gradient descent :0.11540985107421875
```

[13]:
```python
plt.plot(losses)
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.title('Loss Curve')
plt.show()
```



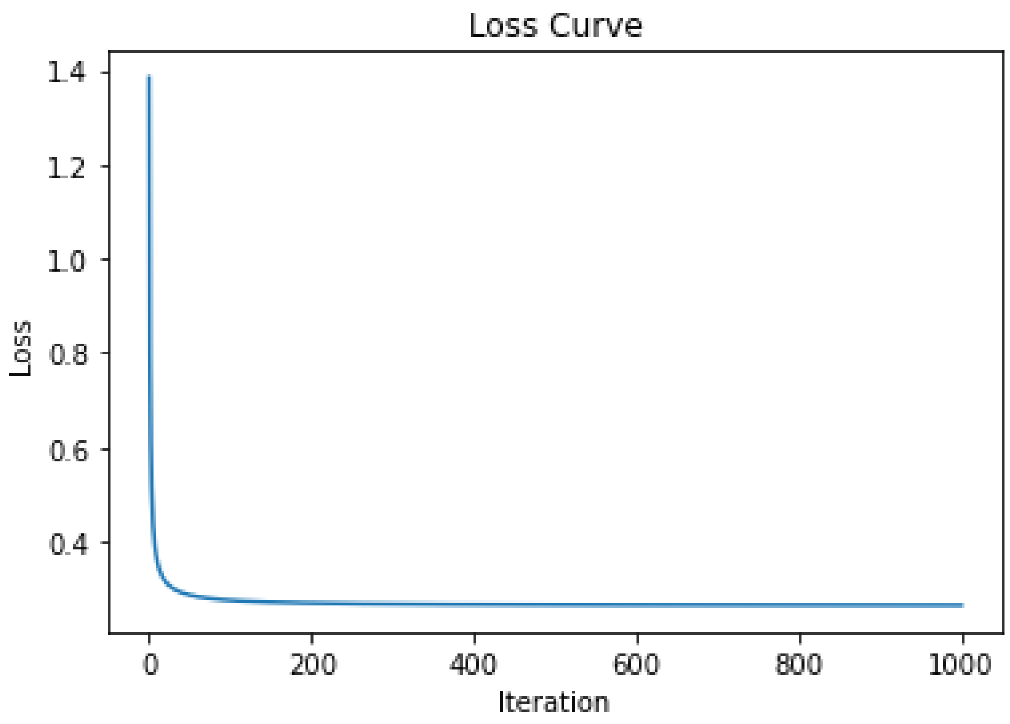
[14]:
```python
# Add bias term to test data
X_test_bias = add_bias_term(X_test_syn)

if multiclass:
    predictions = softmax(linear_prediction(X_test_bias, theta))
    predicted_classes = np.argmax(predictions, axis=1)
    true_classes = np.argmax(y_test_syn, axis=1)
else:
    predictions = sigmoid(linear_prediction(X_test_bias, theta))
    predicted_classes = (predictions >= 0.5).astype(int)
    true_classes = y_test_syn

# Calculate accuracy
accuracy = np.mean(predicted_classes.flatten() == true_classes.flatten())
print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

```
Test Accuracy: 90.00%
```

[15]:
```
start_time = time.time()
theta, losses = train_model(X_train_syn, y_train_syn, learning_rate=0.01,␣
 ↪iterations=1000, batch=False, multiclass=multiclass)
print(f"Time Taken Using Stochastic gradient descent :{time.time() -␣
 ↪start_time}")
```

```
Iteration 0/1000, Loss: 1.3847
Iteration 100/1000, Loss: 0.2755
Iteration 200/1000, Loss: 0.2695
Iteration 300/1000, Loss: 0.2675
Iteration 400/1000, Loss: 0.2666
Iteration 500/1000, Loss: 0.2660
Iteration 600/1000, Loss: 0.2657
Iteration 700/1000, Loss: 0.2654
Iteration 800/1000, Loss: 0.2653
Iteration 900/1000, Loss: 0.2652
Time Taken Using Stochastic gradient descent :4.87663459777832
```

[16]:
```
plt.plot(losses)
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.title('Loss Curve')
plt.show()
```



[17]:
```
# Add bias term to test data
X_test_bias = add_bias_term(X_test_syn)

if multiclass:
    predictions = softmax(linear_prediction(X_test_bias, theta))
    predicted_classes = np.argmax(predictions, axis=1)
    true_classes = np.argmax(y_test_syn, axis=1)
else:
    predictions = sigmoid(linear_prediction(X_test_bias, theta))
    predicted_classes = (predictions >= 0.5).astype(int)
    true_classes = y_test_syn

# Calculate accuracy
accuracy = np.mean(predicted_classes.flatten() == true_classes.flatten())
print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

```
Test Accuracy: 92.50%
```

1.9 For Image what could be possible changes??

[18]:
```
# For our puffer surver we need to browse via a proxy!!

# Set HTTP and HTTPS proxy
os.environ['http_proxy'] = 'http://192.41.170.23:3128'
os.environ['https_proxy'] = 'http://192.41.170.23:3128'
```

[19]:
```
# Check if GPU is available
# device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
device = torch.device("cpu")
print(f"Using device: {device}")
```

```
Using device: cpu
```

[20]:
```
cifar_train = datasets.CIFAR10('../LAB_04/data', train=True, download=True␣
 ↪,transform=transforms.ToTensor())
cifar_test = datasets.CIFAR10('../LAB_04/data', train=False, download=True␣
 ↪,transform=transforms.ToTensor())
```

```
Files already downloaded and verified
Files already downloaded and verified
```

[21]:
```
# Function to subsample CIFAR-10 dataset
def subsample_dataset(dataset, sample_size=1000):
    indices = np.random.choice(len(dataset), sample_size, replace=False)
    subset = Subset(dataset, indices)
    return subset
```

```
# Subsample the training and test datasets
sample_size = 1000
train_subset = subsample_dataset(cifar_train, sample_size=sample_size)
test_subset = subsample_dataset(cifar_test, sample_size=int(sample_size * 0.4))

# Load data into PyTorch DataLoader
train_loader = DataLoader(train_subset, batch_size=sample_size, shuffle=True)
test_loader = DataLoader(test_subset, batch_size=int(sample_size * 0.4),␣
 ↪shuffle=False)

# Fetch all data and labels for easier handling
X_train, y_train = next(iter(train_loader))
X_test, y_test = next(iter(test_loader))

print("Before Flattening")
print(f"Training data shape: {X_train.shape}")
print(f"Test data shape: {X_test.shape}")

# Reshape the images to 2D for the KNN algorithm
X_train = X_train.view(X_train.size(0), -1).to(device)  # Flatten
X_test = X_test.view(X_test.size(0), -1).to(device)
y_train = y_train.to(device)
y_test = y_test.to(device)

print("After Flattening")
print(f"Training data shape: {X_train.shape}")
print(f"Test data shape: {X_test.shape}")
```

```
Before Flattening
Training data shape: torch.Size([1000, 3, 32, 32])
Test data shape: torch.Size([400, 3, 32, 32])
After Flattening
Training data shape: torch.Size([1000, 3072])
Test data shape: torch.Size([400, 3072])
```

[22]:
```
class ImageLinearClassifier:
    def __init__(self, input_size, n_classes):
        self.W = np.random.randn(n_classes, input_size) * 0.01  # Small random␣
 ↪weights (10,3072)
        self.b = np.zeros((n_classes, 1))  # Bias initialized to zero (10,1)

    def predict(self, X):
        # Reshape X to (input_size, batch_size)
        X=X.T  # Transpose to shape (3072,1000)
        return np.dot(self.W, X) + self.b

    def compute_loss(self, X, y):
        """
            X: (batch_size, input_size) = (1000, 3072)
            y: (batch_size,) = (1000,) with class labels (0-9)
        """
        m = X.shape[0]
        z = self.predict(X)
        probs = self.softmax(z)
        log_likelihood = -np.log(probs[y, range(m)])
        return np.sum(log_likelihood) / m

    def softmax(self, z):
        exp_z = np.exp(z - np.max(z, axis=0, keepdims=True))  # Numerical␣
↪stability
        return exp_z / np.sum(exp_z, axis=0, keepdims=True)

    def gradient_descent(self, X, y, learning_rate=0.001):
        # Compute the gradient and update W, b
        m = X.shape[0]
        z = self.predict(X)
        probs = self.softmax(z)
        probs[y, range(m)] -= 1  # Gradient of softmax loss wrt z
        dW = np.dot(probs, X) / m  #Gradient wrt weights
        db = np.sum(probs, axis=1, keepdims=True) / m

        # Update weights and bias
        self.W -= learning_rate * dW
        self.b -= learning_rate * db
```

[23]:
```
def train(classifier, X_train, y_train, epochs, learning_rate):
    losses = []
    for i in range(epochs):
        loss = classifier.compute_loss(X_train, y_train)
        losses.append(loss)
        print(f'Epoch {i+1}, Loss: {loss}')
        classifier.gradient_descent(X_train, y_train, learning_rate)
    return losses
```

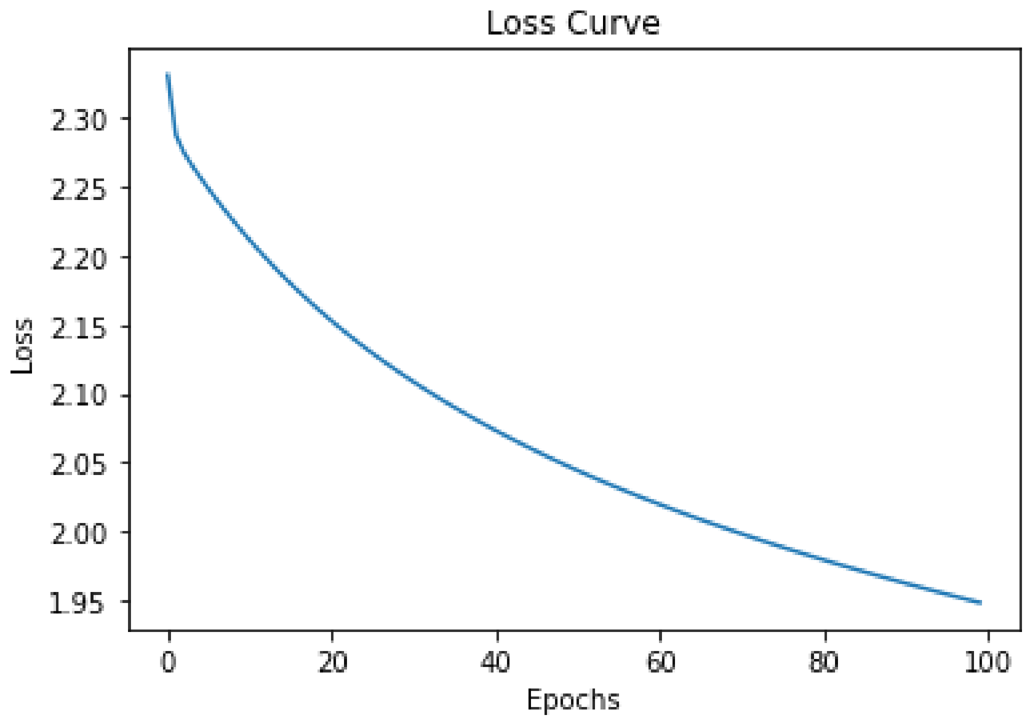
[24]:
```
print(f"Training data: {len(cifar_train)}")
print(f"Test data: {len(cifar_test)}")

image, label = cifar_train[0]
# Now you can check the shape of the image
print(f"Image shape: {image.shape}")
```

```
Training data: 50000
Test data: 10000
Image shape: torch.Size([3, 32, 32])
```

[25]:
```python
# Example usage
n_classes = 10  # For CIFAR-10
image_size = 32 * 32 * 3  # CIFAR-10 images are 32x32x3
classifier = ImageLinearClassifier(input_size=image_size, n_classes=n_classes)

# X_train is shape (image_size, batch_size) and y_train is (batch_size,)
losses = train(classifier, X_train, y_train, epochs=100, learning_rate=0.01)
```

```
Epoch 1, Loss: 2.3304049007235244
Epoch 2, Loss: 2.287276283110353
Epoch 3, Loss: 2.2742305920019072
Epoch 4, Loss: 2.2648641862213035
Epoch 5, Loss: 2.2562884338556035
Epoch 6, Loss: 2.24807288302408
Epoch 7, Loss: 2.2401503632463027
Epoch 8, Loss: 2.2325006646169703
Epoch 9, Loss: 2.225109864392534
Epoch 10, Loss: 2.2179653966146318
Epoch 11, Loss: 2.211055410306573
Epoch 12, Loss: 2.2043686632259583
Epoch 13, Loss: 2.1978944881247684
Epoch 14, Loss: 2.1916227692378647
Epoch 15, Loss: 2.185543919837034
Epoch 16, Loss: 2.179648859541945
Epoch 17, Loss: 2.1739289913805266
Epoch 18, Loss: 2.1683761788111506
Epoch 19, Loss: 2.162982722932181
Epoch 20, Loss: 2.1577413400739913
Epoch 21, Loss: 2.1526451399307374
Epoch 22, Loss: 2.1476876043532847
Epoch 23, Loss: 2.1428625668931343
Epoch 24, Loss: 2.13816419316034
Epoch 25, Loss: 2.133586962036072
Epoch 26, Loss: 2.1291256477620863
Epoch 27, Loss: 2.124775302914536
Epoch 28, Loss: 2.120531242257661
Epoch 29, Loss: 2.1163890274636326
Epoch 30, Loss: 2.1123444526776516
Epoch 31, Loss: 2.1083935309020814
Epoch 32, Loss: 2.1045324811694957
Epoch 33, Loss: 2.1007577164719087
Epoch 34, Loss: 2.097065832411761
Epoch 35, Loss: 2.093453596539401
Epoch 36, Loss: 2.089917938341535
Epoch 37, Loss: 2.0864559398454086
Epoch 38, Loss: 2.0830648268040988
Epoch 39, Loss: 2.079741960429232
Epoch 40, Loss: 2.076484829638582
Epoch 41, Loss: 2.07329104378729
Epoch 42, Loss: 2.070158325852821
Epoch 43, Loss: 2.067084506045223
Epoch 44, Loss: 2.0640675158157307
Epoch 45, Loss: 2.0611053822381855
Epoch 46, Loss: 2.058196222739248
Epoch 47, Loss: 2.055338240154743
Epoch 48, Loss: 2.05252971809088
Epoch 49, Loss: 2.0497690165704063
Epoch 50, Loss: 2.0470545679450023
Epoch 51, Loss: 2.0443848730564445
Epoch 52, Loss: 2.041758497630196
Epoch 53, Loss: 2.039174068886163
Epoch 54, Loss: 2.036630272352381
Epoch 55, Loss: 2.034125848868332
Epoch 56, Loss: 2.0316595917655262
Epoch 57, Loss: 2.0292303442137745
Epoch 58, Loss: 2.0268369967224142
Epoch 59, Loss: 2.024478484786454
Epoch 60, Loss: 2.0221537866682913
Epoch 61, Loss: 2.0198619213063216
Epoch 62, Loss: 2.0176019463423165
Epoch 63, Loss: 2.015372956260031
Epoch 64, Loss: 2.013174080628008
Epoch 65, Loss: 2.011004482440021
Epoch 66, Loss: 2.008863356547055
Epoch 67, Loss: 2.006749928175124
Epoch 68, Loss: 2.0046634515236326
Epoch 69, Loss: 2.002603208439318
Epoch 70, Loss: 2.000568507161169
Epoch 71, Loss: 1.9985586811320053
Epoch 72, Loss: 1.9965730878727113
Epoch 73, Loss: 1.9946111079153532
Epoch 74, Loss: 1.9926721437916959
Epoch 75, Loss: 1.9907556190738287
Epoch 76, Loss: 1.9888609774638522
Epoch 77, Loss: 1.9869876819297587
Epoch 78, Loss: 1.9851352138848326
Epoch 79, Loss: 1.9833030724080722
Epoch 80, Loss: 1.981490773503284
Epoch 81, Loss: 1.9796978493946595
Epoch 82, Loss: 1.9779238478567795
Epoch 83, Loss: 1.9761683315771201
Epoch 84, Loss: 1.9744308775492576
Epoch 85, Loss: 1.9727110764950715
Epoch 86, Loss: 1.9710085323143711
Epoch 87, Loss: 1.9693228615604437
Epoch 88, Loss: 1.9676536929401252
Epoch 89, Loss: 1.9660006668370866
Epoch 90, Loss: 1.9643634348570918
Epoch 91, Loss: 1.962741659394071
Epoch 92, Loss: 1.9611350132159107
Epoch 93, Loss: 1.95954317906894
Epoch 94, Loss: 1.9579658493001377
Epoch 95, Loss: 1.9564027254961565
Epoch 96, Loss: 1.9548535181382973
Epoch 97, Loss: 1.9533179462726316
Epoch 98, Loss: 1.9517957371945063
Epoch 99, Loss: 1.9502866261467104
Epoch 100, Loss: 1.9487903560306272
```

[26]:
```
plt.plot(losses)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Loss Curve')
plt.show()
```



1.10 Key Components:

1. Input Layer: The input data, similar to your previous setup.
2. Hidden Layer(s): These layers will have weights, biases, and non-linear activations like ReLU.

3. Output Layer: This will have a softmax activation for classification.
4. Loss Function: Cross-entropy loss for classification.
5. Backpropagation: To update weights using gradients from the loss.

1.11 MLP Structure Example:

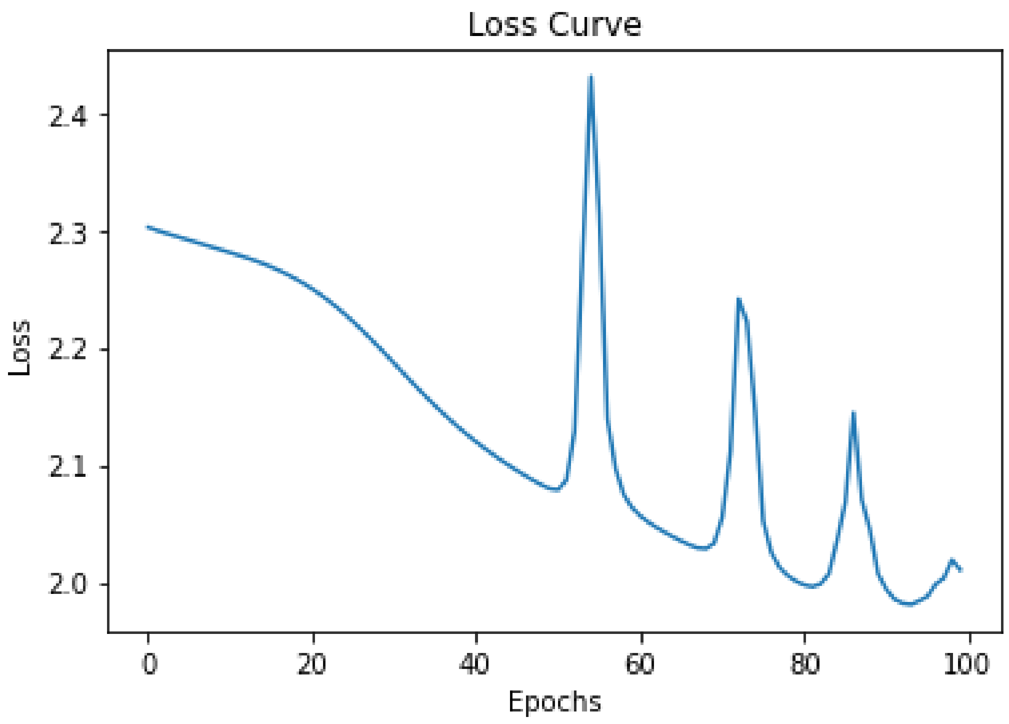
1. Input Layer: (3072 neurons, corresponding to image size 32x32x3 in CIFAR-10)
2. Hidden Layer 1: Fully connected, with a non-linear activation like ReLU.
3. Hidden Layer 2: Another fully connected layer (optional).
4. Output Layer: A fully connected layer with 10 neurons (for 10 classes) and softmax activation.

[27]:
```python
class MLPClassifier:
    def __init__(self, input_size, hidden_size, output_size):
        # Weight initialization
        self.W1 = np.random.randn(hidden_size, input_size) * 0.01  #␣
↪(hidden_size, input_size)
        self.b1 = np.zeros((hidden_size, 1))  # (hidden_size, 1)
        self.W2 = np.random.randn(output_size, hidden_size) * 0.01  #␣
↪(output_size, hidden_size)
        self.b2 = np.zeros((output_size, 1))  # (output_size, 1)

    def relu(self, z):
        return np.maximum(0, z)

    def relu_derivative(self, z):
        return np.where(z > 0, 1, 0)

    def softmax(self, z):
        exp_z = np.exp(z - np.max(z, axis=0, keepdims=True))  # Numerical␣
↪stability
        return exp_z / np.sum(exp_z, axis=0, keepdims=True)

    def forward(self, X):
        """
        Forward pass through the network.
        X: input data of shape (input_size, batch_size)
        """
        # Layer 1 (hidden layer)
        self.Z1 = np.dot(self.W1, X) + self.b1  # (hidden_size, batch_size)
        self.A1 = self.relu(self.Z1)  # Apply ReLU activation

        ## ADD ANOTHER HIDDEN LAYER IN YOUR TAKE HOME EXERCISE

        # Layer 2 (output layer)
        self.Z2 = np.dot(self.W2, self.A1) + self.b2  # (output_size,␣
↪batch_size)
        self.A2 = self.softmax(self.Z2)  # Apply softmax activation
        return self.A2

    def compute_loss(self, A2, y):
        """
        Compute cross-entropy loss.
        A2: output from softmax, shape (output_size, batch_size)
        y: true labels, shape (batch_size,)
        """
        m = y.shape[0]  # batch size
        log_likelihood = -np.log(A2[y, range(m)])
        loss = np.sum(log_likelihood) / m
        return loss

    def backward(self, X, y, learning_rate=0.01):
        """
        Perform backward propagation and update weights.
        X: input data of shape (input_size, batch_size)
        y: true labels of shape (batch_size,)
        """
        m = X.shape[1]  # Batch size

        # Gradient of the loss w.r.t. Z2
        dZ2 = self.A2  # Softmax probabilities
        dZ2[y, range(m)] -= 1  # Subtract 1 from the correct class probabilities
        dZ2 /= m

        # Gradients for W2 and b2
        dW2 = np.dot(dZ2, self.A1.T)  # (output_size, hidden_size)
        db2 = np.sum(dZ2, axis=1, keepdims=True)  # (output_size, 1)

        # Gradients for the hidden layer (backprop through ReLU)
        dA1 = np.dot(self.W2.T, dZ2)  # (hidden_size, batch_size)
        dZ1 = dA1 * self.relu_derivative(self.Z1)  # Backprop through ReLU

        # Gradients for W1 and b1
        dW1 = np.dot(dZ1, X.T)  # (hidden_size, input_size)
        db1 = np.sum(dZ1, axis=1, keepdims=True)  # (hidden_size, 1)

        # Update weights and biases
        self.W1 -= learning_rate * dW1
        self.b1 -= learning_rate * db1
        self.W2 -= learning_rate * dW2
        self.b2 -= learning_rate * db2

    def train(self, X_train, y_train, epochs=100, learning_rate=0.01):
        """
        Train the network.
        X_train: input data, shape (input_size, batch_size)
        y_train: true labels, shape (batch_size,)
        """
        losses = []
        for i in range(epochs):
            # Forward pass
            A2 = self.forward(X_train)

            # Compute the loss
            loss = self.compute_loss(A2, y_train)
            print(f'Epoch {i+1}, Loss: {loss}')
            losses.append(loss)

            # Backward pass
            self.backward(X_train, y_train, learning_rate)
        return losses
```

[28]:
```
input_size = 3072  # CIFAR-10 images are 32x32x3
hidden_size = 100  # Arbitrary hidden layer size
output_size = 10  # 10 classes for CIFAR-10

mlp = MLPClassifier(input_size, hidden_size, output_size)

X_train_copy = X_train.T

losses = mlp.train(X_train_copy, y_train, epochs=100, learning_rate=0.1)
```

```
Epoch 1, Loss: 2.302355374060994
Epoch 2, Loss: 2.299853177391281
Epoch 3, Loss: 2.2975974559576366
Epoch 4, Loss: 2.295492334861766
Epoch 5, Loss: 2.2934228384245516
Epoch 6, Loss: 2.291342372439294
Epoch 7, Loss: 2.2892346736474494
Epoch 8, Loss: 2.2871147814729693
Epoch 9, Loss: 2.2849892607400557
Epoch 10, Loss: 2.2828621200387857
Epoch 11, Loss: 2.280724138787242
Epoch 12, Loss: 2.278568867491627
Epoch 13, Loss: 2.276337765541998
Epoch 14, Loss: 2.273979308180716
Epoch 15, Loss: 2.2714220939878413
Epoch 16, Loss: 2.268622357740994
Epoch 17, Loss: 2.2655360333298793
Epoch 18, Loss: 2.262131620413363
Epoch 19, Loss: 2.2584277533219828
Epoch 20, Loss: 2.254374770210219
Epoch 21, Loss: 2.249954676625499
Epoch 22, Loss: 2.2451471396375537
Epoch 23, Loss: 2.2399334015779995
Epoch 24, Loss: 2.23432129190244
Epoch 25, Loss: 2.228329836602421
Epoch 26, Loss: 2.2219856188883407
Epoch 27, Loss: 2.2153121891146355
Epoch 28, Loss: 2.2083530868670285
Epoch 29, Loss: 2.2011620901610573
Epoch 30, Loss: 2.193809979403862
Epoch 31, Loss: 2.1863668924374537
Epoch 32, Loss: 2.1789079339176394
Epoch 33, Loss: 2.1714991686737624
Epoch 34, Loss: 2.1642098572753956
Epoch 35, Loss: 2.157086930057767
Epoch 36, Loss: 2.1501714397540077
Epoch 37, Loss: 2.1434940130950646
Epoch 38, Loss: 2.137071633822875
Epoch 39, Loss: 2.1309250860803273
Epoch 40, Loss: 2.1250562552416095
Epoch 41, Loss: 2.11945915776428
Epoch 42, Loss: 2.1141169350702795
Epoch 43, Loss: 2.1090168467358987
Epoch 44, Loss: 2.1041418088880035
Epoch 45, Loss: 2.09947643110609
Epoch 46, Loss: 2.094992756540658
Epoch 47, Loss: 2.0906921337356197
Epoch 48, Loss: 2.086590622930542
Epoch 49, Loss: 2.082767905562145
Epoch 50, Loss: 2.079644564262264
Epoch 51, Loss: 2.078738776073633
Epoch 52, Loss: 2.086930306482191
Epoch 53, Loss: 2.128429980726308
Epoch 54, Loss: 2.2816391256166613
Epoch 55, Loss: 2.4307144796405837
Epoch 56, Loss: 2.3139509345260367
Epoch 57, Loss: 2.1390766841948836
Epoch 58, Loss: 2.0967038717037565
Epoch 59, Loss: 2.0745420552862424
Epoch 60, Loss: 2.0634522020111152
Epoch 61, Loss: 2.0564074873016818
Epoch 62, Loss: 2.0510180989149576
Epoch 63, Loss: 2.0464916314612513
Epoch 64, Loss: 2.042391808711395
Epoch 65, Loss: 2.0385633559223884
Epoch 66, Loss: 2.0349696128381964
Epoch 67, Loss: 2.031702429488864
Epoch 68, Loss: 2.0291459109429395
Epoch 69, Loss: 2.0285101850465144
Epoch 70, Loss: 2.033079566201084
Epoch 71, Loss: 2.0554132061477226
Epoch 72, Loss: 2.1113433351196864
Epoch 73, Loss: 2.2414131112534013
Epoch 74, Loss: 2.2220771986182015
Epoch 75, Loss: 2.1462394097740747
Epoch 76, Loss: 2.0518004422970515
Epoch 77, Loss: 2.0247714723264543
Epoch 78, Loss: 2.01229504843018
Epoch 79, Loss: 2.0054308848431837
Epoch 80, Loss: 2.0006968688591216
Epoch 81, Loss: 1.9975520736411811
Epoch 82, Loss: 1.996082973094864
Epoch 83, Loss: 1.9984849189675309
Epoch 84, Loss: 2.007017060908138
Epoch 85, Loss: 2.035652433941653
Epoch 86, Loss: 2.0677128009686356
Epoch 87, Loss: 2.144324140444247
Epoch 88, Loss: 2.070622561739935
Epoch 89, Loss: 2.0447223259574274
Epoch 90, Loss: 2.0069764506317123
Epoch 91, Loss: 1.993857282491808
Epoch 92, Loss: 1.9853501548337895
Epoch 93, Loss: 1.9819252691045408
Epoch 94, Loss: 1.9808520774987064
Epoch 95, Loss: 1.9840093918388666
Epoch 96, Loss: 1.9877709803490153
Epoch 97, Loss: 1.998156908901964
Epoch 98, Loss: 2.0033919559020847
Epoch 99, Loss: 2.0189353751423242
Epoch 100, Loss: 2.0104883916187757
```

[29]:
```
plt.plot(losses)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Loss Curve')
plt.show()
```



1.12 TAKE AWAY EXERCISE

- In the picture above the learning rate is high so the model is not able to find a minima and is overshooting.
- Use a learning rate scheduler in the above implementation, increase the no. of epochs and train on full data.
- Also increase the complexity of structure by adding another hidden layer

[30]:
```
# your code here
class MLPClassifier:
    def __init__(self, input_size, hidden_size, output_size):
        # Weight initialization
        self.W1 = np.random.randn(hidden_size, input_size) * 0.01  #␣
 ↪(hidden_size, input_size)
        self.b1 = np.zeros((hidden_size, 1))  # (hidden_size, 1)
        self.W2 = np.random.randn(output_size, hidden_size) * 0.01  #␣
 ↪(output_size, hidden_size)
        self.b2 = np.zeros((output_size, 1))  # (output_size, 1)
        self.W3 = np.random.randn(hidden_size, hidden_size) * 0.01  #␣
 ↪(hidden_size, hidden_size)
        self.b3 = np.zeros((hidden_size, 1))  # (hidden_size, 1)
        self.best_loss = float('inf')
    def relu(self, z):
        return np.maximum(0, z)

    def relu_derivative(self, z):
        return np.where(z > 0, 1, 0)

    def softmax(self, z):
        exp_z = np.exp(z - np.max(z, axis=0, keepdims=True))  # Numerical␣
↪stability
        return exp_z / np.sum(exp_z, axis=0, keepdims=True)

    def forward(self, X):
        """
        Forward pass through the network.
        X: input data of shape (input_size, batch_size)
        """
        # Layer 1 (hidden layer)
        self.Z1 = np.dot(self.W1, X) + self.b1  # (hidden_size, batch_size)
        self.A1 = self.relu(self.Z1)  # Apply ReLU activation

        ## ADD ANOTHER HIDDEN LAYER IN YOUR TAKE HOME EXERCISE
        self.Z3 = np.dot(self.W3, self.A1) + self.b3  # (hidden_size,␣
↪batch_size)
        self.A3 = self.relu(self.Z3)  # Apply ReLU activation

        # Layer 2 (output layer)
        self.Z2 = np.dot(self.W2, self.A3) + self.b2  # (output_size,␣
↪batch_size)
        self.A2 = self.softmax(self.Z2)  # Apply softmax activation
        return self.A2

    def compute_loss(self, A2, y):
        """
        Compute cross-entropy loss.
        A2: output from softmax, shape (output_size, batch_size)
        y: true labels, shape (batch_size,)
        """
        m = y.shape[0]  # batch size
        epsilon = 1e-10  # Small value to prevent log(0)
        log_likelihood = -np.log(A2[y, range(m)] + epsilon)

        loss = np.sum(log_likelihood) / m
        return loss

    def backward(self, X, y, learning_rate=0.01, clip_threshold=1.0):
        """
        Perform backward propagation and update weights, with gradient clipping.
        X: input data of shape (input_size, batch_size)
        y: true labels of shape (batch_size,)
        clip_threshold: the maximum allowed value for gradients, used to clip␣
↪them
        """
        m = X.shape[1]  # Batch size

        # Gradient of the loss w.r.t. Z2
        dZ2 = self.A2  # Softmax probabilities
        dZ2[y, range(m)] -= 1  # Subtract 1 from the correct class probabilities
        dZ2 /= m

        # Gradients for W2 and b2
        dW2 = np.dot(dZ2, self.A3.T)  # (output_size, hidden_size)
        db2 = np.sum(dZ2, axis=1, keepdims=True)  # (output_size, 1)

        # Gradients for the second hidden layer
        dA3 = np.dot(self.W2.T, dZ2)  # (hidden_size, batch_size)
        dZ3 = dA3 * self.relu_derivative(self.Z3)  # Backprop through ReLU

        # Gradients for W3 and b3
        dW3 = np.dot(dZ3, self.A1.T)  # (hidden_size, hidden_size)
        db3 = np.sum(dZ3, axis=1, keepdims=True)  # (hidden_size, 1)

        # Gradients for the first hidden layer
        dA1 = np.dot(self.W3.T, dZ3)  # (hidden_size, batch_size)
        dZ1 = dA1 * self.relu_derivative(self.Z1)  # Backprop through ReLU

        # Gradients for W1 and b1
        dW1 = np.dot(dZ1, X.T)  # (hidden_size, input_size)
        db1 = np.sum(dZ1, axis=1, keepdims=True)  # (hidden_size, 1)

        # Clip gradients to avoid exploding gradients
        dW1 = np.clip(dW1, -clip_threshold, clip_threshold)
        db1 = np.clip(db1, -clip_threshold, clip_threshold)
        dW2 = np.clip(dW2, -clip_threshold, clip_threshold)
        db2 = np.clip(db2, -clip_threshold, clip_threshold)
        dW3 = np.clip(dW3, -clip_threshold, clip_threshold)
        db3 = np.clip(db3, -clip_threshold, clip_threshold)

        # Update weights and biases
        self.W1 -= learning_rate * dW1
        self.b1 -= learning_rate * db1
        self.W2 -= learning_rate * dW2
        self.b2 -= learning_rate * db2
        self.W3 -= learning_rate * dW3
        self.b3 -= learning_rate * db3
```

```
    def train(self, train_loader, epochs=100, learning_rate=0.01,␣
↪step_size=100, gamma=0.5, patience = 5, min_delta=0.01):
        """
        Train the network with step decay for the learning rate.

        X_train: input data, shape (input_size, batch_size)
        y_train: true labels, shape (batch_size,)
        epochs: number of training epochs
        learning_rate: initial learning rate
        step_size: number of epochs after which to reduce the learning rate
        gamma: factor by which to reduce the learning rate (e.g., 0.1 means 90%␣
↪reduction)
        """
        losses = []
        current_lr = learning_rate
        epochs_without_improvement = 0

        for i in range(epochs):
            for batch_idx, (X_train, y_train) in enumerate(train_loader):
                X_train = X_train.view(X_train.size(0), -1).to(device).T  #␣
↪Flatten
                y_train = y_train.to(device)
                # Forward pass
                A2 = self.forward(X_train)

                # Compute the loss
                loss = self.compute_loss(A2, y_train)
                # if batch_idx % 10 == 0:
                #     print(f'Epoch {i+1}, Batch {batch_idx}, Loss: {loss}')

                # Backward pass
                self.backward(X_train, y_train, current_lr)

#              # # Step decay: Reduce learning rate every 'step_size' epochs
#              if (i + 1) % step_size == 0:
#                  current_lr *= gamma  # Reduce learning rate by a factor of␣
↪gamma
#                  print(f"Current learning rate: {current_lr}")

            print(f'Epoch {i+1}, Loss: {loss}')
            losses.append(loss)

            # Check if loss is close enough to the best loss for early stopping
            if loss < self.best_loss:
                # If the loss improved, update the best loss and reset the␣
↪counter
                self.best_loss = loss
                epochs_without_improvement = 0  # Reset the counter if we see␣
↪improvement
            elif abs(loss - self.best_loss) <= min_delta:
                # If loss is close to the best loss (within min_delta),␣
↪increment the counter
                epochs_without_improvement += 1

            # Learning rate scheduler, reduces learning rate (default: by 50%)␣
↪when loss doesn't improve for half of the patience for early stopping
            if epochs_without_improvement >= patience/2:
                current_lr *= gamma  # Reduce learning rate by a factor of gamma
                print(f"Current learning rate: {current_lr}")

            # Early stopping condition
            if epochs_without_improvement >= patience:
                print(f"No significant loss improvement for {patience}, early␣
↪stopping")
                break

        return losses
```

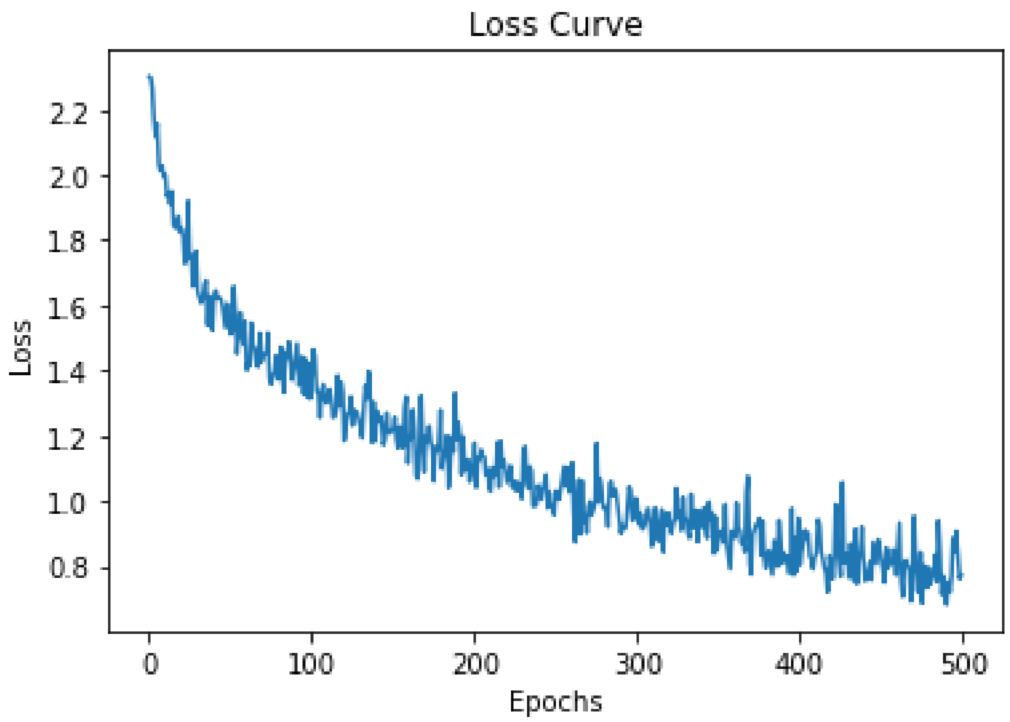
[31]:
```
# Load data into PyTorch DataLoader
train_loader = DataLoader(cifar_train, batch_size=sample_size, shuffle=True)
test_loader = DataLoader(cifar_test, batch_size=int(sample_size * 0.4),␣
 ↪shuffle=False)

# Fetch all data and labels for easier handling
X_train, y_train = next(iter(train_loader))
X_test, y_test = next(iter(test_loader))

print("Before Flattening")
print(f"Training data shape: {X_train.shape}")
print(f"Test data shape: {X_test.shape}")

# Reshape the images to 2D for the KNN algorithm
X_train = X_train.view(X_train.size(0), -1).to(device)  # Flatten
X_test = X_test.view(X_test.size(0), -1).to(device)
y_train = y_train.to(device)
y_test = y_test.to(device)

print("After Flattening")
print(f"Training data shape: {X_train.shape}")
print(f"Test data shape: {X_test.shape}")
```

```
Before Flattening
Training data shape: torch.Size([1000, 3, 32, 32])
Test data shape: torch.Size([400, 3, 32, 32])
After Flattening
Training data shape: torch.Size([1000, 3072])
Test data shape: torch.Size([400, 3072])
```

[32]:
```
input_size = 3072  # CIFAR-10 images are 32x32x3
hidden_size = 100  # Arbitrary hidden layer size
output_size = 10  # 10 classes for CIFAR-10

mlp = MLPClassifier(input_size, hidden_size, output_size)

losses = mlp.train(train_loader, epochs=500, learning_rate=0.1)
```

```
Epoch 1, Loss: 2.301742399766407
Epoch 2, Loss: 2.2987258243515516
Epoch 3, Loss: 2.2651327655462623
Epoch 4, Loss: 2.169229320542623
Epoch 5, Loss: 2.116859089516477
Epoch 6, Loss: 2.1587197555443454
Epoch 7, Loss: 2.0322556014602795
Epoch 8, Loss: 2.0130487128827075
Epoch 9, Loss: 2.028449100076992
Epoch 10, Loss: 1.9946963906680943
Epoch 11, Loss: 2.00015296949154
Epoch 12, Loss: 1.934084817233704
Epoch 13, Loss: 1.9443830706142098
Epoch 14, Loss: 1.9104803427860082
Epoch 15, Loss: 1.949687804462938
Epoch 16, Loss: 1.8480495574467255
Epoch 17, Loss: 1.8386119158799472
Epoch 18, Loss: 1.836574267113099
Epoch 19, Loss: 1.8729075929763814
Epoch 20, Loss: 1.8243571190183425
Epoch 21, Loss: 1.8371585373776227
Epoch 22, Loss: 1.81540176465582
Epoch 23, Loss: 1.727982177684409
Epoch 24, Loss: 1.7403078463206685
Epoch 25, Loss: 1.9207515646757165
Epoch 26, Loss: 1.7437017664209906
Epoch 27, Loss: 1.753798884274286
Epoch 28, Loss: 1.6589460900943882
Epoch 29, Loss: 1.665476255439959
Epoch 30, Loss: 1.7663645249741098
Epoch 31, Loss: 1.6405873444306915
Epoch 32, Loss: 1.625004701556212
Epoch 33, Loss: 1.6075927868158435
Epoch 34, Loss: 1.6218453431891062
Epoch 35, Loss: 1.6639117828077097
Epoch 36, Loss: 1.676811784926828
Epoch 37, Loss: 1.5389087399067398
Epoch 38, Loss: 1.6288938370909163
Epoch 39, Loss: 1.6095938487478827
Epoch 40, Loss: 1.5227788374120277
Epoch 41, Loss: 1.6301632409794267
Epoch 42, Loss: 1.6451389755760006
Epoch 43, Loss: 1.6221701334063487
Epoch 44, Loss: 1.6247101709793363
Epoch 45, Loss: 1.6216308441009928
Epoch 46, Loss: 1.603801701832449
Epoch 47, Loss: 1.5686786521197151
Epoch 48, Loss: 1.5316793293680682
Epoch 49, Loss: 1.605810643802435
Epoch 50, Loss: 1.5490969658324936
Epoch 51, Loss: 1.5122760226167247
Epoch 52, Loss: 1.517852019787517
Epoch 53, Loss: 1.6579383115812911
Epoch 54, Loss: 1.582959831603109
Epoch 55, Loss: 1.4509956182214634
Epoch 56, Loss: 1.507299083443307
Epoch 57, Loss: 1.5785712241551109
Epoch 58, Loss: 1.5091525991251806
Epoch 59, Loss: 1.4794493425511492
Epoch 60, Loss: 1.5516127680254517
Epoch 61, Loss: 1.4007543188205978
Epoch 62, Loss: 1.4240123478151117
Epoch 63, Loss: 1.4149561696727158
Epoch 64, Loss: 1.545979820506339
Epoch 65, Loss: 1.4730903552037136
Epoch 66, Loss: 1.4685547147298625
Epoch 67, Loss: 1.4215450911605307
Epoch 68, Loss: 1.4115417170837317
Epoch 69, Loss: 1.5146105543709
Epoch 70, Loss: 1.4252218508814
Epoch 71, Loss: 1.4439742952381565
Epoch 72, Loss: 1.4518095734589005
Epoch 73, Loss: 1.4542920269860444
Epoch 74, Loss: 1.51590431995354
Epoch 75, Loss: 1.3761040119827805
Epoch 76, Loss: 1.357974293545328
Epoch 77, Loss: 1.3841027517977527
Epoch 78, Loss: 1.3936010418160216
Epoch 79, Loss: 1.4458994530784197
Epoch 80, Loss: 1.3741274922503361
Epoch 81, Loss: 1.3793626281716524
Epoch 82, Loss: 1.472280245406704
Epoch 83, Loss: 1.3651089236443392
Epoch 84, Loss: 1.3321501770430044
Epoch 85, Loss: 1.4561543771837355
Epoch 86, Loss: 1.4388406799311069
Epoch 87, Loss: 1.4878840734935828
Epoch 88, Loss: 1.4323628447450014
Epoch 89, Loss: 1.3709731822731948
Epoch 90, Loss: 1.405163214783564
Epoch 91, Loss: 1.4471951575330082
Epoch 92, Loss: 1.4824388546150444
Epoch 93, Loss: 1.3548605553289754
Epoch 94, Loss: 1.3859420730686103
Epoch 95, Loss: 1.4435428149734018
Epoch 96, Loss: 1.327953607791409
Epoch 97, Loss: 1.43583632994461
Epoch 98, Loss: 1.3190942176830691
Epoch 99, Loss: 1.4226854311146044
Epoch 100, Loss: 1.313816829056107
Epoch 101, Loss: 1.3926417275135932
Epoch 102, Loss: 1.4642801440915305
Epoch 103, Loss: 1.444521696720662
Epoch 104, Loss: 1.3357463295850607
Epoch 105, Loss: 1.335099900088125
Epoch 106, Loss: 1.2558513515335565
Epoch 107, Loss: 1.3336545046520296
Epoch 108, Loss: 1.357997439305942
Epoch 109, Loss: 1.312207418101807
Epoch 110, Loss: 1.299223396856978
Epoch 111, Loss: 1.3268129679421832
Epoch 112, Loss: 1.3419168937090857
Epoch 113, Loss: 1.313525439497353
Epoch 114, Loss: 1.2553916697638077
Epoch 115, Loss: 1.257898091385288
Epoch 116, Loss: 1.2764309651171435
Epoch 117, Loss: 1.384913243109796
Epoch 118, Loss: 1.2917337948115537
Epoch 119, Loss: 1.3652366290703588
Epoch 120, Loss: 1.3345265110146893
Epoch 121, Loss: 1.184547013446605
Epoch 122, Loss: 1.2341771561260049
Epoch 123, Loss: 1.2667584258083502
Epoch 124, Loss: 1.2720375968222608
Epoch 125, Loss: 1.3198750677148618
Epoch 126, Loss: 1.2311892014568926
Epoch 127, Loss: 1.27993494158858
Epoch 128, Loss: 1.272992154375114
Epoch 129, Loss: 1.2720671542636182
Epoch 130, Loss: 1.2461149481990001
Epoch 131, Loss: 1.2283710416520546
Epoch 132, Loss: 1.1940597668620971
Epoch 133, Loss: 1.2953897836933437
Epoch 134, Loss: 1.3549134364787467
Epoch 135, Loss: 1.3141192494608926
Epoch 136, Loss: 1.3973970952414372
Epoch 137, Loss: 1.3101880613004726
Epoch 138, Loss: 1.1789663965574655
Epoch 139, Loss: 1.3039675050805515
Epoch 140, Loss: 1.183327883340539
Epoch 141, Loss: 1.2753448702732686
Epoch 142, Loss: 1.2435802089229575
Epoch 143, Loss: 1.2591968284304913
Epoch 144, Loss: 1.2399536275229506
Epoch 145, Loss: 1.171391384952114
Epoch 146, Loss: 1.1894869068835097
Epoch 147, Loss: 1.2695099432886645
Epoch 148, Loss: 1.2114568865693491
Epoch 149, Loss: 1.2078169643333505
Epoch 150, Loss: 1.2147311443228073
Epoch 151, Loss: 1.223106204686455
Epoch 152, Loss: 1.2577898287438234
Epoch 153, Loss: 1.1886879208401464
Epoch 154, Loss: 1.1612014412679084
Epoch 155, Loss: 1.2279409295059833
Epoch 156, Loss: 1.2120643564605338
Epoch 157, Loss: 1.1630064368233644
Epoch 158, Loss: 1.2851268277584034
Epoch 159, Loss: 1.3189887642459006
Epoch 160, Loss: 1.1156018220025201
Epoch 161, Loss: 1.1931823772698216
Epoch 162, Loss: 1.1916329246968875
Epoch 163, Loss: 1.2802356695231678
Epoch 164, Loss: 1.215420851651457
Epoch 165, Loss: 1.0788182019712222
Epoch 166, Loss: 1.0680901346565543
Epoch 167, Loss: 1.1578624527500483
Epoch 168, Loss: 1.3220683408030716
Epoch 169, Loss: 1.1919695997454112
Epoch 170, Loss: 1.0883034094144568
Epoch 171, Loss: 1.1981618255621818
Epoch 172, Loss: 1.181405506134321
Epoch 173, Loss: 1.2291427890700626
Epoch 174, Loss: 1.183834231912675
Epoch 175, Loss: 1.1561486070544624
Epoch 176, Loss: 1.0607774647564754
Epoch 177, Loss: 1.166194078877635
Epoch 178, Loss: 1.1516521285704124
Epoch 179, Loss: 1.1787199641434418
Epoch 180, Loss: 1.2791163874371003
Epoch 181, Loss: 1.0995300341761882
Epoch 182, Loss: 1.1477419954842334
Epoch 183, Loss: 1.1368024041757376
Epoch 184, Loss: 1.2001004206984462
Epoch 185, Loss: 1.0402258620305525
Epoch 186, Loss: 1.0924701323962034
Epoch 187, Loss: 1.1954648246928437
Epoch 188, Loss: 1.151598805413023
Epoch 189, Loss: 1.3309799469021844
Epoch 190, Loss: 1.1867205677496049
Epoch 191, Loss: 1.2412930314427904
Epoch 192, Loss: 1.2002152822546974
Epoch 193, Loss: 1.0811227052424996
Epoch 194, Loss: 1.1958787632121621
Epoch 195, Loss: 1.0950921465237475
Epoch 196, Loss: 1.104379476061772
Epoch 197, Loss: 1.127205347026063
Epoch 198, Loss: 1.0597068754448238
Epoch 199, Loss: 1.1044515644922848
Epoch 200, Loss: 1.0859652596403917
Epoch 201, Loss: 1.1778090691655956
Epoch 202, Loss: 1.040981435436192
Epoch 203, Loss: 1.1320293717200602
Epoch 204, Loss: 1.1257470356101316
Epoch 205, Loss: 1.1572404349390912
Epoch 206, Loss: 1.13902639893613
Epoch 207, Loss: 1.1345362543574522
Epoch 208, Loss: 1.079977799274282
Epoch 209, Loss: 1.0979163964073055
Epoch 210, Loss: 1.035226243865151
Epoch 211, Loss: 1.0302667815801942
Epoch 212, Loss: 1.101775747343716
Epoch 213, Loss: 1.0765642410091538
Epoch 214, Loss: 1.0889785848972855
Epoch 215, Loss: 1.1798274383475036
Epoch 216, Loss: 1.0423824679377407
Epoch 217, Loss: 1.1832516949542768
Epoch 218, Loss: 1.1071097317192502
Epoch 219, Loss: 1.1246198391455977
Epoch 220, Loss: 1.0919310921896204
Epoch 221, Loss: 1.0576022793383493
Epoch 222, Loss: 1.0712565405279406
Epoch 223, Loss: 1.107996965275911
Epoch 224, Loss: 1.0425087914482334
Epoch 225, Loss: 1.0621254066827235
Epoch 226, Loss: 1.0327938847236842
Epoch 227, Loss: 1.058905330328492
Epoch 228, Loss: 1.031591821655238
Epoch 229, Loss: 1.0528670233810224
Epoch 230, Loss: 1.0068963326872102
Epoch 231, Loss: 1.148407859726818
Epoch 232, Loss: 1.1663560856529491
Epoch 233, Loss: 1.057393583110898
Epoch 234, Loss: 1.0377601108970909
Epoch 235, Loss: 1.1057220997364552
Epoch 236, Loss: 1.094163546159485
Epoch 237, Loss: 0.9844226950777109
Epoch 238, Loss: 0.9845318023891018
Epoch 239, Loss: 0.9876639770510228
Epoch 240, Loss: 1.0484831812202184
Epoch 241, Loss: 1.0242252867119654
Epoch 242, Loss: 1.005397598135603
Epoch 243, Loss: 1.0414063516690777
Epoch 244, Loss: 1.0599241964705821
Epoch 245, Loss: 1.0805756545110679
Epoch 246, Loss: 0.9781468665772308
Epoch 247, Loss: 1.02553921860279
Epoch 248, Loss: 0.9852056176991513
Epoch 249, Loss: 0.979457716972761
Epoch 250, Loss: 0.9570250740499959
Epoch 251, Loss: 1.0307183580568409
Epoch 252, Loss: 1.0207307998347275
Epoch 253, Loss: 1.0054601561211165
Epoch 254, Loss: 1.0390877314510263
Epoch 255, Loss: 1.0719487096097409
Epoch 256, Loss: 1.1049161588779717
Epoch 257, Loss: 1.0455582698602028
Epoch 258, Loss: 1.1022045936151876
Epoch 259, Loss: 1.0279621406805508
Epoch 260, Loss: 1.0333915356713583
Epoch 261, Loss: 1.118414426701692
Epoch 262, Loss: 0.9648508517263683
Epoch 263, Loss: 0.8736545119829325
Epoch 264, Loss: 1.0064930848283795
Epoch 265, Loss: 1.065569366708511
Epoch 266, Loss: 0.8985351115652429
Epoch 267, Loss: 1.0625963861090608
Epoch 268, Loss: 0.9353373932465581
Epoch 269, Loss: 0.9566365146534279
Epoch 270, Loss: 0.9021756904190286
Epoch 271, Loss: 0.9797311492389272
Epoch 272, Loss: 0.9978571820057955
Epoch 273, Loss: 0.9568438110416259
Epoch 274, Loss: 0.9723731865305245
Epoch 275, Loss: 0.98685156332957
Epoch 276, Loss: 1.1759907645948358
Epoch 277, Loss: 0.9980109859153322
Epoch 278, Loss: 1.0696445751953674
Epoch 279, Loss: 1.0488637262470328
Epoch 280, Loss: 0.9816839418967788
Epoch 281, Loss: 0.9753316293799176
Epoch 282, Loss: 0.9824640071006215
Epoch 283, Loss: 0.9210488253323856
Epoch 284, Loss: 1.0526935597866294
Epoch 285, Loss: 1.0603601423489921
Epoch 286, Loss: 1.0144590146670558
Epoch 287, Loss: 1.0370822517900433
Epoch 288, Loss: 1.0276103956394316
Epoch 289, Loss: 0.9853709471964455
Epoch 290, Loss: 0.9435833972410713
Epoch 291, Loss: 0.9005692379478002
Epoch 292, Loss: 0.925178281897003
Epoch 293, Loss: 0.9142469833576413
Epoch 294, Loss: 0.9178039089922474
Epoch 295, Loss: 0.9863087704638103
Epoch 296, Loss: 1.0437628647195647
Epoch 297, Loss: 1.0309269285696894
Epoch 298, Loss: 0.939344893814238
Epoch 299, Loss: 0.9465533868484983
Epoch 300, Loss: 1.0083256092021957
Epoch 301, Loss: 0.9869358760139192
Epoch 302, Loss: 0.9282316213886316
Epoch 303, Loss: 0.9609521275262537
Epoch 304, Loss: 0.9135555531172086
Epoch 305, Loss: 0.9194398911939721
Epoch 306, Loss: 0.9362922042659918
Epoch 307, Loss: 0.9759734287813746
Epoch 308, Loss: 0.9757483540015052
Epoch 309, Loss: 0.9281470908131139
Epoch 310, Loss: 0.8607386721639102
Epoch 311, Loss: 0.9355830635684496
Epoch 312, Loss: 0.9789006871050776
Epoch 313, Loss: 0.9009257199229124
Epoch 314, Loss: 0.8764526737775983
Epoch 315, Loss: 0.9352772413024435
Epoch 316, Loss: 0.9742730017912663
Epoch 317, Loss: 0.8429400117300117
Epoch 318, Loss: 0.9587221344792229
Epoch 319, Loss: 0.9268250661138158
Epoch 320, Loss: 0.9633449057683745
Epoch 321, Loss: 0.9295617077032211
Epoch 322, Loss: 0.9002901336887664
Epoch 323, Loss: 0.9297076668692635
Epoch 324, Loss: 0.9463954019282992
Epoch 325, Loss: 1.039370758838524
Epoch 326, Loss: 0.9464730037599501
Epoch 327, Loss: 0.9666092737661754
Epoch 328, Loss: 0.9096551117215342
Epoch 329, Loss: 1.0112596520055026
Epoch 330, Loss: 0.9423848569609089
Epoch 331, Loss: 0.9427847266932443
Epoch 332, Loss: 0.9018659078164838
Epoch 333, Loss: 0.8857151878004874
Epoch 334, Loss: 1.0214215397863378
Epoch 335, Loss: 0.9179276946905327
Epoch 336, Loss: 0.9176391177686563
Epoch 337, Loss: 0.9723824363336577
Epoch 338, Loss: 0.8747299009585576
Epoch 339, Loss: 0.9692604507966347
Epoch 340, Loss: 0.9364585936834785
Epoch 341, Loss: 0.9180928552971297
Epoch 342, Loss: 0.9792237065496168
Epoch 343, Loss: 0.9021169391913844
Epoch 344, Loss: 0.8854563092740612
Epoch 345, Loss: 0.9976223510025413
Epoch 346, Loss: 0.9441992740617868
Epoch 347, Loss: 0.9629839558215738
Epoch 348, Loss: 0.8389141305885919
Epoch 349, Loss: 0.951889566158605
Epoch 350, Loss: 0.8485229771509487
Epoch 351, Loss: 0.888635407773432
Epoch 352, Loss: 0.9053185699317362
Epoch 353, Loss: 0.9082747013269197
Epoch 354, Loss: 0.872935951585215
Epoch 355, Loss: 0.989973839926412
Epoch 356, Loss: 0.8611380741959777
Epoch 357, Loss: 0.8251320633395847
Epoch 358, Loss: 0.7911992252438685
Epoch 359, Loss: 0.9068605220986854
Epoch 360, Loss: 0.9036529641493212
Epoch 361, Loss: 0.8867043723189142
Epoch 362, Loss: 0.9945489439122059
Epoch 363, Loss: 0.91512712442518
Epoch 364, Loss: 0.9086149635100071
Epoch 365, Loss: 0.8686811560599683
Epoch 366, Loss: 0.8814465181329613
Epoch 367, Loss: 0.8436974116585454
Epoch 368, Loss: 1.0257268190455928
Epoch 369, Loss: 1.0758815606705403
Epoch 370, Loss: 0.8052327489884323
Epoch 371, Loss: 0.7747907575010935
Epoch 372, Loss: 0.8999329215786859
Epoch 373, Loss: 0.9117979686688161
Epoch 374, Loss: 0.9275892384388045
Epoch 375, Loss: 0.9318416215504798
Epoch 376, Loss: 0.9473561309522356
Epoch 377, Loss: 0.8342041731644798
Epoch 378, Loss: 0.9379879743545065
Epoch 379, Loss: 0.8237380610117772
Epoch 380, Loss: 0.7927620731553244
Epoch 381, Loss: 0.8407433831995776
Epoch 382, Loss: 0.8476587973238127
Epoch 383, Loss: 0.8102563383782289
Epoch 384, Loss: 0.7738533263361702
Epoch 385, Loss: 0.8784540795048661
Epoch 386, Loss: 0.8532304032346775
Epoch 387, Loss: 0.7984176127334877
Epoch 388, Loss: 0.9402519992159175
Epoch 389, Loss: 0.8848700512376912
Epoch 390, Loss: 0.8075291063344698
Epoch 391, Loss: 0.8410653288757939
Epoch 392, Loss: 0.8175042003801672
Epoch 393, Loss: 0.8318733863798822
Epoch 394, Loss: 0.779177455628829
Epoch 395, Loss: 0.7793989425519129
Epoch 396, Loss: 0.9771038319639929
Epoch 397, Loss: 0.7725532570540907
Epoch 398, Loss: 0.8849226608985531
Epoch 399, Loss: 0.7844221609166576
Epoch 400, Loss: 0.947852809187531
Epoch 401, Loss: 0.8031279076074149
Epoch 402, Loss: 0.8450863385143119
Epoch 403, Loss: 0.870613832839962
Epoch 404, Loss: 0.9102791472854184
Epoch 405, Loss: 0.8831181183008943
Epoch 406, Loss: 0.9026007748045889
Epoch 407, Loss: 0.8296099577552659
Epoch 408, Loss: 0.8231491287347161
Epoch 409, Loss: 0.7911171692847229
Epoch 410, Loss: 0.8264170610058926
Epoch 411, Loss: 0.8399474312023777
Epoch 412, Loss: 0.943058879249316
Epoch 413, Loss: 0.9268456210960466
Epoch 414, Loss: 0.84181783633122
Epoch 415, Loss: 0.8221154185930063
Epoch 416, Loss: 0.7929323502397074
Epoch 417, Loss: 0.7807807874305843
Epoch 418, Loss: 0.7202621940728845
Epoch 419, Loss: 0.8318155350139566
Epoch 420, Loss: 0.805799214757113
Epoch 421, Loss: 0.7607972405027283
Epoch 422, Loss: 0.8270693781558056
Epoch 423, Loss: 0.9896543041143796
Epoch 424, Loss: 0.8523191148466032
Epoch 425, Loss: 0.8110632209966724
Epoch 426, Loss: 0.7679781108351197
Epoch 427, Loss: 1.0569460835020217
Epoch 428, Loss: 0.8097730355821409
Epoch 429, Loss: 0.8305314381810367
Epoch 430, Loss: 0.8382135851999547
Epoch 431, Loss: 0.7651431507070172
Epoch 432, Loss: 0.8684283881570097
Epoch 433, Loss: 0.8174664945230652
Epoch 434, Loss: 0.7540740791707856
Epoch 435, Loss: 0.9139445870678855
Epoch 436, Loss: 0.745498644896541
Epoch 437, Loss: 0.8154904336637401
Epoch 438, Loss: 0.8439546142730492
Epoch 439, Loss: 0.9212144219862259
Epoch 440, Loss: 0.8237306109723685
Epoch 441, Loss: 0.7544917316093547
Epoch 442, Loss: 0.7606149975826573
Epoch 443, Loss: 0.7837777081064662
Epoch 444, Loss: 0.8131612833413974
Epoch 445, Loss: 0.757338153949293
Epoch 446, Loss: 0.8739152305795205
Epoch 447, Loss: 0.8423918112372836
Epoch 448, Loss: 0.8080130566011744
Epoch 449, Loss: 0.8815755594176387
Epoch 450, Loss: 0.8605590725488584
Epoch 451, Loss: 0.8190601553398258
Epoch 452, Loss: 0.789904855977573
Epoch 453, Loss: 0.7499046419265353
Epoch 454, Loss: 0.8562250806886125
Epoch 455, Loss: 0.7930468628444513
Epoch 456, Loss: 0.8092923113925251
Epoch 457, Loss: 0.8075882198178151
Epoch 458, Loss: 0.8510401079422723
Epoch 459, Loss: 0.8427639553820561
Epoch 460, Loss: 0.7727640963086553
Epoch 461, Loss: 0.8529946845711222
Epoch 462, Loss: 0.9323909127376578
Epoch 463, Loss: 0.7543358419994154
Epoch 464, Loss: 0.7060659367990513
Epoch 465, Loss: 0.8115809943912812
Epoch 466, Loss: 0.81777525183977
Epoch 467, Loss: 0.7983047775814616
Epoch 468, Loss: 0.7948622800858948
Epoch 469, Loss: 0.6918052624312264
Epoch 470, Loss: 0.7903389765351303
Epoch 471, Loss: 0.9532818339742872
Epoch 472, Loss: 0.8232709398898237
Epoch 473, Loss: 0.7233155508903792
Epoch 474, Loss: 0.718859816500421
Epoch 475, Loss: 0.8419226692555773
Epoch 476, Loss: 0.6861652543589969
Epoch 477, Loss: 0.8022268634475412
Epoch 478, Loss: 0.7840280526303336
Epoch 479, Loss: 0.7343857734045943
Epoch 480, Loss: 0.7962861555746924
Epoch 481, Loss: 0.7436102410115762
Epoch 482, Loss: 0.7681962534021097
Epoch 483, Loss: 0.7761799488480472
Epoch 484, Loss: 0.8329531081881453
Epoch 485, Loss: 0.7524294566090339
Epoch 486, Loss: 0.9399504196260888
Epoch 487, Loss: 0.8471029787578215
Epoch 488, Loss: 0.7133431710233386
Epoch 489, Loss: 0.769212411029913
Epoch 490, Loss: 0.7089808382171957
Epoch 491, Loss: 0.6819742338510147
Epoch 492, Loss: 0.7519122448895273
Epoch 493, Loss: 0.7191923890014856
Epoch 494, Loss: 0.7693372959397035
Epoch 495, Loss: 0.8869321547688072
Epoch 496, Loss: 0.8436311512471171
Epoch 497, Loss: 0.9068931663314289
Epoch 498, Loss: 0.8267610107738744
Epoch 499, Loss: 0.7611086605937817
Epoch 500, Loss: 0.7729428492626437
```

[33]:
```
plt.plot(losses)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Loss Curve')
plt.show()
```


Loss Curve
Loss
Epochs
2.2
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0
100
200
300
400
500

```
[ ]:
```