classifiers_step_by_step

September 23, 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)
```

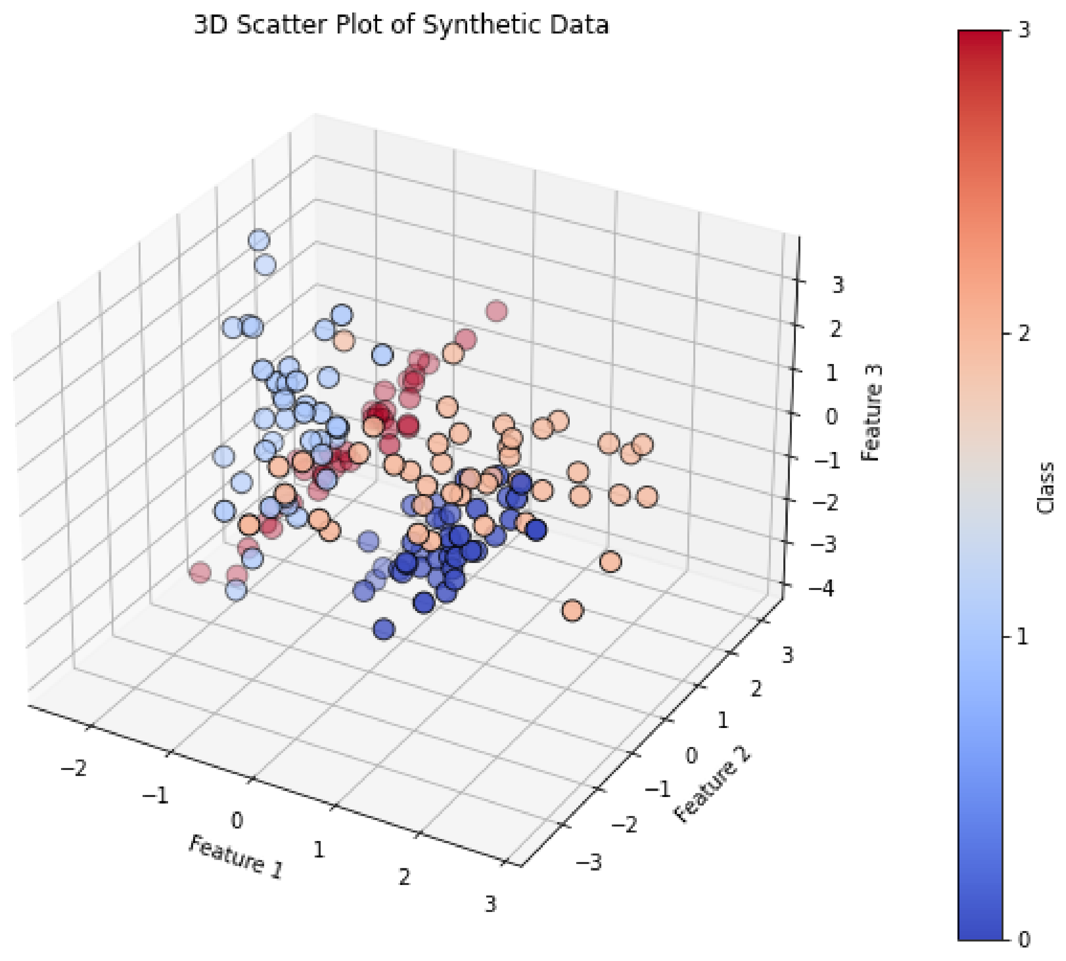
1.8.1 OKAY LETS GO A HEAD AND TRAIN OUR MODEL !!

```
[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)
```

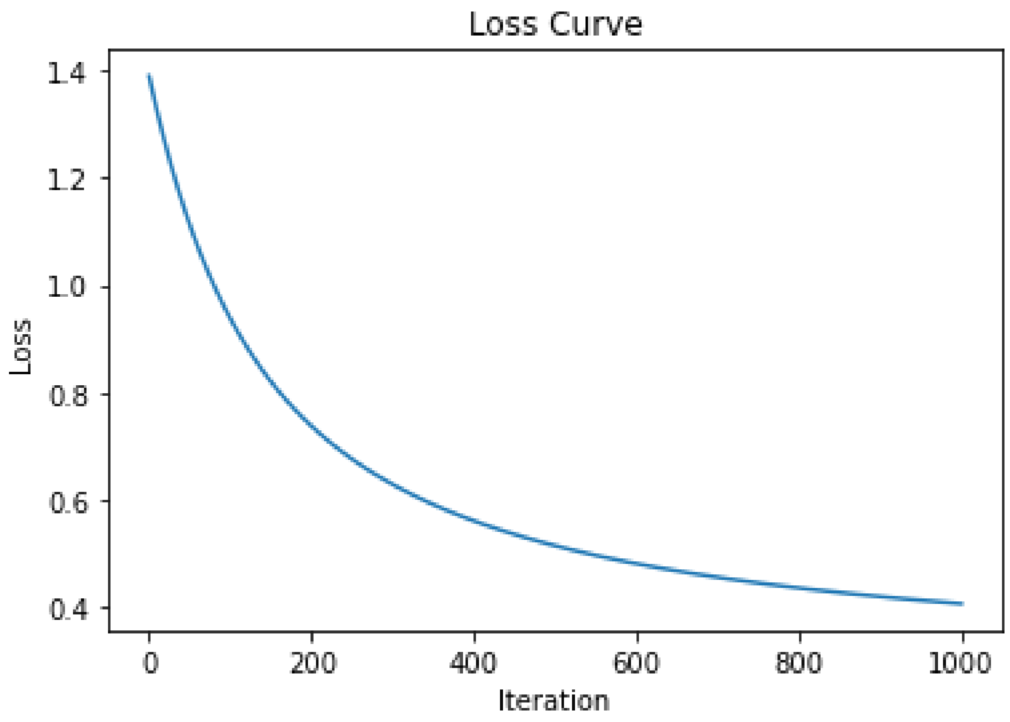
[11]:
```
if multiclass:
    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection='3d')
    # Convert one-hot encoding to class labels for plotting
    y_train_labels = np.argmax(y_train_syn, axis=1)
    # Use the 3 features for the scatter plot
    scatter = ax.scatter(X_train_syn[:, 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: 1.3897
Iteration 100/1000, Loss: 0.9375
Iteration 200/1000, Loss: 0.7372
Iteration 300/1000, Loss: 0.6284
Iteration 400/1000, Loss: 0.5605
Iteration 500/1000, Loss: 0.5143
Iteration 600/1000, Loss: 0.4808
Iteration 700/1000, Loss: 0.4554
Iteration 800/1000, Loss: 0.4355
Iteration 900/1000, Loss: 0.4194
Time Taken Using Batch gradient descent :0.11577463150024414
```

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



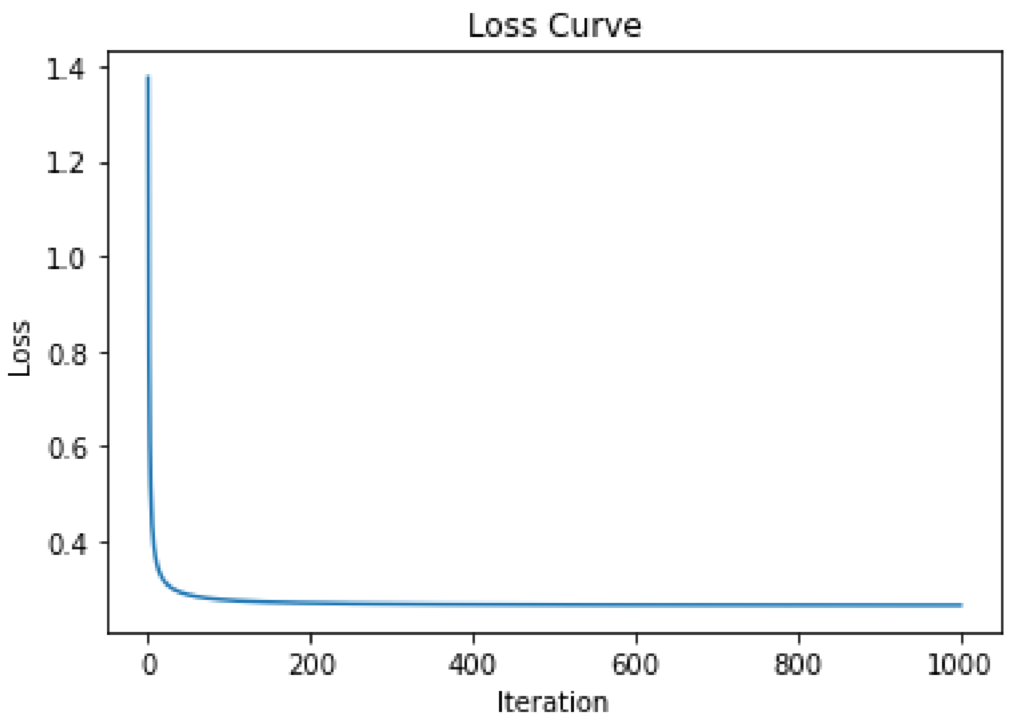
[14]:
```python
# Add bias term to test data
X_test_bias = add_bias_term(X_test_syn)

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: 1.3764
Iteration 100/1000, Loss: 0.2756
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 :4.846238374710083
```

```
[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
```

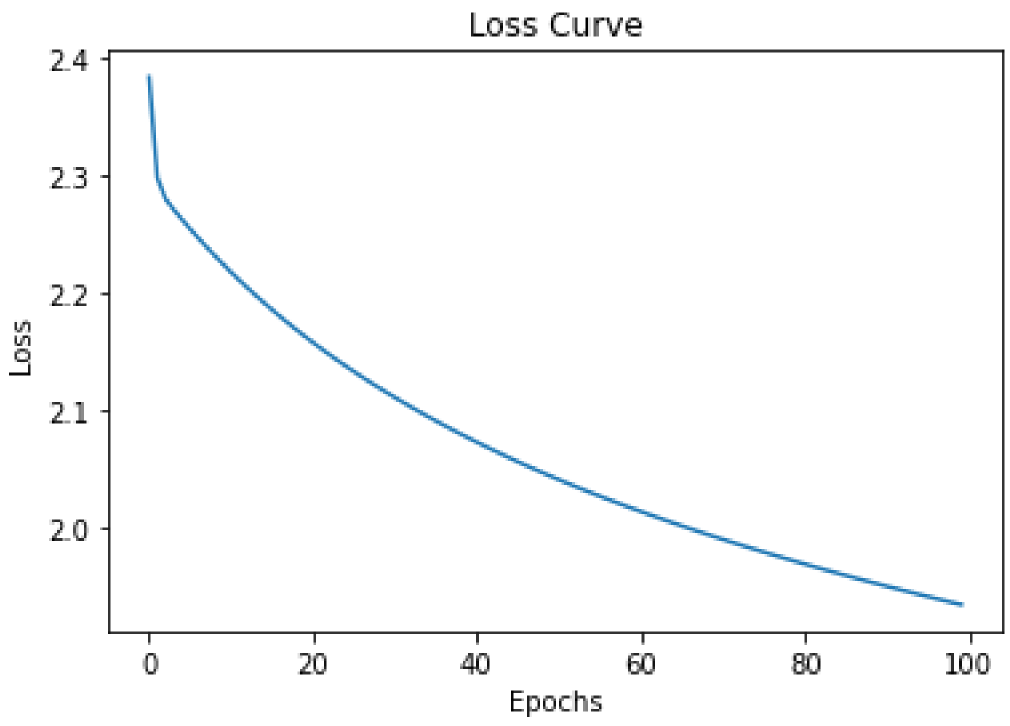
[24]:
```
print(f"Training data: {len(cifar_train)}")
print(f"Test data: {len(cifar_test)}")

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: 2.384246048289845
Epoch 2, Loss: 2.2985756331920193
Epoch 3, Loss: 2.2804714424255508
Epoch 4, Loss: 2.271323583364853
Epoch 5, Loss: 2.262966246654286
Epoch 6, Loss: 2.254897907845101
Epoch 7, Loss: 2.2470767328392607
Epoch 8, Loss: 2.2394885182435673
Epoch 9, Loss: 2.2321227561912567
Epoch 10, Loss: 2.224969835601894
Epoch 11, Loss: 2.218020650626824
Epoch 12, Loss: 2.2112665312413986
Epoch 13, Loss: 2.2046992195069817
Epoch 14, Loss: 2.1983108522664443
Epoch 15, Loss: 2.192093944614625
Epoch 16, Loss: 2.1860413734568254
Epoch 17, Loss: 2.1801463611915683
Epoch 18, Loss: 2.174402459633263
Epoch 19, Loss: 2.1688035342772087
Epoch 20, Loss: 2.163343748986846
Epoch 21, Loss: 2.1580175511631197
Epoch 22, Loss: 2.152819657438954
Epoch 23, Loss: 2.147745039927967
Epoch 24, Loss: 2.1427889130450968
Epoch 25, Loss: 2.1379467209074647
Epoch 26, Loss: 2.1332141253162664
Epoch 27, Loss: 2.1285869943144253
Epoch 28, Loss: 2.124061391310006
Epoch 29, Loss: 2.1196335647516955
Epoch 30, Loss: 2.115299938339911
Epoch 31, Loss: 2.11105710175505
Epoch 32, Loss: 2.106901801883013
Epoch 33, Loss: 2.102830934517214
Epoch 34, Loss: 2.0988415365158306
Epoch 35, Loss: 2.094930778392864
Epoch 36, Loss: 2.0910959573217194
Epoch 37, Loss: 2.0873344905302837
Epoch 38, Loss: 2.0836439090670167
Epoch 39, Loss: 2.0800218519180933
Epoch 40, Loss: 2.076466060456374
Epoch 41, Loss: 2.0729743732036807
Epoch 42, Loss: 2.0695447208886675
Epoch 43, Loss: 2.0661751217833597
Epoch 44, Loss: 2.062863677302259
Epoch 45, Loss: 2.0596085678487115
Epoch 46, Loss: 2.056408048894056
Epoch 47, Loss: 2.0532604472758185
Epoch 48, Loss: 2.05016415770202
Epoch 49, Loss: 2.047117639449354
Epoch 50, Loss: 2.0441194132437324
Epoch 51, Loss: 2.0411680583123495
Epoch 52, Loss: 2.0382622095970615
Epoch 53, Loss: 2.035400555119498
Epoch 54, Loss: 2.0325818334888917
Epoch 55, Loss: 2.0298048315441664
Epoch 56, Loss: 2.027068382122344
Epoch 57, Loss: 2.024371361945811
Epoch 58, Loss: 2.0217126896214572
Epoch 59, Loss: 2.019091323745122
Epoch 60, Loss: 2.016506261105193
Epoch 61, Loss: 2.0139565349796
Epoch 62, Loss: 2.011441213520763
Epoch 63, Loss: 2.0089593982234497
Epoch 64, Loss: 2.0065102224707574
Epoch 65, Loss: 2.004092850153763
Epoch 66, Loss: 2.001706474360645
Epoch 67, Loss: 1.9993503161313528
Epoch 68, Loss: 1.9970236232741212
Epoch 69, Loss: 1.9947256692403819
Epoch 70, Loss: 1.9924557520548039
Epoch 71, Loss: 1.9902131932974236
Epoch 72, Loss: 1.987997337134988
Epoch 73, Loss: 1.9858075493988159
Epoch 74, Loss: 1.9836432167066542
Epoch 75, Loss: 1.9815037456261388
Epoch 76, Loss: 1.9793885618776252
Epoch 77, Loss: 1.9772971095742895
Epoch 78, Loss: 1.975228850497511
Epoch 79, Loss: 1.9731832634056796
Epoch 80, Loss: 1.971159843374667
Epoch 81, Loss: 1.9691581011683197
Epoch 82, Loss: 1.967177562637403
Epoch 83, Loss: 1.965217768145545
Epoch 84, Loss: 1.963278272020786
Epoch 85, Loss: 1.9613586420314373
Epoch 86, Loss: 1.959458458885017
Epoch 87, Loss: 1.9575773157491032
Epoch 88, Loss: 1.9557148177930097
Epoch 89, Loss: 1.9538705817492514
Epoch 90, Loss: 1.9520442354938237
Epoch 91, Loss: 1.950235417644369
Epoch 92, Loss: 1.9484437771753664
Epoch 93, Loss: 1.9466689730495124
Epoch 94, Loss: 1.944910673864519
Epoch 95, Loss: 1.9431685575145887
Epoch 96, Loss: 1.9414423108658718
Epoch 97, Loss: 1.9397316294452394
Epoch 98, Loss: 1.9380362171417558
Epoch 99, Loss: 1.9363557859202476
Epoch 100, Loss: 1.9346900555464168
```

[26]:
```python
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:

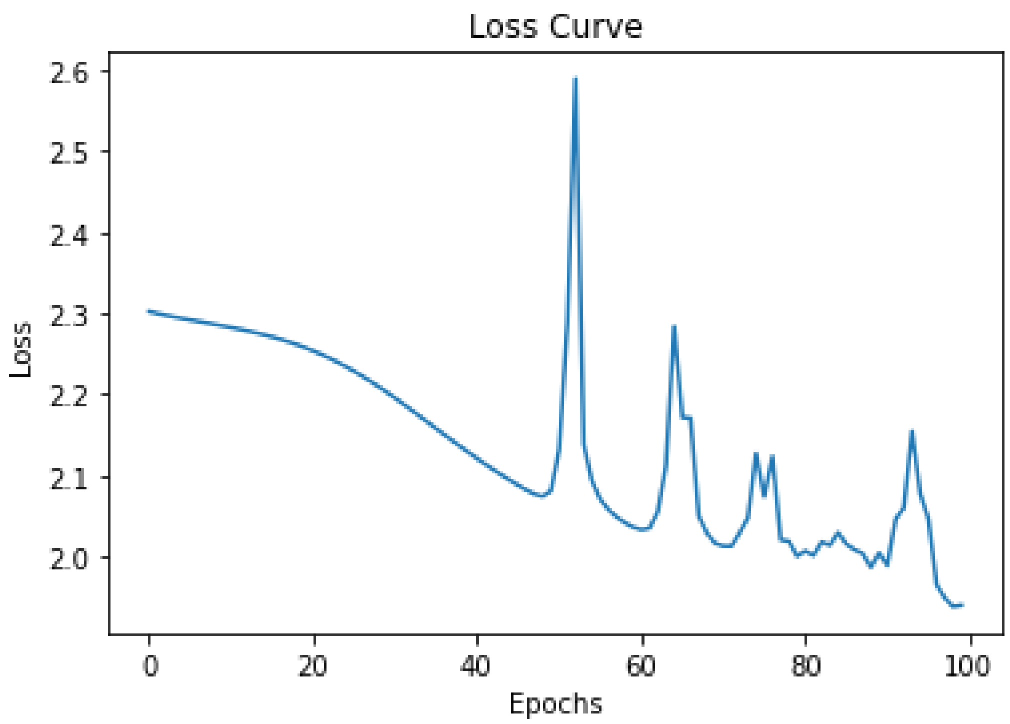
1. Input Layer: (3072 neurons, corresponding to image size 32x32x3 in CIFAR-10)
2. Hidden Layer 1: Fully connected, with a non-linear activation like ReLU.
3. Hidden Layer 2: Another fully connected layer (optional).
4. Output Layer: A fully connected layer with 10 neurons (for 10 classes) and softmax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

[27]:
```
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: 2.302176394750433
Epoch 2, Loss: 2.299709144344611
Epoch 3, Loss: 2.2975403577895057
Epoch 4, Loss: 2.295550989471332
Epoch 5, Loss: 2.293623768465235
Epoch 6, Loss: 2.291769990659246
Epoch 7, Loss: 2.289948936392972
Epoch 8, Loss: 2.2881111264243583
Epoch 9, Loss: 2.28623642479386
Epoch 10, Loss: 2.284339097982528
Epoch 11, Loss: 2.2824135505735383
Epoch 12, Loss: 2.280395955558015
Epoch 13, Loss: 2.2782627462465266
Epoch 14, Loss: 2.275985388262265
Epoch 15, Loss: 2.273532279664589
Epoch 16, Loss: 2.270874879401249
Epoch 17, Loss: 2.2679870229761083
Epoch 18, Loss: 2.2648460206487435
Epoch 19, Loss: 2.261430936332772
Epoch 20, Loss: 2.2577150485606943
Epoch 21, Loss: 2.253687947016365
Epoch 22, Loss: 2.2493374876850525
Epoch 23, Loss: 2.2446475121116825
Epoch 24, Loss: 2.2396105223213336
Epoch 25, Loss: 2.2342329889346195
Epoch 26, Loss: 2.228511592633883
Epoch 27, Loss: 2.222457395437618
Epoch 28, Loss: 2.2160805922009486
Epoch 29, Loss: 2.2094106660102826
Epoch 30, Loss: 2.202487591613427
Epoch 31, Loss: 2.1953400287184106
Epoch 32, Loss: 2.188015308100088
Epoch 33, Loss: 2.180548491671266
Epoch 34, Loss: 2.1729860335097837
Epoch 35, Loss: 2.1653840665219724
Epoch 36, Loss: 2.157782495744141
Epoch 37, Loss: 2.1502229787275957
Epoch 38, Loss: 2.1427515101069115
Epoch 39, Loss: 2.1353924381126586
Epoch 40, Loss: 2.128162140982986
Epoch 41, Loss: 2.121087838771742
Epoch 42, Loss: 2.114179830806891
Epoch 43, Loss: 2.1074469794357586
Epoch 44, Loss: 2.100895412428276
Epoch 45, Loss: 2.094533944861625
Epoch 46, Loss: 2.088382887924383
Epoch 47, Loss: 2.0825365233179243
Epoch 48, Loss: 2.0773658680659324
Epoch 49, Loss: 2.074605941109272
Epoch 50, Loss: 2.0814405587331226
Epoch 51, Loss: 2.132018827906251
Epoch 52, Loss: 2.28840696872497
Epoch 53, Loss: 2.588990716757046
Epoch 54, Loss: 2.1382616896356947
Epoch 55, Loss: 2.0936678350049425
Epoch 56, Loss: 2.0712625170085195
Epoch 57, Loss: 2.0583810842292922
Epoch 58, Loss: 2.049329407203853
Epoch 59, Loss: 2.042283742739378
Epoch 60, Loss: 2.0367498316744883
Epoch 61, Loss: 2.0335500191822304
Epoch 62, Loss: 2.035415159641085
Epoch 63, Loss: 2.05552197724038
Epoch 64, Loss: 2.112671389048445
Epoch 65, Loss: 2.283743434313785
Epoch 66, Loss: 2.171673130913981
Epoch 67, Loss: 2.170786627096445
Epoch 68, Loss: 2.0504352086221735
Epoch 69, Loss: 2.0292626228779027
Epoch 70, Loss: 2.0167703029974478
Epoch 71, Loss: 2.0134357348027057
Epoch 72, Loss: 2.0133326190857646
Epoch 73, Loss: 2.029889494985288
Epoch 74, Loss: 2.048331854023347
Epoch 75, Loss: 2.1273231158117687
Epoch 76, Loss: 2.0742715208005023
Epoch 77, Loss: 2.1234269080574166
Epoch 78, Loss: 2.0211145788631417
Epoch 79, Loss: 2.0192786131888902
Epoch 80, Loss: 2.00085592330856
Epoch 81, Loss: 2.007420654797926
Epoch 82, Loss: 2.001783537916504
Epoch 83, Loss: 2.0185779322429034
Epoch 84, Loss: 2.014254129925309
Epoch 85, Loss: 2.030093958959645
Epoch 86, Loss: 2.0158984334964116
Epoch 87, Loss: 2.0089424586902482
Epoch 88, Loss: 2.0042533485099767
Epoch 89, Loss: 1.986943376504626
Epoch 90, Loss: 2.005171280922642
Epoch 91, Loss: 1.9893797375251299
Epoch 92, Loss: 2.046593953020298
Epoch 93, Loss: 2.060507269238974
Epoch 94, Loss: 2.15438739029621
Epoch 95, Loss: 2.078002274188491
Epoch 96, Loss: 2.0466577621029383
Epoch 97, Loss: 1.9654912779622304
Epoch 98, Loss: 1.949284553613692
Epoch 99, Loss: 1.938746985925558
Epoch 100, Loss: 1.940502905993727
```

[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

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

        return losses
```

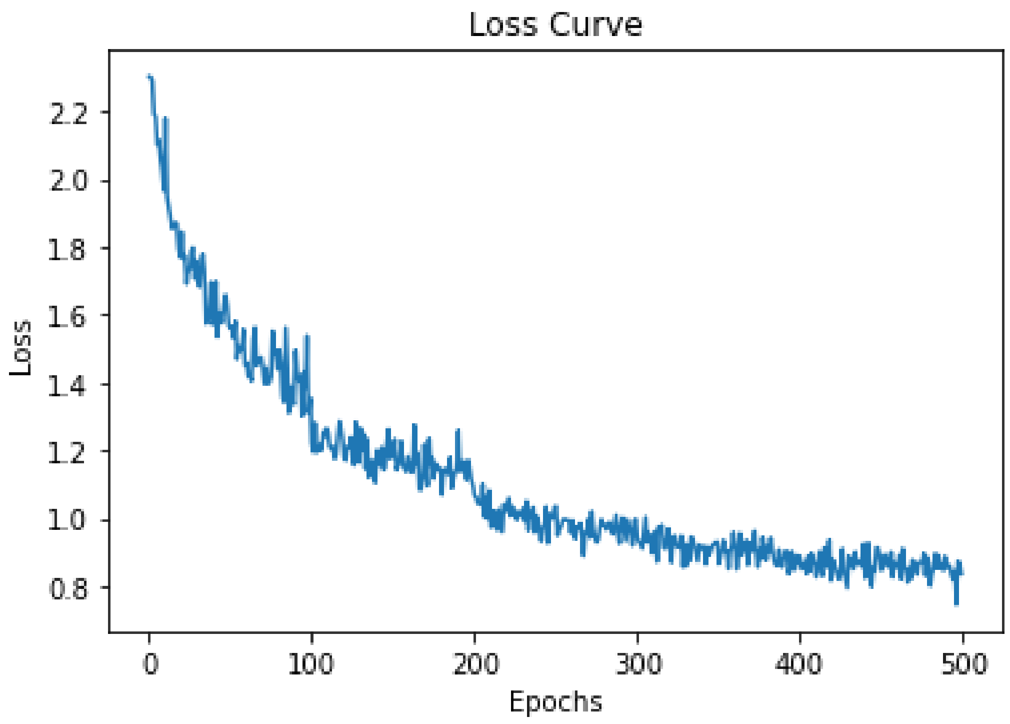
[31]:
```
# Load data into PyTorch DataLoader
train_loader = DataLoader(cifar_train, batch_size=sample_size, shuffle=True)
test_loader = DataLoader(cifar_test, batch_size=int(sample_size * 0.4),␣
↪shuffle=False)

# 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: 2.302221277525989
Epoch 2, Loss: 2.3003693202819617
Epoch 3, Loss: 2.288336618669395
Epoch 4, Loss: 2.1916840024813418
Epoch 5, Loss: 2.1848863276538966
Epoch 6, Loss: 2.1028822253047257
Epoch 7, Loss: 2.116766869401725
Epoch 8, Loss: 2.063026926645907
Epoch 9, Loss: 2.0277613140292776
Epoch 10, Loss: 1.9665150464602366
Epoch 11, Loss: 2.179432743116003
Epoch 12, Loss: 1.948192188978906
Epoch 13, Loss: 1.9201844420047838
Epoch 14, Loss: 1.8899755209378477
Epoch 15, Loss: 1.8532098206504237
Epoch 16, Loss: 1.8596676235443526
Epoch 17, Loss: 1.871657162984942
Epoch 18, Loss: 1.8693836073205572
Epoch 19, Loss: 1.7965937768043956
Epoch 20, Loss: 1.7697573012581582
Epoch 21, Loss: 1.8440913567538755
Epoch 22, Loss: 1.7651822027028703
Epoch 23, Loss: 1.7729310280788244
Epoch 24, Loss: 1.6915928220680343
Epoch 25, Loss: 1.7336011974642398
Epoch 26, Loss: 1.7332549193337468
Epoch 27, Loss: 1.7884639545668117
Epoch 28, Loss: 1.797319287032372
Epoch 29, Loss: 1.7075898451222027
Epoch 30, Loss: 1.7583403388438044
Epoch 31, Loss: 1.7037639550343646
Epoch 32, Loss: 1.6831116793330916
Epoch 33, Loss: 1.7619592985531207
Epoch 34, Loss: 1.7789716023745776
Epoch 35, Loss: 1.7292548036119408
Epoch 36, Loss: 1.5741201135223104
Epoch 37, Loss: 1.5967141546353503
Epoch 38, Loss: 1.5976834118019236
Epoch 39, Loss: 1.6965177032134686
Epoch 40, Loss: 1.5712816424772156
Epoch 41, Loss: 1.601366909749111
Epoch 42, Loss: 1.6984126802718726
Epoch 43, Loss: 1.5352388891097428
Epoch 44, Loss: 1.6068968974374889
Epoch 45, Loss: 1.5826342098150248
Epoch 46, Loss: 1.5764021736432654
Epoch 47, Loss: 1.6309557278918483
Epoch 48, Loss: 1.6587005164725466
Epoch 49, Loss: 1.6280647461160376
Epoch 50, Loss: 1.568102427319624
Epoch 51, Loss: 1.5593342167481095
Epoch 52, Loss: 1.565834974816124
Epoch 53, Loss: 1.530741394137332
Epoch 54, Loss: 1.582089313420318
Epoch 55, Loss: 1.469622846097493
Epoch 56, Loss: 1.5070087866612047
Epoch 57, Loss: 1.4944682484865006
Epoch 58, Loss: 1.5068976887890078
Epoch 59, Loss: 1.5578191857815213
Epoch 60, Loss: 1.4513777778216406
Epoch 61, Loss: 1.4584672813536355
Epoch 62, Loss: 1.4196204116655256
Epoch 63, Loss: 1.418075735645247
Epoch 64, Loss: 1.4045632169667033
Epoch 65, Loss: 1.5277598127445098
Epoch 66, Loss: 1.5631393838253163
Epoch 67, Loss: 1.4503479719066164
Epoch 68, Loss: 1.4667803858890842
Epoch 69, Loss: 1.469265218772185
Epoch 70, Loss: 1.4739108897724242
Epoch 71, Loss: 1.4407880275690745
Epoch 72, Loss: 1.3971220660143304
Epoch 73, Loss: 1.4439261319568777
Epoch 74, Loss: 1.3963443269008406
Epoch 75, Loss: 1.4000517779669976
Epoch 76, Loss: 1.4177192439290107
Epoch 77, Loss: 1.552119414545338
Epoch 78, Loss: 1.4839265940311377
Epoch 79, Loss: 1.4930839090544943
Epoch 80, Loss: 1.4400438490347685
Epoch 81, Loss: 1.4966840578953018
Epoch 82, Loss: 1.4581459363454323
Epoch 83, Loss: 1.3579941100517148
Epoch 84, Loss: 1.3417743932958774
Epoch 85, Loss: 1.562304987410337
Epoch 86, Loss: 1.4352934392928893
Epoch 87, Loss: 1.3110678488166445
Epoch 88, Loss: 1.3876642272034128
Epoch 89, Loss: 1.3492341793976907
Epoch 90, Loss: 1.3358213768329432
Epoch 91, Loss: 1.4950093382021343
Epoch 92, Loss: 1.429973912742209
Epoch 93, Loss: 1.4054227318771748
Epoch 94, Loss: 1.4187682973013584
Epoch 95, Loss: 1.3015224528296467
Epoch 96, Loss: 1.4290145486278345
Epoch 97, Loss: 1.385333859781768
Epoch 98, Loss: 1.5383292815909781
Epoch 99, Loss: 1.3100718428218416
Current learning rate: 0.05
Epoch 100, Loss: 1.3548771545555864
Epoch 101, Loss: 1.2212277937597573
Epoch 102, Loss: 1.1954267964083705
Epoch 103, Loss: 1.2811640322867097
Epoch 104, Loss: 1.194886249768077
Epoch 105, Loss: 1.2008436334255312
Epoch 106, Loss: 1.2160067036757014
Epoch 107, Loss: 1.200137404731897
Epoch 108, Loss: 1.2545625400236642
Epoch 109, Loss: 1.2415828091683208
Epoch 110, Loss: 1.2649873273181773
Epoch 111, Loss: 1.2532099499728924
Epoch 112, Loss: 1.2118273434504407
Epoch 113, Loss: 1.2038228024557758
Epoch 114, Loss: 1.2104680302377508
Epoch 115, Loss: 1.1761954251898752
Epoch 116, Loss: 1.1910952586252974
Epoch 117, Loss: 1.2429973854755167
Epoch 118, Loss: 1.2853663658032395
Epoch 119, Loss: 1.265137154887609
Epoch 120, Loss: 1.231297859187986
Epoch 121, Loss: 1.177979269319748
Epoch 122, Loss: 1.1716474555821326
Epoch 123, Loss: 1.2103485818664619
Epoch 124, Loss: 1.209144390825816
Epoch 125, Loss: 1.2381950604982577
Epoch 126, Loss: 1.202963473632894
Epoch 127, Loss: 1.157494676933427
Epoch 128, Loss: 1.283505208222072
Epoch 129, Loss: 1.2341189687178657
Epoch 130, Loss: 1.1648374243755681
Epoch 131, Loss: 1.2654356710578598
Epoch 132, Loss: 1.2010692116724546
Epoch 133, Loss: 1.2484193860695427
Epoch 134, Loss: 1.1462137477641292
Epoch 135, Loss: 1.2336769754200563
Epoch 136, Loss: 1.1214650675887339
Epoch 137, Loss: 1.166190494295296
Epoch 138, Loss: 1.1702736297968472
Epoch 139, Loss: 1.1678169202798176
Epoch 140, Loss: 1.1058361730915194
Epoch 141, Loss: 1.2002537901627912
Epoch 142, Loss: 1.169677761772361
Epoch 143, Loss: 1.1562167413039097
Epoch 144, Loss: 1.208864237615582
Epoch 145, Loss: 1.1435482272772395
Epoch 146, Loss: 1.1488182709606494
Epoch 147, Loss: 1.2213976733027077
Epoch 148, Loss: 1.2616925579805804
Epoch 149, Loss: 1.17687929942129
Epoch 150, Loss: 1.222670669789664
Epoch 151, Loss: 1.1812118088940768
Epoch 152, Loss: 1.2330668222435066
Epoch 153, Loss: 1.1432203128938203
Epoch 154, Loss: 1.154695254342926
Epoch 155, Loss: 1.170538492197946
Epoch 156, Loss: 1.2274459371018052
Epoch 157, Loss: 1.1578983317868512
Epoch 158, Loss: 1.1482391575162882
Epoch 159, Loss: 1.1374330370516015
Epoch 160, Loss: 1.1826234961789195
Epoch 161, Loss: 1.159861802067049
Epoch 162, Loss: 1.136234971475696
Epoch 163, Loss: 1.1356752046325438
Epoch 164, Loss: 1.2752535408210306
Epoch 165, Loss: 1.15830658578774
Epoch 166, Loss: 1.1931873495066685
Epoch 167, Loss: 1.0859543167843357
Epoch 168, Loss: 1.0843652584056334
Epoch 169, Loss: 1.116215985198752
Epoch 170, Loss: 1.2173083763515757
Epoch 171, Loss: 1.1455345115741686
Epoch 172, Loss: 1.0957792695290187
Epoch 173, Loss: 1.2346390365573408
Epoch 174, Loss: 1.157796861049561
Epoch 175, Loss: 1.1772600481862958
Epoch 176, Loss: 1.1177060314835245
Epoch 177, Loss: 1.1612261947035445
Epoch 178, Loss: 1.1555516723056698
Epoch 179, Loss: 1.1416158670361314
Epoch 180, Loss: 1.1451491521428725
Epoch 181, Loss: 1.06976770829031
Epoch 182, Loss: 1.1413885276960791
Epoch 183, Loss: 1.147158931937291
Epoch 184, Loss: 1.1328479095542827
Epoch 185, Loss: 1.1312618522540279
Epoch 186, Loss: 1.180254184225452
Epoch 187, Loss: 1.088993424037214
Epoch 188, Loss: 1.119069959664857
Epoch 189, Loss: 1.1370335216939806
Epoch 190, Loss: 1.1464322878972462
Epoch 191, Loss: 1.258609768923572
Epoch 192, Loss: 1.137135637695111
Epoch 193, Loss: 1.1661592594652075
Epoch 194, Loss: 1.168268209409492
Epoch 195, Loss: 1.1448068446602742
Epoch 196, Loss: 1.1135424457864538
Epoch 197, Loss: 1.1716235691978905
Epoch 198, Loss: 1.1440550636245825
Epoch 199, Loss: 1.1111900074998489
Current learning rate: 0.025
Epoch 200, Loss: 1.0977307568295447
Epoch 201, Loss: 1.066479959104508
Epoch 202, Loss: 1.064047067824137
Epoch 203, Loss: 1.044113783624165
Epoch 204, Loss: 1.0482357934858582
Epoch 205, Loss: 1.0395927935337468
Epoch 206, Loss: 1.1008778796064662
Epoch 207, Loss: 1.0047808078286924
Epoch 208, Loss: 1.0656936516970177
Epoch 209, Loss: 0.9991186383889953
Epoch 210, Loss: 1.0819597065679352
Epoch 211, Loss: 0.9743781071107783
Epoch 212, Loss: 0.9750920012730172
Epoch 213, Loss: 1.023295819800732
Epoch 214, Loss: 1.0027257301530386
Epoch 215, Loss: 0.9653865274813218
Epoch 216, Loss: 1.026464430668797
Epoch 217, Loss: 0.9938620247447345
Epoch 218, Loss: 0.9606355619566348
Epoch 219, Loss: 1.0413229641398574
Epoch 220, Loss: 1.019261677500072
Epoch 221, Loss: 1.0395943992687398
Epoch 222, Loss: 1.059384026010134
Epoch 223, Loss: 1.0076163335586945
Epoch 224, Loss: 1.0343929870823798
Epoch 225, Loss: 1.029991553062969
Epoch 226, Loss: 1.007030005820799
Epoch 227, Loss: 0.9955619055041073
Epoch 228, Loss: 1.0210873967663465
Epoch 229, Loss: 1.0091517019186316
Epoch 230, Loss: 1.0089594403748203
Epoch 231, Loss: 1.0297950007623762
Epoch 232, Loss: 0.9913903304442979
Epoch 233, Loss: 1.0509044406353039
Epoch 234, Loss: 0.9642352041090944
Epoch 235, Loss: 1.011480950817208
Epoch 236, Loss: 1.0091110514412356
Epoch 237, Loss: 1.0320760644972864
Epoch 238, Loss: 0.9586897316928649
Epoch 239, Loss: 1.0070088744307495
Epoch 240, Loss: 0.999097108807359
Epoch 241, Loss: 0.9612076415477749
Epoch 242, Loss: 0.9327979368886964
Epoch 243, Loss: 0.9724172676842725
Epoch 244, Loss: 0.9808856817365545
Epoch 245, Loss: 1.0344009182410232
Epoch 246, Loss: 0.926750023976547
Epoch 247, Loss: 1.0131566815255566
Epoch 248, Loss: 1.0069282504999897
Epoch 249, Loss: 0.9995472919274587
Epoch 250, Loss: 1.0314259693100352
Epoch 251, Loss: 1.0353898796809986
Epoch 252, Loss: 0.9499912016689835
Epoch 253, Loss: 0.9715300515601546
Epoch 254, Loss: 0.9725731697137048
Epoch 255, Loss: 0.9916576117742607
Epoch 256, Loss: 0.9983197042200422
Epoch 257, Loss: 0.9961522992624475
Epoch 258, Loss: 0.994946312315187
Epoch 259, Loss: 0.9928502719453988
Epoch 260, Loss: 0.9576836962284657
Epoch 261, Loss: 0.994409519235439
Epoch 262, Loss: 0.9599758627821721
Epoch 263, Loss: 0.9397132176837276
Epoch 264, Loss: 0.976122847589209
Epoch 265, Loss: 0.9705921910696201
Epoch 266, Loss: 0.9851680516708127
Epoch 267, Loss: 0.9587843591437393
Epoch 268, Loss: 0.8896618451264552
Epoch 269, Loss: 0.9566966657438293
Epoch 270, Loss: 0.9551129598476409
Epoch 271, Loss: 0.9981413723960361
Epoch 272, Loss: 0.9431746886627962
Epoch 273, Loss: 1.0206923198472388
Epoch 274, Loss: 0.9679373882436086
Epoch 275, Loss: 0.9446753844823063
Epoch 276, Loss: 0.9295993705992133
Epoch 277, Loss: 0.9377898339183584
Epoch 278, Loss: 0.97413642038448
Epoch 279, Loss: 0.997409826336777
Epoch 280, Loss: 0.9792476450612323
Epoch 281, Loss: 0.9792489341968763
Epoch 282, Loss: 0.9674874938360031
Epoch 283, Loss: 0.9842166410416141
Epoch 284, Loss: 0.9882432425188541
Epoch 285, Loss: 0.9739104088720154
Epoch 286, Loss: 0.9542285427265506
Epoch 287, Loss: 0.9805926278947388
Epoch 288, Loss: 1.009377475495394
Epoch 289, Loss: 0.9534903798189177
Epoch 290, Loss: 0.925963813225286
Epoch 291, Loss: 0.9786953746858303
Epoch 292, Loss: 1.0022922186998104
Epoch 293, Loss: 0.9672993581862621
Epoch 294, Loss: 0.9336574535481422
Epoch 295, Loss: 0.9967025037626384
Epoch 296, Loss: 0.9777109037065962
Epoch 297, Loss: 0.9556968377210835
Epoch 298, Loss: 0.9203863810898506
Epoch 299, Loss: 0.992857958632211
Current learning rate: 0.0125
Epoch 300, Loss: 0.9980829203318523
Epoch 301, Loss: 0.936624443477405
Epoch 302, Loss: 0.9321717652191089
Epoch 303, Loss: 0.9216428285354192
Epoch 304, Loss: 0.9122542338011388
Epoch 305, Loss: 0.9583648767679618
Epoch 306, Loss: 1.0036508447506123
Epoch 307, Loss: 0.9307098008307443
Epoch 308, Loss: 0.9566447162916125
Epoch 309, Loss: 0.9444316073542459
Epoch 310, Loss: 0.8979465517576496
Epoch 311, Loss: 0.9108270907086975
Epoch 312, Loss: 0.9612635880506205
Epoch 313, Loss: 0.8722006541263504
Epoch 314, Loss: 0.9749595829231776
Epoch 315, Loss: 0.9649252925624808
Epoch 316, Loss: 0.9800747921656786
Epoch 317, Loss: 0.9141128548858719
Epoch 318, Loss: 0.911521816324524
Epoch 319, Loss: 0.9310858454600002
Epoch 320, Loss: 0.969501060003319
Epoch 321, Loss: 0.9039686124116957
Epoch 322, Loss: 0.8698308872539051
Epoch 323, Loss: 0.9458432621211345
Epoch 324, Loss: 0.9484004871902229
Epoch 325, Loss: 0.9186304014644381
Epoch 326, Loss: 0.9176121123133762
Epoch 327, Loss: 0.9145515295866561
Epoch 328, Loss: 0.9487971293608733
Epoch 329, Loss: 0.8572890013044216
Epoch 330, Loss: 0.9248188684100883
Epoch 331, Loss: 0.8587554630199558
Epoch 332, Loss: 0.9213391265014607
Epoch 333, Loss: 0.8749430151071419
Epoch 334, Loss: 0.9451491408326981
Epoch 335, Loss: 0.9412014537779454
Epoch 336, Loss: 0.9182259036096156
Epoch 337, Loss: 0.8981167741419352
Epoch 338, Loss: 0.8740661330047677
Epoch 339, Loss: 0.9217630127076497
Epoch 340, Loss: 0.9117057740553206
Epoch 341, Loss: 0.9128816876140808
Epoch 342, Loss: 0.9174450526013934
Epoch 343, Loss: 0.8648276772293183
Epoch 344, Loss: 0.9157688463139445
Epoch 345, Loss: 0.8914188755871035
Epoch 346, Loss: 0.9026471802273978
Epoch 347, Loss: 0.9203600820075498
Epoch 348, Loss: 0.9233446186710219
Epoch 349, Loss: 0.9288597469541224
Epoch 350, Loss: 0.9312230724955911
Epoch 351, Loss: 0.8660800448796815
Epoch 352, Loss: 0.88042628983898
Epoch 353, Loss: 0.909124970995594
Epoch 354, Loss: 0.9351437474878369
Epoch 355, Loss: 0.9025374069595786
Epoch 356, Loss: 0.9036885985415747
Epoch 357, Loss: 0.8514825982825457
Epoch 358, Loss: 0.9055239427593624
Epoch 359, Loss: 0.9148008064379202
Epoch 360, Loss: 0.958831346722087
Epoch 361, Loss: 0.9163483097054889
Epoch 362, Loss: 0.8501351043678059
Epoch 363, Loss: 0.8943577152001646
Epoch 364, Loss: 0.95615441706316
Epoch 365, Loss: 0.8911646312701755
Epoch 366, Loss: 0.9097560164256087
Epoch 367, Loss: 0.9094917867736826
Epoch 368, Loss: 0.9350876931444436
Epoch 369, Loss: 0.8625812101080632
Epoch 370, Loss: 0.9292730084505111
Epoch 371, Loss: 0.9016899491582661
Epoch 372, Loss: 0.914399678681008
Epoch 373, Loss: 0.9626504813259346
Epoch 374, Loss: 0.8999643374877774
Epoch 375, Loss: 0.8569108859831094
Epoch 376, Loss: 0.9090458727088856
Epoch 377, Loss: 0.8803799513841644
Epoch 378, Loss: 0.9437082833197913
Epoch 379, Loss: 0.913573188941203
Epoch 380, Loss: 0.8760249120550645
Epoch 381, Loss: 0.9182089763747728
Epoch 382, Loss: 0.9023115343174208
Epoch 383, Loss: 0.9424999048008036
Epoch 384, Loss: 0.9012791725216264
Epoch 385, Loss: 0.8741815023457143
Epoch 386, Loss: 0.8630770787080385
Epoch 387, Loss: 0.8600767786966671
Epoch 388, Loss: 0.9076745412376321
Epoch 389, Loss: 0.8647798007039303
Epoch 390, Loss: 0.8565505108040785
Epoch 391, Loss: 0.8842201298675355
Epoch 392, Loss: 0.9041450011949257
Epoch 393, Loss: 0.9025835203458679
Epoch 394, Loss: 0.8363399316136647
Epoch 395, Loss: 0.854623936412702
Epoch 396, Loss: 0.9027435594615826
Epoch 397, Loss: 0.848571385593616
Epoch 398, Loss: 0.8507947092508604
Epoch 399, Loss: 0.8905737108220715
Current learning rate: 0.00625
Epoch 400, Loss: 0.8701084370938588
Epoch 401, Loss: 0.8884527002802072
Epoch 402, Loss: 0.8510083574943733
Epoch 403, Loss: 0.8559083706550198
Epoch 404, Loss: 0.8693601486303055
Epoch 405, Loss: 0.836764784202269
Epoch 406, Loss: 0.8884501876442035
Epoch 407, Loss: 0.8439849792521739
Epoch 408, Loss: 0.8965187794495846
Epoch 409, Loss: 0.8735398793473033
Epoch 410, Loss: 0.8664840020116406
Epoch 411, Loss: 0.8274204683293163
Epoch 412, Loss: 0.869137309576349
Epoch 413, Loss: 0.9013611435898543
Epoch 414, Loss: 0.8439821843083889
Epoch 415, Loss: 0.8298054370194969
Epoch 416, Loss: 0.9134002582696993
Epoch 417, Loss: 0.8785572217282464
Epoch 418, Loss: 0.8554925395398574
Epoch 419, Loss: 0.9359551211952826
Epoch 420, Loss: 0.8166879559660725
Epoch 421, Loss: 0.8785048707328679
Epoch 422, Loss: 0.8678883550992222
Epoch 423, Loss: 0.8493209957191169
Epoch 424, Loss: 0.8286969302911702
Epoch 425, Loss: 0.8203073143537843
Epoch 426, Loss: 0.8498691529406263
Epoch 427, Loss: 0.910302206918798
Epoch 428, Loss: 0.8399676254267697
Epoch 429, Loss: 0.8163472313946015
Epoch 430, Loss: 0.795191227989163
Epoch 431, Loss: 0.8890750033936311
Epoch 432, Loss: 0.8728484721017203
Epoch 433, Loss: 0.857831875954389
Epoch 434, Loss: 0.8515885621618426
Epoch 435, Loss: 0.9012135591043798
Epoch 436, Loss: 0.8664263562200276
Epoch 437, Loss: 0.8751634601278098
Epoch 438, Loss: 0.8688243168589757
Epoch 439, Loss: 0.8577713622443955
Epoch 440, Loss: 0.9251348604998698
Epoch 441, Loss: 0.8241667689652389
Epoch 442, Loss: 0.8364007117149713
Epoch 443, Loss: 0.9277644530176846
Epoch 444, Loss: 0.8532486341599135
Epoch 445, Loss: 0.7984862942867227
Epoch 446, Loss: 0.8546088382046159
Epoch 447, Loss: 0.8458067066081439
Epoch 448, Loss: 0.9140204923374285
Epoch 449, Loss: 0.9120405304442065
Epoch 450, Loss: 0.8882083811934586
Epoch 451, Loss: 0.8466477779895031
Epoch 452, Loss: 0.8663414238644545
Epoch 453, Loss: 0.8930473255109874
Epoch 454, Loss: 0.8849088280127414
Epoch 455, Loss: 0.8563033736010479
Epoch 456, Loss: 0.8733278329076761
Epoch 457, Loss: 0.8999615964333119
Epoch 458, Loss: 0.8268152172381231
Epoch 459, Loss: 0.8692960112361247
Epoch 460, Loss: 0.8637091302796742
Epoch 461, Loss: 0.8551175654675758
Epoch 462, Loss: 0.8920562299980848
Epoch 463, Loss: 0.8209263082429392
Epoch 464, Loss: 0.9110141879560161
Epoch 465, Loss: 0.85829103528064
Epoch 466, Loss: 0.8468741976715062
Epoch 467, Loss: 0.8114708427369364
Epoch 468, Loss: 0.8536686033296919
Epoch 469, Loss: 0.8214151825444989
Epoch 470, Loss: 0.8768687277890292
Epoch 471, Loss: 0.8703294938179416
Epoch 472, Loss: 0.8605903224776946
Epoch 473, Loss: 0.8354176788352108
Epoch 474, Loss: 0.8699369210114306
Epoch 475, Loss: 0.8704858018552631
Epoch 476, Loss: 0.870990669560411
Epoch 477, Loss: 0.8970497670815255
Epoch 478, Loss: 0.8309216684889503
Epoch 479, Loss: 0.885563728132189
Epoch 480, Loss: 0.8684309355207938
Epoch 481, Loss: 0.80107602755888
Epoch 482, Loss: 0.8372480103321795
Epoch 483, Loss: 0.8510346346804969
Epoch 484, Loss: 0.8958314253923193
Epoch 485, Loss: 0.8531767264080103
Epoch 486, Loss: 0.8907741665437373
Epoch 487, Loss: 0.8525280074494344
Epoch 488, Loss: 0.8621789347336615
Epoch 489, Loss: 0.8467641887333502
Epoch 490, Loss: 0.8876465473795122
Epoch 491, Loss: 0.8541066420872292
Epoch 492, Loss: 0.861849488605367
Epoch 493, Loss: 0.8605899194250187
Epoch 494, Loss: 0.832671678278123
Epoch 495, Loss: 0.820334944066262
Epoch 496, Loss: 0.8455726466958581
Epoch 497, Loss: 0.7456529705314096
Epoch 498, Loss: 0.87416559120867
Epoch 499, Loss: 0.8656989217980275
Current learning rate: 0.003125
Epoch 500, Loss: 0.8368914585876528
```

[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

```
[ ]:
```