limhpone / computervision-final-prep

Code Issues Pull requests Actions Projects Wiki Security In

computervision-final-prep / lab / Lab 07 (CNN)-20251128 / lab7_classifiers_step_by_step.ipynb

limhpone Lab 7 40d7fbe · 2 hours ago

1254 lines (1254 loc) · 333 KB

Preview Code Blame Raw

Linear Classifier

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
- Activation (Sigmoid, Softmax)
- Loss (cost) function calculation
- Gradient Descent (both batch and stochastic)
- Weight update
- Training

---

0. Import necessary libraries

In [32]:
```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
import torch.nn as nn
import torch.nn.functional as F

from torchvision import models
from torchvision import datasets, transforms
```

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.

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.

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 $m \times (n+1)$:

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

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):

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

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.

In [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]

# X.shape[0] gives the number of rows in X.
# np.ones((X.shape[0], 1)) creates a column vector of ones with the same num
# np.c_ concatenates the column of ones to the original feature matrix X col
```

In [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
    if multiclass:
        n_classes = y.shape[1]
        theta = np.random.randn(n_features, n_classes) * 0.01  # Small rando
    else:
        theta = np.random.randn(n_features, 1) * 0.01  # Small random weight
    return theta
```

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

3. 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.

Multinomial Classification (Softmax)

In multinomial classification, the softmax function is used to compute probabilities across multiple classes. The softmax function $\text{softmax}(z_i)$ 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.

In [5]:
```python
# 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 numeric
    return exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
```

4. Loss Functions for Classification

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

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.

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.

In [6]:

```
# Binary Cross Entropy Loss
def binary_cross_entropy_loss(y_true, y_pred):
    m = y_true.shape[0]                               # Number of samples
    epsilon = 1e-15                                   # To avoid log(0) this is a
    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_pre

# Categorical Cross Entropy Loss
def categorical_cross_entropy_loss(y_true, y_pred):
    m = y_true.shape[0]                               # Number of samples
    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
```

5. 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.

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.

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).

6. 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.

In [7]:

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

In [8]:

```
# 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]
        prediction = linear_prediction(xi, theta)
        if multiclass:
            prediction = softmax(prediction)
        else:
            prediction = sigmoid(prediction)
        gradients = np.dot(xi.T, (prediction - yi))
        theta = theta - learning_rate * gradients
    return theta
```

7. Training

In [9]:

```
# Training function
def train_model(X, y, learning_rate=0.01, iterations=1000, batch=True, multi
    # 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
        else:
            # Use stochastic gradient descent
            theta = stochastic_gradient_descent_step(X, y, theta, learning_r

        # Print loss every 100 iterations
        if i % 200 == 0:
            print(f"Iteration {i}/{iterations}, Loss: {loss:.4f}")

    return theta, losses
```

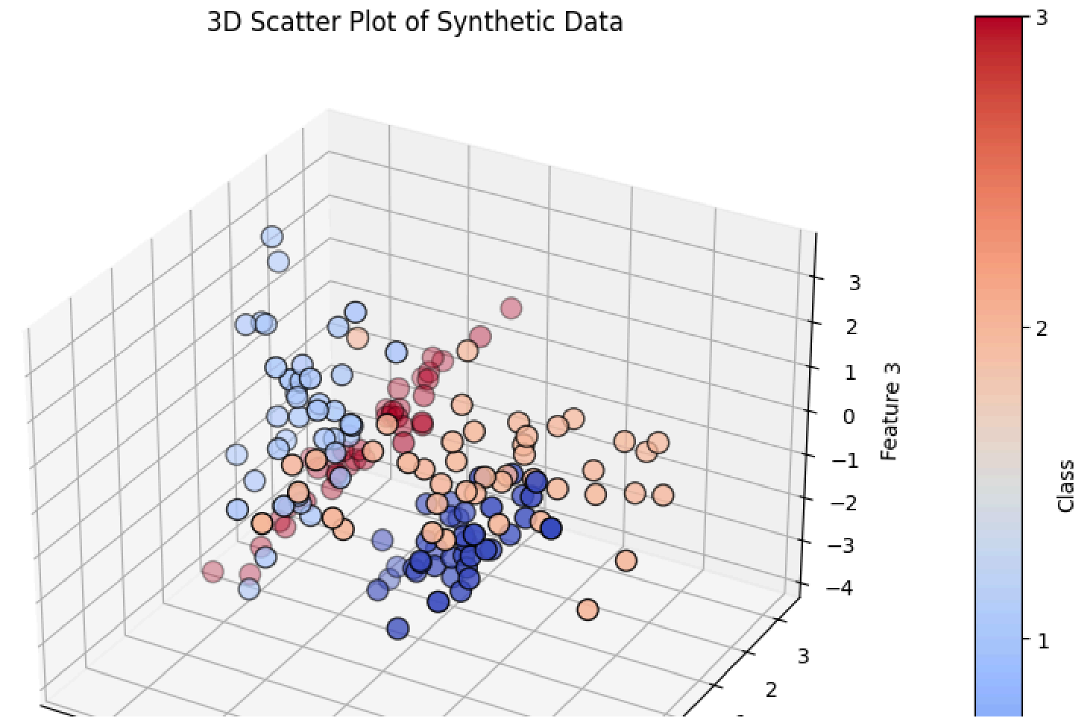
Generate Data and Prepare Training and Test Set

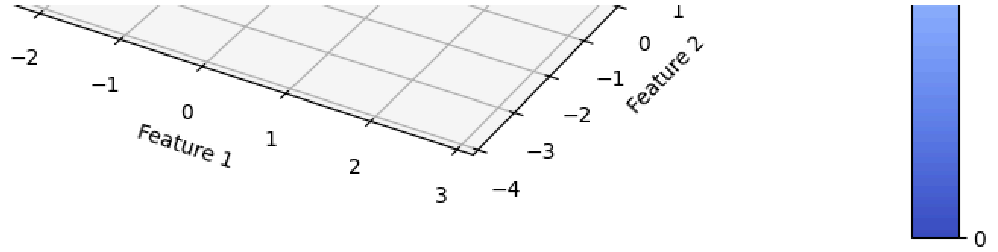
In [10]:

```
# 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_inform
                                       n_redundant=0, n_clusters_per_class=1, n_
    # 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_classe
    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
```

Visualize Training Data

In [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[:
                         c=y_train_labels, cmap='coolwarm', edgecolor='k', s

    # 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)

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

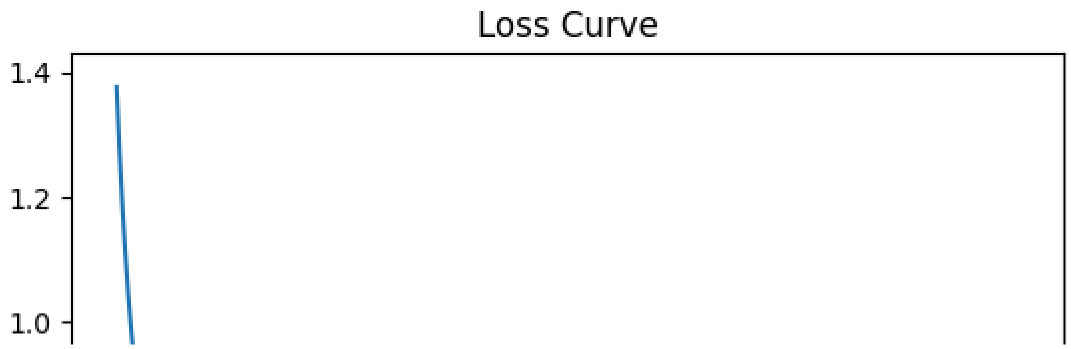


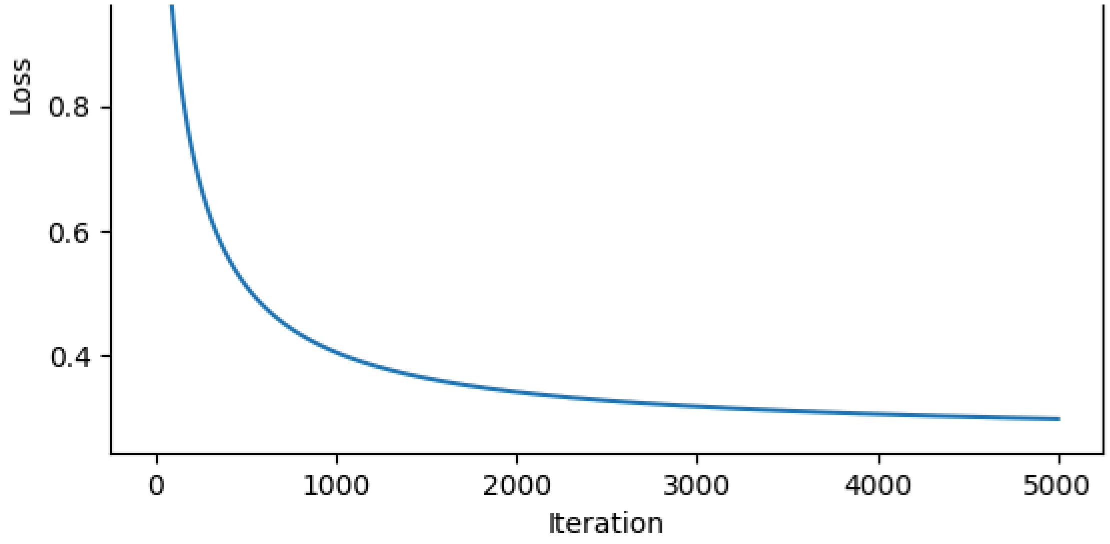


In [12]:
```
start_time = time.time()
theta, losses = train_model(X_train_syn, y_train_syn, learning_rate=0.01, it
print(f"Time Taken Using Batch gradient descent :{time.time() - start_time}"
```

```
Iteration 0/5000, Loss: 1.3763
Iteration 200/5000, Loss: 0.7321
Iteration 400/5000, Loss: 0.5583
Iteration 600/5000, Loss: 0.4796
Iteration 800/5000, Loss: 0.4347
Iteration 1000/5000, Loss: 0.4058
Iteration 1200/5000, Loss: 0.3855
Iteration 1400/5000, Loss: 0.3705
Iteration 1600/5000, Loss: 0.3589
Iteration 1800/5000, Loss: 0.3497
Iteration 2000/5000, Loss: 0.3423
Iteration 2200/5000, Loss: 0.3361
Iteration 2400/5000, Loss: 0.3308
Iteration 2600/5000, Loss: 0.3263
Iteration 2800/5000, Loss: 0.3224
Iteration 3000/5000, Loss: 0.3190
Iteration 3200/5000, Loss: 0.3160
Iteration 3400/5000, Loss: 0.3133
Iteration 3600/5000, Loss: 0.3109
Iteration 3800/5000, Loss: 0.3088
Iteration 4000/5000, Loss: 0.3068
Iteration 4200/5000, Loss: 0.3050
Iteration 4400/5000, Loss: 0.3033
Iteration 4600/5000, Loss: 0.3018
Iteration 4800/5000, Loss: 0.3004
Time Taken Using Batch gradient descent :0.4219813346862793
```

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





In [14]:

```
# 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%
```

For Image what could be possible changes??

In [15]:

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

```
Using device: cuda
```

In [16]:

```
cifar_train = datasets.CIFAR10('./data', train=True, download=True ,transfor
cifar_test = datasets.CIFAR10('./data', train=False, download=True ,transfor
```

In [27]:

```
# 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.

# 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 * 0.2), shu

# 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 tensors and move to device
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([200, 3, 32, 32])
After Flattening
Training data shape: torch.Size([1000, 3072])
Test data shape: torch.Size([200, 3072])
```

In [28]:

```
class ImageLinearClassifier:
    def __init__(self, input_size, n_classes, device=None):
        super().__init__()
        self.device = device or ("cuda" if torch.cuda.is_available() else "c
        print("Using device:", self.device)
        self.W = torch.nn.Parameter(torch.randn(n_classes, input_size, devic
        self.b = torch.nn.Parameter(torch.zeros(n_classes, device=self.devic

    def predict(self, X):
        # X: (batch_size, input_size)
        X = X.to(self.device)
        return X @ self.W.T + self.b  # (batch_size, n_classes)

    def compute_loss(self, X, y):
        X = X.to(self.device)
        y = y.to(self.device).long()
        logits = self.predict(X)
        return F.cross_entropy(logits, y)

    def gradient_descent(self, X, y, learning_rate=0.001):
        loss = self.compute_loss(X, y)
        loss.backward()
        with torch.no_grad():
            self.W -= learning_rate * self.W.grad
            self.b -= learning_rate * self.b.grad
        self.W.grad.zero_()
        self.b.grad.zero_()
        return loss.item()
```

In [29]:

```
def train(classifier, X_train, y_train, epochs, learning_rate):
    losses = []
    for i in range(epochs):
        loss = classifier.gradient_descent(X_train, y_train, learning_rate)
        losses.append(loss)
        print(f"Epoch {i+1}, Loss: {loss:.4f}")
    return losses
```

In [30]:

```
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}")
```

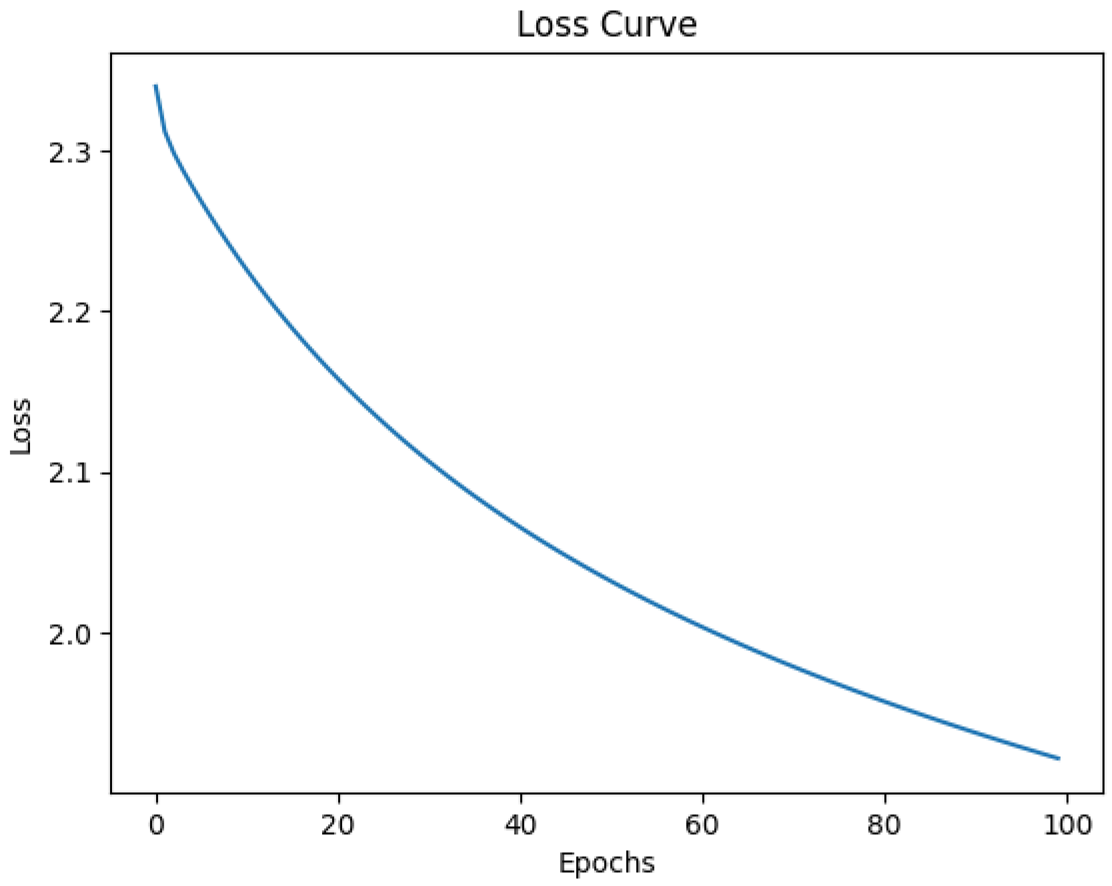
```
Training data: 50000
Test data: 10000
Image shape: torch.Size([3, 32, 32])
```

In [33]:

```
# Example usage
n_classes = 10
image_size = 32 * 32 * 3
classifier = ImageLinearClassifier(input_size=image_size, n_classes=n_classe

# 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)
```

```
Using device: cuda
Epoch 1, Loss: 2.3399
Epoch 2, Loss: 2.3118
Epoch 3, Loss: 2.2982
Epoch 4, Loss: 2.2875
Epoch 5, Loss: 2.2777
Epoch 6, Loss: 2.2682
Epoch 7, Loss: 2.2591
Epoch 8, Loss: 2.2503
Epoch 9, Loss: 2.2418
Epoch 10, Loss: 2.2335
Epoch 11, Loss: 2.2255
Epoch 12, Loss: 2.2178
Epoch 13, Loss: 2.2103
Epoch 14, Loss: 2.2030
Epoch 15, Loss: 2.1960
Epoch 16, Loss: 2.1891
Epoch 17, Loss: 2.1825
Epoch 18, Loss: 2.1761
Epoch 19, Loss: 2.1698
Epoch 20, Loss: 2.1637
Epoch 21, Loss: 2.1578
Epoch 22, Loss: 2.1520
Epoch 23, Loss: 2.1464
Epoch 24, Loss: 2.1409
Epoch 25, Loss: 2.1356
Epoch 26, Loss: 2.1304
Epoch 27, Loss: 2.1253
Epoch 28, Loss: 2.1204
Epoch 29, Loss: 2.1155
Epoch 30, Loss: 2.1108
Epoch 31, Loss: 2.1062
Epoch 32, Loss: 2.1017
Epoch 33, Loss: 2.0973
Epoch 34, Loss: 2.0930
Epoch 35, Loss: 2.0888
Epoch 36, Loss: 2.0847
Epoch 37, Loss: 2.0806
Epoch 38, Loss: 2.0767
Epoch 39, Loss: 2.0728
Epoch 40, Loss: 2.0690
Epoch 41, Loss: 2.0653
Epoch 42, Loss: 2.0616
Epoch 43, Loss: 2.0580
Epoch 44, Loss: 2.0545
Epoch 45, Loss: 2.0511
Epoch 46, Loss: 2.0477
Epoch 47, Loss: 2.0443
Epoch 48, Loss: 2.0411
Epoch 49, Loss: 2.0379
Epoch 50, Loss: 2.0347
Epoch 51, Loss: 2.0316
Epoch 52, Loss: 2.0285
Epoch 53, Loss: 2.0255
Epoch 54, Loss: 2.0226
Epoch 55, Loss: 2.0196
Epoch 56, Loss: 2.0168
Epoch 57, Loss: 2.0140
Epoch 58, Loss: 2.0112
Epoch 59, Loss: 2.0084
Epoch 60, Loss: 2.0057
Epoch 61, Loss: 2.0031
Epoch 62, Loss: 2.0005
Epoch 63, Loss: 1.9979
Epoch 64, Loss: 1.9953
Epoch 65, Loss: 1.9928
Epoch 66, Loss: 1.9903
Epoch 67, Loss: 1.9879
Epoch 68, Loss: 1.9855
Epoch 69, Loss: 1.9831
Epoch 70, Loss: 1.9807
Epoch 71, Loss: 1.9784
Epoch 72, Loss: 1.9761
Epoch 73, Loss: 1.9739
Epoch 74, Loss: 1.9716
Epoch 75, Loss: 1.9694
Epoch 76, Loss: 1.9672
Epoch 77, Loss: 1.9651
Epoch 78, Loss: 1.9629
Epoch 79, Loss: 1.9608
Epoch 80, Loss: 1.9587
Epoch 81, Loss: 1.9567
Epoch 82, Loss: 1.9546
Epoch 83, Loss: 1.9526
Epoch 84, Loss: 1.9506
Epoch 85, Loss: 1.9487
Epoch 86, Loss: 1.9467
Epoch 87, Loss: 1.9448
Epoch 88, Loss: 1.9429
Epoch 89, Loss: 1.9410
Epoch 90, Loss: 1.9391
Epoch 91, Loss: 1.9373
Epoch 92, Loss: 1.9354
Epoch 93, Loss: 1.9336
Epoch 94, Loss: 1.9318
Epoch 95, Loss: 1.9300
Epoch 96, Loss: 1.9283
Epoch 97, Loss: 1.9265
Epoch 98, Loss: 1.9248
Epoch 99, Loss: 1.9231
Epoch 100, Loss: 1.9214
```

In [34]:
```
plt.plot(losses)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Loss Curve')
plt.show()
```



MLP - Multi Layer Perceptron - DeepLearning

MLP structure

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. Output Layer: A fully connected layer with 10 neurons (for 10 classes) and softmax activation.

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 sigmoid for binomial and softmax activation for multinomial classification.
4. Loss Function: Cross-entropy loss for classification.
5. Backpropagation: To update weights using gradients from the loss.

In [35]:

```
class MLPClassifier:
    def __init__(self, input_size, hidden_size, output_size):
        # Weight initialization
        self.W1 = np.random.randn(hidden_size, input_size) * 0.01   # (hidde
        self.b1 = np.zeros((hidden_size, 1))                        # (hidde
        self.W2 = np.random.randn(output_size, hidden_size) * 0.01  # (outpu
        self.b2 = np.zeros((output_size, 1))                        # (outpu

    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))        # Numeri
        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                      # (hidde
        self.A1 = self.relu(self.Z1)                                # Apply

        # Layer 2 (output layer)
        self.Z2 = np.dot(self.W2, self.A1) + self.b2                # (outpu
        self.A2 = self.softmax(self.Z2)                             # Apply
        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
        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

        # Gradient of the loss w.r.t. Z2
        dZ2 = self.A2                                              # Softma
        dZ2[y, range(m)] -= 1                                      # Subtra
        dZ2 /= m

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

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

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

        # 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
```

In [39]:

```
input_size = 3072              # CIFAR-10 images are 32x32x3
```

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

# Move data to CPU and convert to numpy
X_train_np = X_train.detach().cpu().numpy()
y_train_np = y_train.detach().cpu().numpy()

# Use transposed input since your model expects (input_size, batch_size)
X_train_np = X_train_np.T  # shape (3072, 1000)

# Train
mlp = MLPClassifier(input_size, hidden_size, output_size)
losses = mlp.train(X_train_np, y_train_np, epochs=100, learning_rate=0.01)
```

```
Epoch 1, Loss: 2.3033978156876955
Epoch 2, Loss: 2.3032235250783715
Epoch 3, Loss: 2.3030507460766128
Epoch 4, Loss: 2.302879017710775
Epoch 5, Loss: 2.3027090088614033
Epoch 6, Loss: 2.3025413079334784
Epoch 7, Loss: 2.302376501944039
Epoch 8, Loss: 2.302212927074942
Epoch 9, Loss: 2.3020506320777785
Epoch 10, Loss: 2.3018879372810983
Epoch 11, Loss: 2.301726018588772
Epoch 12, Loss: 2.301564500283167
Epoch 13, Loss: 2.301404984081487
Epoch 14, Loss: 2.301246674960572
Epoch 15, Loss: 2.3010896689233014
Epoch 16, Loss: 2.3009327134441717
Epoch 17, Loss: 2.3007754351755954
Epoch 18, Loss: 2.300618272018496
Epoch 19, Loss: 2.300462892375539
Epoch 20, Loss: 2.3003086321664963
Epoch 21, Loss: 2.300154361057534
Epoch 22, Loss: 2.300000762127006
Epoch 23, Loss: 2.2998472058908126
Epoch 24, Loss: 2.299692378407266
Epoch 25, Loss: 2.299536974248486
Epoch 26, Loss: 2.299381489311103
Epoch 27, Loss: 2.299227758947444
Epoch 28, Loss: 2.2990748628325837
Epoch 29, Loss: 2.2989235558347105
Epoch 30, Loss: 2.2987723755927423
Epoch 31, Loss: 2.2986208986150176
Epoch 32, Loss: 2.298469430219463
Epoch 33, Loss: 2.2983178331711227
Epoch 34, Loss: 2.2981654934775406
Epoch 35, Loss: 2.298013272917081
Epoch 36, Loss: 2.297861889709511
Epoch 37, Loss: 2.2977103982478986
Epoch 38, Loss: 2.297558812227994
Epoch 39, Loss: 2.297407861520469
Epoch 40, Loss: 2.29725574622175
Epoch 41, Loss: 2.2971041510535617
Epoch 42, Loss: 2.296952767961231
Epoch 43, Loss: 2.296800213978342
Epoch 44, Loss: 2.2966468538942078
Epoch 45, Loss: 2.2964927519750304
Epoch 46, Loss: 2.296338377882077
Epoch 47, Loss: 2.296183490879301
Epoch 48, Loss: 2.2960278439786053
Epoch 49, Loss: 2.29587297816373
Epoch 50, Loss: 2.295718329391879
Epoch 51, Loss: 2.2955630742340483
Epoch 52, Loss: 2.2954073973236317
Epoch 53, Loss: 2.2952509740571876
Epoch 54, Loss: 2.295093891247854
Epoch 55, Loss: 2.2949359985630853
Epoch 56, Loss: 2.2947770706458037
Epoch 57, Loss: 2.2946177998063426
Epoch 58, Loss: 2.2944578564879783
Epoch 59, Loss: 2.2942969754338627
Epoch 60, Loss: 2.2941354646809877
Epoch 61, Loss: 2.2939734934188167
Epoch 62, Loss: 2.2938106743201643
Epoch 63, Loss: 2.2936472184956314
Epoch 64, Loss: 2.2934834175207435
Epoch 65, Loss: 2.2933193808986267
Epoch 66, Loss: 2.2931539604206486
Epoch 67, Loss: 2.2929867915689996
Epoch 68, Loss: 2.292818156217661
Epoch 69, Loss: 2.2926477431562216
Epoch 70, Loss: 2.292476428645309
Epoch 71, Loss: 2.2923042167557033
Epoch 72, Loss: 2.2921316822311146
Epoch 73, Loss: 2.2919577435332186
Epoch 74, Loss: 2.2917817942470604
Epoch 75, Loss: 2.291604025504943
Epoch 76, Loss: 2.2914246955130273
Epoch 77, Loss: 2.2912435986709867
Epoch 78, Loss: 2.2910618163137477
Epoch 79, Loss: 2.2908793817025805
Epoch 80, Loss: 2.290696178366223
Epoch 81, Loss: 2.29051203556387
Epoch 82, Loss: 2.2903249059076036
Epoch 83, Loss: 2.2901355830457693
Epoch 84, Loss: 2.2899439627542324
Epoch 85, Loss: 2.289750416071454
Epoch 86, Loss: 2.289555797986927
Epoch 87, Loss: 2.2893595414123076
```