Final_Project_DS-5220

Ruchik Jani (NUID - 002825482)

2024-04-26

Programming Language Used in this project: Python

Method-1: Deep Neural Network with Convolution Layer

Introduction

In this method, I have split the execution into two different codes. These codes collectively illustrate a full pipeline for training, evaluating, and utilizing a CNN model for image classification tasks using PyTorch and torchvision. I have trained and evaluated the CNN model on all 100 classes.

CODE-1: In this program, the process begins with the definition of a custom CNN architecture consisting of multiple convolution layers, batch normalization, and residual blocks. The training process iterates over epochs and updates the model parameters using Adam optimizer with OneCycleLR scheduler. After training, the model's weights and optimizer states are saved, followed by visualization of training and test loss and accuracy curves. I have selected 50 epochs as the appropriate value after running various experiments. This value should not cause overfitting.

CODE-2: This program demonstrates how to evaluate the trained model on the CIFAR-100 test dataset. It loads the model, processes the test dataset with appropriate transformations, and displays predictions alongside the original images. The CIFAR-100 class names are retrieved for interpretation of model predictions. The trained model is loaded, evaluated, and results are displayed interactively through image visualization and corresponding predicted class labels.

Code-1: Model Training, saving the weights to a file

and visualize training performance

Import Statements

```
## Import Statements
import torch
import sys
import torchvision
import torchmetrics
from torchvision.datasets import CIFAR100
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import matplotlib.pyplot as plt
import torchvision.models
import time
import seaborn


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

```
## Using device: cuda
```

Define the Neural Network Architecture and create a model

```
# Create a sequence of convolution layers with batch normalization and ReLU activation.
def convolution_function(input_ch, output_ch, pool=False):
    layers = [nn.Conv2d(input_ch, output_ch, kernel_size=3, padding=1),
              nn.BatchNorm2d(output_ch),
              nn.ReLU(inplace=True)]
    if pool: layers.append(nn.MaxPool2d(2))
    return nn.Sequential(*layers)

# Custom neural network architecture composed of convolution layers and residual connect
ions,
# followed by a classifier for image classification.
class MyNetwork(nn.Module):
    #Initialize the network architecture.
    def __init__(self, input_ch, num_classes=100):
        super().__init__()
         # Define convolution layers
        self.convolution1 = convolution_function(input_ch, 64)
        self.convolution2 = convolution_function(64, 128, pool=True)
        self.reslayer1 = nn.Sequential(convolution_function(128, 128), convolution_funct
ion(128, 128))

        self.convolution_3 = convolution_function(128, 256, pool=True)
        self.convolution_4 = convolution_function(256, 512, pool=True)
        self.reslayer2 = nn.Sequential(convolution_function(512, 512), convolution_funct
ion(512, 512))

        self.convolution_5 = convolution_function(512, 1028, pool=True)
        self.reslayer3 = nn.Sequential(convolution_function(1028, 1028), convolution_fun
ction(1028, 1028))

        # Define classifier
        self.classifier = nn.Sequential(nn.MaxPool2d(2), # Global average pooling to red
uce spatial dimensions to 1x1
                                        nn.Flatten(), # Flatten the feature map
                                        nn.Linear(1028, num_classes)) # Fully connected
layer for classification

    # Forward pass through the network
    def forward(self, x):
        output = self.convolution1(x)
        output = self.convolution2(output)
        output = self.reslayer1(output) + output
        output = self.convolution_3(output)
        output = self.convolution_4(output)
        output = self.reslayer2(output) + output
        output = self.convolution_5(output)
        output = self.reslayer3(output) + output
        output = self.classifier(output)
        return output
```

```
# Create an instance of the MyNetwork class and move it to the appropriate device (GPU i
f available).
def create_model(num_classes):
    # Create the final model
    model = MyNetwork(3, num_classes)
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(device)
    model = model.to(device)

    return model, device
```

Model Training

```
# Train the model using the given hyper-parameters
def train_model(train_loader, val_loader, device, model, criterion, optimizer, schedule
r, epochs):

    training_losses = []  # List to store training losses for each epoch
    validation_losses = []  # List to store validation losses for each epoch
    training_accuracies = []  # List to store training accuracies for each epoch
    validation_accuracies = []  # List to store validation accuracies for each epoch
    training_times = []  # List to store training times for each epoch

    # Metric for computing accuracy
    train_acc_metric = torchmetrics.Accuracy(num_classes=100, average='macro', task='mul
ticlass')
    val_acc_metric = torchmetrics.Accuracy(num_classes=100, average='macro', task='multi
class')

    for epoch in range(epochs):
        start_time = time.time()
        print(f'Training epoch {epoch + 1}')

        # Training
        model.train()
        train_batch_losses = [] # List to store training losses for each batch
        for inputs, targets in train_loader:
            inputs, targets = inputs.to(device), targets.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, targets)
            loss.backward()
            optimizer.step()
            train_batch_losses.append(loss.item())
            train_acc_metric(outputs.cpu(), targets.cpu())

        # Testing
        model.eval()
        val_batch_losses = [] # List to store validation losses for each batch
        for inputs, targets in val_loader:
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, targets)
            val_batch_losses.append(loss.item())
            val_acc_metric(outputs.cpu(), targets.cpu())

        # Update loss and accuracy lists
        training_losses.append(sum(train_batch_losses) / len(train_batch_losses))
        validation_losses.append(sum(val_batch_losses) / len(val_batch_losses))
        training_accuracies.append(train_acc_metric.compute())
        validation_accuracies.append(val_acc_metric.compute())
        train_acc_metric.reset()
        val_acc_metric.reset()
```

```
        scheduler.step()

        end_time = time.time()
        epoch_time = end_time - start_time
        training_times.append(epoch_time)

        print(f'Training Loss: {training_losses[-1]:.3f}, Training Accuracy: {training_a
ccuracies[-1]:.3f}, '
              f'Test Loss: {validation_losses[-1]:.3f}, Test Accuracy: {validation_accur
acies[-1]:.3f}, '
              f'Time: {epoch_time:.2f}s')

    print('Training is complete.')

    # Calculate averages
    avg_training_loss = sum(training_losses) / len(training_losses)
    avg_validation_loss = sum(validation_losses) / len(validation_losses)
    avg_training_accuracy = sum(training_accuracies) / len(training_accuracies)
    avg_validation_accuracy = sum(validation_accuracies) / len(validation_accuracies)
    avg_training_time = sum(training_times) / len(training_times)

    print(f'Average Training Loss: {avg_training_loss:.3f}')
    print(f'Average Test Loss: {avg_validation_loss:.3f}')
    print(f'Average Training Accuracy: {avg_training_accuracy:.3f}')
    print(f'Average Test Accuracy: {avg_validation_accuracy:.3f}')
    print(f'Average Training Time per Epoch: {avg_training_time:.2f}s')

    return training_losses, training_accuracies, validation_losses, validation_accuracie
s
```

Plot the loss and accuracy curves

```
# Function to plot the training and validation loss and accuracy curves.
def visualize_training(loss_train, loss_val, loss_acc, acc_val, epoch):
    seaborn.set_style("darkgrid") # Set plot style to seaborn.

    # Plot training and validation loss
    plt.figure(figsize=(10, 5))
    plt.plot(loss_train, label='Training Loss')
    plt.plot(loss_val, label='Test Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Training and Test Loss')
    plt.legend()
    plt.xticks(range(0, epoch + 1))  # Set x-axis ticks for all epochs
    plt.show()

    # Plot training and validation accuracy
    plt.figure(figsize=(10, 5))
    plt.plot(loss_acc, label='Training Accuracy')
    plt.plot(acc_val, label='Test Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.title('Training and Test Accuracy')
    plt.legend()
    plt.xticks(range(0, epoch + 1))  # Set x-axis ticks for all epochs
    plt.show()
    return
```

Main Function

Here I have commented the code and displayed the output of the code by storing it in a text file and loading it in the r-chunk. This is because it will take too long to knit the document if the CNN model training code is executed. The accuracy and loss curves have been stored as images and I will display them in the Analysis section.

```
# Main function to train a neural network model on the CIFAR-100 dataset
# def main(argv):
#
#     random_seed = 1
#     torch.manual_seed(random_seed)
#
#     # Path to the folder containing CIFAR-100 dataset
#     ROOT_PATH = '/media/rj/New Volume/Northeastern University/Semester-2/DS 5220 - SML
T/Projects/Final Project/Code/'
#
#     # Define batch size
#     BATCH_SIZE = 128
#
#     # Define data transformations
#     train_transform = transforms.Compose([
#     transforms.RandomCrop(32, padding=4, padding_mode='reflect'),
#     transforms.RandomHorizontalFlip(),
#     transforms.RandomRotation(10),
#     transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),  #
Apply color jittering
#     transforms.ToTensor(),
#     transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
#     ])
#
#     test_transform = transforms.Compose([
#         transforms.ToTensor(),
#         transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
#     ])
#
#     # Load CIFAR-100 training and evaluation datasets
#     train_dataset = CIFAR100(root=ROOT_PATH, download=False, train=True, transform=tra
in_transform)
#     eval_dataset = CIFAR100(root=ROOT_PATH, download=False, train=False, transform=tes
t_transform)
#
#     # Create data loaders for training and evaluation
#     train_data_loader = DataLoader(dataset=train_dataset, num_workers=2, batch_size=BA
TCH_SIZE, shuffle=True, pin_memory=True)
#     eval_data_loader = DataLoader(dataset=eval_dataset, num_workers=2, batch_size=BATC
H_SIZE*2, shuffle=False, pin_memory=True)
#
#     # Instantiate the model and move it to the appropriate device
#     model, device = create_model(num_classes=100)
#
#     # Define loss function and optimizer
#     criterion = nn.CrossEntropyLoss()
#
#     # Define L2 regularization strength
#     w_decay = 0.001
#     optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=w_decay)
#
#     modelsavepath = "./models/model.pth"
```

```
#     optimizersavepath = "./models/optimizer.pth"
#     epochs = 50
#
#     # Train the model
#     #train_model(model, train_data_loader, eval_data_loader, optimizer, criterion, epo
chs)
#
#     #total_steps = len(train_data_loader)*epochs
#     steps_per_epoch = len(train_data_loader)
#     # Define learning rate scheduler
#     scheduler = optim.lr_scheduler.OneCycleLR(optimizer, max_lr=0.001, epochs=epochs,
steps_per_epoch=steps_per_epoch)
#
#
#     # Model Training Phase
#     loss_train, loss_acc, loss_val, acc_val = train_model(
#         train_data_loader, eval_data_loader, device, model,
#         criterion, optimizer, scheduler, epochs
#     )
#
#     # Save model checkpoint and optimizer state
#     torch.save(model.state_dict(), modelsavepath)
#     torch.save(optimizer.state_dict(), optimizersavepath)
#
#     # Plot the accuracy and loss curves
#     visualize_training(loss_train, loss_val, loss_acc, acc_val, epochs)
#
#     return
#
# if __name__ == "__main__":
#     main(sys.argv)

# Define the path to your text file
file_path = "/media/rj/New Volume/Northeastern University/Semester-2/DS 5220 - SMLT/Proj
ects/Final Project/Code/CNNResults.txt"

# Open the text file in read mode
with open(file_path, 'r') as file:
    # Read all lines from the file
    file_contents = file.readlines()

# Print the contents of the file
for line in file_contents:
    print(line.strip())  # Use strip() to remove any leading/trailing whitespace
```

```
## cuda
## Training epoch 1
## Training Loss: 3.681, Training Accuracy: 0.146, Test Loss: 3.048, Test Accuracy: 0.25
7, Time: 65.32s
## Training epoch 2
## Training Loss: 3.058, Training Accuracy: 0.247, Test Loss: 2.702, Test Accuracy: 0.32
0, Time: 65.42s
## Training epoch 3
## Training Loss: 2.721, Training Accuracy: 0.310, Test Loss: 2.354, Test Accuracy: 0.37
9, Time: 65.68s
## Training epoch 4
## Training Loss: 2.477, Training Accuracy: 0.362, Test Loss: 2.213, Test Accuracy: 0.42
1, Time: 65.86s
## Training epoch 5
## Training Loss: 2.295, Training Accuracy: 0.402, Test Loss: 2.033, Test Accuracy: 0.45
9, Time: 66.00s
## Training epoch 6
## Training Loss: 2.148, Training Accuracy: 0.435, Test Loss: 2.007, Test Accuracy: 0.46
9, Time: 66.07s
## Training epoch 7
## Training Loss: 2.019, Training Accuracy: 0.463, Test Loss: 1.845, Test Accuracy: 0.50
1, Time: 66.17s
## Training epoch 8
## Training Loss: 1.909, Training Accuracy: 0.488, Test Loss: 1.801, Test Accuracy: 0.51
4, Time: 66.30s
## Training epoch 9
## Training Loss: 1.806, Training Accuracy: 0.512, Test Loss: 1.748, Test Accuracy: 0.52
9, Time: 66.42s
## Training epoch 10
## Training Loss: 1.718, Training Accuracy: 0.532, Test Loss: 1.661, Test Accuracy: 0.54
1, Time: 66.69s
## Training epoch 11
## Training Loss: 1.633, Training Accuracy: 0.553, Test Loss: 1.645, Test Accuracy: 0.54
6, Time: 66.82s
## Training epoch 12
## Training Loss: 1.554, Training Accuracy: 0.574, Test Loss: 1.576, Test Accuracy: 0.57
5, Time: 67.53s
## Training epoch 13
## Training Loss: 1.492, Training Accuracy: 0.586, Test Loss: 1.580, Test Accuracy: 0.57
2, Time: 67.98s
## Training epoch 14
## Training Loss: 1.420, Training Accuracy: 0.607, Test Loss: 1.546, Test Accuracy: 0.57
8, Time: 67.97s
## Training epoch 15
## Training Loss: 1.364, Training Accuracy: 0.620, Test Loss: 1.470, Test Accuracy: 0.58
9, Time: 67.86s
## Training epoch 16
## Training Loss: 1.296, Training Accuracy: 0.638, Test Loss: 1.472, Test Accuracy: 0.58
8, Time: 67.36s
## Training epoch 17
## Training Loss: 1.246, Training Accuracy: 0.650, Test Loss: 1.475, Test Accuracy: 0.59
3, Time: 67.66s
```

```
## Training epoch 18
## Training Loss: 1.195, Training Accuracy: 0.665, Test Loss: 1.437, Test Accuracy: 0.59
9, Time: 67.89s
## Training epoch 19
## Training Loss: 1.151, Training Accuracy: 0.674, Test Loss: 1.420, Test Accuracy: 0.60
9, Time: 68.22s
## Training epoch 20
## Training Loss: 1.089, Training Accuracy: 0.691, Test Loss: 1.412, Test Accuracy: 0.61
4, Time: 68.82s
## Training epoch 21
## Training Loss: 1.050, Training Accuracy: 0.702, Test Loss: 1.348, Test Accuracy: 0.62
2, Time: 69.02s
## Training epoch 22
## Training Loss: 1.006, Training Accuracy: 0.714, Test Loss: 1.367, Test Accuracy: 0.62
5, Time: 69.38s
## Training epoch 23
## Training Loss: 0.968, Training Accuracy: 0.721, Test Loss: 1.311, Test Accuracy: 0.63
5, Time: 69.12s
## Training epoch 24
## Training Loss: 0.927, Training Accuracy: 0.735, Test Loss: 1.338, Test Accuracy: 0.63
0, Time: 69.19s
## Training epoch 25
## Training Loss: 0.890, Training Accuracy: 0.744, Test Loss: 1.319, Test Accuracy: 0.63
1, Time: 69.23s
## Training epoch 26
## Training Loss: 0.845, Training Accuracy: 0.758, Test Loss: 1.329, Test Accuracy: 0.63
3, Time: 69.13s
## Training epoch 27
## Training Loss: 0.813, Training Accuracy: 0.767, Test Loss: 1.321, Test Accuracy: 0.63
5, Time: 69.13s
## Training epoch 28
## Training Loss: 0.776, Training Accuracy: 0.778, Test Loss: 1.326, Test Accuracy: 0.63
3, Time: 69.11s
## Training epoch 29
## Training Loss: 0.745, Training Accuracy: 0.787, Test Loss: 1.317, Test Accuracy: 0.64
0, Time: 69.09s
## Training epoch 30
## Training Loss: 0.715, Training Accuracy: 0.794, Test Loss: 1.308, Test Accuracy: 0.64
0, Time: 69.07s
## Training epoch 31
## Training Loss: 0.679, Training Accuracy: 0.806, Test Loss: 1.258, Test Accuracy: 0.65
1, Time: 69.09s
## Training epoch 32
## Training Loss: 0.652, Training Accuracy: 0.812, Test Loss: 1.262, Test Accuracy: 0.65
3, Time: 69.05s
## Training epoch 33
## Training Loss: 0.626, Training Accuracy: 0.822, Test Loss: 1.283, Test Accuracy: 0.65
0, Time: 69.05s
## Training epoch 34
## Training Loss: 0.605, Training Accuracy: 0.828, Test Loss: 1.278, Test Accuracy: 0.65
2, Time: 69.05s
## Training epoch 35
```

```
## Training Loss: 0.577, Training Accuracy: 0.834, Test Loss: 1.287, Test Accuracy: 0.64
8, Time: 69.05s
## Training epoch 36
## Training Loss: 0.548, Training Accuracy: 0.844, Test Loss: 1.276, Test Accuracy: 0.65
4, Time: 69.06s
## Training epoch 37
## Training Loss: 0.528, Training Accuracy: 0.852, Test Loss: 1.251, Test Accuracy: 0.65
9, Time: 69.05s
## Training epoch 38
## Training Loss: 0.511, Training Accuracy: 0.856, Test Loss: 1.258, Test Accuracy: 0.66
0, Time: 69.05s
## Training epoch 39
## Training Loss: 0.494, Training Accuracy: 0.861, Test Loss: 1.314, Test Accuracy: 0.65
2, Time: 69.04s
## Training epoch 40
## Training Loss: 0.464, Training Accuracy: 0.869, Test Loss: 1.297, Test Accuracy: 0.65
2, Time: 69.39s
## Training epoch 41
## Training Loss: 0.465, Training Accuracy: 0.869, Test Loss: 1.299, Test Accuracy: 0.65
5, Time: 69.05s
## Training epoch 42
## Training Loss: 0.432, Training Accuracy: 0.880, Test Loss: 1.278, Test Accuracy: 0.65
6, Time: 69.40s
## Training epoch 43
## Training Loss: 0.425, Training Accuracy: 0.882, Test Loss: 1.268, Test Accuracy: 0.66
0, Time: 69.06s
## Training epoch 44
## Training Loss: 0.404, Training Accuracy: 0.888, Test Loss: 1.261, Test Accuracy: 0.66
4, Time: 69.05s
## Training epoch 45
## Training Loss: 0.388, Training Accuracy: 0.893, Test Loss: 1.298, Test Accuracy: 0.65
9, Time: 69.06s
## Training epoch 46
## Training Loss: 0.376, Training Accuracy: 0.896, Test Loss: 1.253, Test Accuracy: 0.66
8, Time: 69.05s
## Training epoch 47
## Training Loss: 0.368, Training Accuracy: 0.898, Test Loss: 1.293, Test Accuracy: 0.65
8, Time: 69.06s
## Training epoch 48
## Training Loss: 0.350, Training Accuracy: 0.905, Test Loss: 1.247, Test Accuracy: 0.66
7, Time: 69.05s
## Training epoch 49
## Training Loss: 0.337, Training Accuracy: 0.908, Test Loss: 1.279, Test Accuracy: 0.66
1, Time: 69.42s
## Training epoch 50
## Training Loss: 0.330, Training Accuracy: 0.910, Test Loss: 1.276, Test Accuracy: 0.66
4, Time: 69.07s
## Training is complete.
## Average Training Loss: 1.111
## Average Test Loss: 1.505
## Average Training Accuracy: 0.699
```

```
## Average Test Accuracy: 0.594
## Average Training Time per Epoch: 68.23s
```

Code-2: Evaluate the trained model on the test dataset and display the predictions

Import Statements

```
# import statements
import sys
import torch
import torchvision
import matplotlib.pyplot as plt
import numpy as np
from cifar100classify import MyNetwork
```

```
## Using device: cuda
```

```
from torch.utils.data import DataLoader
from torchvision.datasets import CIFAR100
```

Load the names of 100 classes

```
# Function to load CIFAR-100 class names
def load_cifar100_class_names():
    # CIFAR-100 class names
    class_names = [
        'apple', 'aquarium_fish', 'baby', 'bear', 'beaver', 'bed', 'bee', 'beetle',
        'bicycle', 'bottle', 'bowl', 'boy', 'bridge', 'bus', 'butterfly', 'camel',
        'can', 'castle', 'caterpillar', 'cattle', 'chair', 'chimpanzee', 'clock',
        'cloud', 'cockroach', 'couch', 'crab', 'crocodile', 'cup', 'dinosaur',
        'dolphin', 'elephant', 'flatfish', 'forest', 'fox', 'girl', 'hamster',
        'house', 'kangaroo', 'keyboard', 'lamp', 'lawn_mower', 'leopard', 'lion',
        'lizard', 'lobster', 'man', 'maple_tree', 'motorcycle', 'mountain', 'mouse',
        'mushroom', 'oak_tree', 'orange', 'orchid', 'otter', 'palm_tree', 'pear',
        'pickup_truck', 'pine_tree', 'plain', 'plate', 'poppy', 'porcupine', 'possum',
        'rabbit', 'raccoon', 'ray', 'road', 'rocket', 'rose', 'sea', 'seal', 'shark',
        'shrew', 'skunk', 'skyscraper', 'snail', 'snake', 'spider', 'squirrel',
        'streetcar', 'sunflower', 'sweet_pepper', 'table', 'tank', 'telephone',
        'television', 'tiger', 'tractor', 'train', 'trout', 'tulip', 'turtle',
        'wardrobe', 'whale', 'willow_tree', 'wolf', 'woman', 'worm'
    ]
    return class_names
```

Evaluate the model on test dataset images and display image with

predictions and true labels

```
# Model Evaluation
def evaluate_model(model, test_loader):
    # Load CIFAR-100 class names
    class_names = load_cifar100_class_names()

    # Create a 3x3 grid of subplots to visualize images and predictions
    fig, axes = plt.subplots(3, 3, figsize=(8, 8))

    # Iterate over the first 9 examples in the test set
    for i, (image, label) in enumerate(test_loader):
        if i >= 9:  # Only plot the first 9 examples
            break

        # Forward pass through the model to obtain predictions
        output = model(image)
        prediction = torch.argmax(output, dim=1)

        # Undo normalization to display original images
        mean = torch.tensor([0.4914, 0.4822, 0.4465], device=image.device).unsqueeze(0).
unsqueeze(2).unsqueeze(3)
        std = torch.tensor([0.2023, 0.1994, 0.2010], device=image.device).unsqueeze(0).u
nsqueeze(2).unsqueeze(3)
        image = (image * std + mean) * 255
        image = image.permute(0, 2, 3, 1).byte().cpu().numpy()

        # Display the original image
        axes[i // 3, i % 3].imshow(image[0], interpolation='nearest')  # Index [0] to ge
t the first image in the batch
        axes[i // 3, i % 3].set_title(f'Predicted: {class_names[prediction[0].item()]}\n
True: {class_names[label.item()]}')
        axes[i // 3, i % 3].axis('off')

    # Adjust layout and display the plot
    plt.tight_layout()
    plt.show()
```

Main Function

```
# Main function to evaluate the trained model on the test set
def main(argv):
    # Path to the folder containing CIFAR-100 dataset
    ROOT_PATH = '/media/rj/New Volume/Northeastern University/Semester-2/DS 5220 - SMLT/
Projects/Final Project/Code/'

    # Define batch size
    BATCH_SIZE = 1

    num_classes = 100

    # Load the trained network from file and set the model to evaluation mode
    model = MyNetwork(3, num_classes)
    model.load_state_dict(torch.load('./models/model.pth'))
    model.eval()

    # Define the transformation for the test dataset
    test_transform = torchvision.transforms.Compose([
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.20
10))
    ])

    # Load CIFAR-100 test dataset
    test_dataset = CIFAR100(root=ROOT_PATH, download=False, train=False, transform=test_
transform)

    # Create data loader for the test dataset
    test_loader = DataLoader(dataset=test_dataset, num_workers=2, batch_size=BATCH_SIZE,
shuffle=False, pin_memory=True)

    # Call the function to evaluate the model
    evaluate_model(model, test_loader)

if __name__ == "__main__":
    main(sys.argv)
```

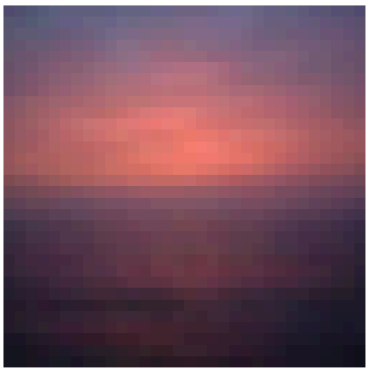
Predicted: road
True: mountain

Predicted: forest
True: forest

Predicted: otter
True: seal

Predicted: otter
True: mushroom

Predicted: sea
True: sea



Predicted: bee
True: tulip

Predicted: camel
True: camel

Predicted: mouse
True: butterfly

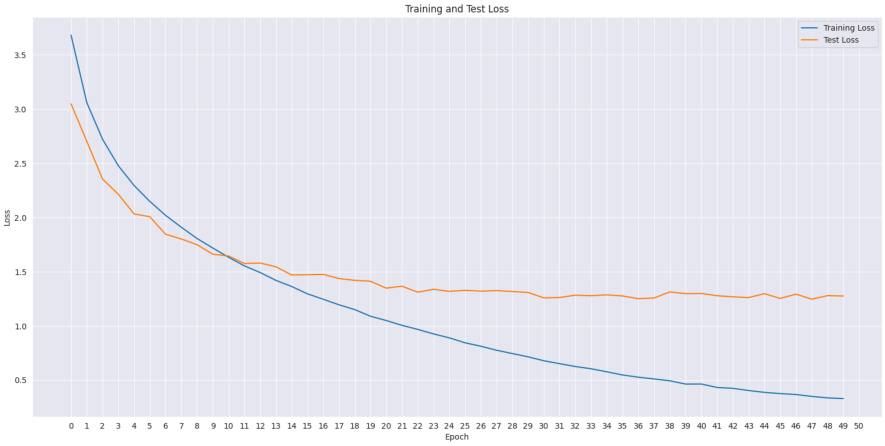
Predicted: sea
True: cloud

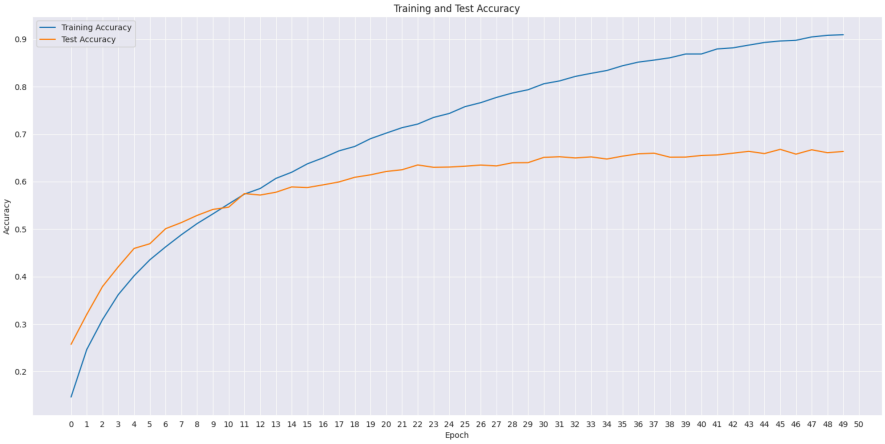
Analysis of results

The accuracy of the trained model on the first 9 images of the test dataset (displayed above) is 33.33%. However, this is not enough to determine the generalization performance of the model. If we shuffle the images in the test dataset, we can get better results. We need extensive testing in order to truly comment about the performance of the model to unseen data.

```
# Load the training loss, test loss, training accuracy and test accuracy curves stored a
s images
import matplotlib.image as mpimg
# Define paths to your images
image_paths = [
    "/media/rj/New Volume/Northeastern University/Semester-2/DS 5220 - SMLT/Projects/Fin
al Project/CUSTOMNETWORKLOSS.png",
    "/media/rj/New Volume/Northeastern University/Semester-2/DS 5220 - SMLT/Projects/Fin
al Project/CUSTOMNETWORKACC.png",
]

# Loop through image paths and display each image
for path in image_paths:
    # Load the image using matplotlib
    img = mpimg.imread(path)

    # Get image dimensions
    height, width, _ = img.shape  # Extract height and width of the image

    # Calculate DPI (dots per inch) for the desired resolution (e.g., 100 DPI)
    dpi = 200

    # Calculate figure size in inches based on image dimensions and DPI
    figsize = (width*4 / dpi, height*4 / dpi)

    # Create a new figure with the specified figure size and DPI
    plt.figure(figsize=figsize, dpi=dpi)

    # Display the image using pyplot
    plt.imshow(img)
    plt.axis('off')  # Hide axes
    plt.show()  # Show the image
```

```
## <Figure size 7680x3636 with 0 Axes>
## <matplotlib.image.AxesImage object at 0x7f46ae79d460>
## (-0.5, 1919.5, 908.5, -0.5)
## <Figure size 7680x3636 with 0 Axes>
## <matplotlib.image.AxesImage object at 0x7f46ae71a160>
## (-0.5, 1919.5, 908.5, -0.5)
```





Based on the observations from the training and test performance graphs of the model, here is a brief analysis of the results:

Training Loss:

- Trend: The training loss steadily decreases throughout the training process, indicating that the model is effectively learning from the data over epochs. The curve exhibits a smooth decline, suggesting that the model is converging towards minimizing the training loss.
- Initial and Final Values: The training loss starts relatively high at 3.681 and reduces significantly to 0.330 after 50 epochs, indicating substantial improvement in model performance.
- Average Loss: The average training loss over the epochs is 1.111, which reflects the overall progress in minimizing loss during training.

Test Loss:

- Trend: The test loss decreases initially but reaches a plateau after around 32 epochs, with minor fluctuations across the training process. The overall trend indicates some improvement in generalization but stabilizes without significant further reduction.
- Initial and Final Values: The test loss starts at 3.048 and decreases to 1.262 after 32 epochs, settling at 1.276 by the end of training.
- Average Loss: The average test loss over epochs is 1.505, which reflects the generalization performance of the model on unseen data.

Training Accuracy:

- Trend: The training accuracy shows a continuous increase across epochs, reaching 91% by the end of training. The curve is relatively smooth with very minor fluctuations, indicating consistent improvement in predicting the training dataset labels.
- Initial and Final Values: The training accuracy starts at a low value of 14.6% and steadily climbs to 91% after 50 epochs, demonstrating the learning capacity of the model.
- Average Accuracy: The average training accuracy is approximately 70%, showcasing the model's ability to learn and generalize from the training data.

Test Accuracy:

- Trend: The test accuracy increases during the initial epochs but stabilizes after about 38 epochs, showing moderate improvement and subsequent stabilization. The test accuracy curve shares similar characteristics with the test loss curve showing minor fluctuations here and there.
- Initial and Final Values: The test accuracy begins at 25.7% and increases to 66% after 38 epochs, settling at 66.4% by the end of training.
- Average Accuracy: The average test accuracy is around 60%, indicating the model's performance on unseen data.

Analysis:

- The model exhibits good training dynamics with consistent reduction in training loss and improvement in training accuracy over epochs.
- The test performance shows some improvement in both loss and accuracy, but the stabilization of these metrics after certain epochs suggests potential limitations in further performance gains or generalization capacity.
- Minor fluctuations in test loss and accuracy indicate potential model sensitivity to specific data characteristics or training dynamics.

Overall, the model demonstrates effective learning capabilities, with significant reduction in loss and improvement in accuracy, albeit with some stabilization in performance metrics towards the later epochs, highlighting areas for potential optimization or further investigation to enhance model generalization. Given the complex nature of the CIFAR-100 dataset, with it's multiple classes which are distinct from one another, and the computation resources at my disposal, this is the best performance I could get on this dataset after experimenting with different neural networks, hyper parameters, etc. But, I believe we can improve the model performance by making changes to the regularization and data augmentation techniques used in the code to improve the model performance. Hyper parameter optimization along with cross validation is an approach which can also be explored.

Method-2: Support Vector Machine (SVM) with Kernel

Introduction

In this method, I have split the execution into four different codes. These codes collectively illustrate a full pipeline for training, performance metric calculations, data preprocessing (label mapping), and utilizing a SVM model for image classification tasks using scikit-learn. I have trained and evaluated the SVM model on the 20 Super classes.

CODE-1: In this code, the CIFAR-100 dataset is loaded from local files, and preprocessing steps are performed. The training data is standardized, and then an SVM with an RBF (Radial Basis Function) kernel is trained on the preprocessed data. The training time of the SVM model is recorded and the trained model is saved to a file.

CODE-2: The trained SVM model is loaded from the saved file. The test data is preprocessed similarly to the training data, and predictions are made in batches using the SVM model. Accuracy scores are calculated for both training and test sets, and the loss (negative mean of the decision function values) is computed for both datasets. Finally, the results are visualized using plots for loss and accuracy over iterations, and the convergence speed of the SVM model is printed.

CODE-3: In this code, the CIFAR-100 dataset and its associated meta-information are loaded from local files. The coarse label names are extracted from the meta-data, and a mapping between fine label IDs and their respective coarse labels is established. This mapping is then saved to a CSV file, containing information about fine label IDs, corresponding coarse label IDs, and coarse label names.

CODE-4: Here, a trained SVM model is loaded, and the CIFAR-100 test dataset is prepared for evaluation. The CSV file created before is utilized to retrieve the mapping between numerical labels and superclass names. The test dataset is standardized, and predictions are made using the SVM model on 9 images from the test set. The predicted labels, true labels, and corresponding images are displayed in a 3x3 grid for visualization.

Code-1: SVM Model Training and saving the model

The reason to comment out this code is that it will take to long to Knit the document since the SVM model takes very long to train.

```
# import numpy as np
# import matplotlib.pyplot as plt
# from sklearn.svm import SVC
# from sklearn.metrics import accuracy_score, log_loss
# from sklearn.model_selection import train_test_split
# from sklearn.preprocessing import StandardScaler
# import pickle
# import time
# import csv
# from tqdm import tqdm
#
# # Load CIFAR-100 dataset from local file
# def load_cifar100(file_path):
#     with open(file_path, 'rb') as f:
#         cifar_data = pickle.load(f, encoding='bytes')
#     return cifar_data
#
# # Specify the path to the CIFAR-100 dataset directory
# directory_path = '/media/rj/New Volume/Northeastern University/Semester-2/DS 5220 - SM
LT/Projects/Final Project/Code/'
#
# # Load CIFAR-100 training dataset
# train_file_path = directory_path + 'cifar-100-python/train'
# cifar100_train_data = load_cifar100(train_file_path)
#
# # Get the mapping between numerical labels and superclass names
# train_label_names = {}
# with open(directory_path + 'train_label_names.csv', 'r') as f:
#     reader = csv.reader(f)
#     next(reader)  # Skip header row
#     for row in reader:
#         fine_label_id = int(row[0])
#         coarse_label_id = int(row[1])
#         coarse_label_name = row[2]
#         train_label_names[fine_label_id] = (coarse_label_id, coarse_label_name)
#
# # Convert fine labels to coarse (superclass) labels
# X_train = cifar100_train_data[b'data'].astype(np.float32) / 255.0  # Normalize pixel v
alues
# train_coarse_labels = np.array([train_label_names[label][0] for label in cifar100_trai
n_data[b'fine_labels']])
# y_train = train_coarse_labels
#
# test_label_names = {}
# with open(directory_path + 'test_label_names.csv', 'r') as f:
#     reader = csv.reader(f)
#     next(reader)  # Skip header row
#     for row in reader:
#         fine_label_id = int(row[0])
#         coarse_label_id = int(row[1])
#         coarse_label_name = row[2]
#         test_label_names[fine_label_id] = (coarse_label_id, coarse_label_name)
```

```
#
# # Load CIFAR-100 testing dataset
# test_file_path = directory_path + 'cifar-100-python/test'
# cifar100_test_data = load_cifar100(test_file_path)
# X_test = cifar100_test_data[b'data'].astype(np.float32) / 255.0  # Normalize pixel val
ues
# test_coarse_labels = np.array([test_label_names[label][0] for label in cifar100_test_d
ata[b'fine_labels']])
# y_test = test_coarse_labels
#
# # Standardize features by removing the mean and scaling to unit variance
# scaler = StandardScaler()
# X_train_scaled = scaler.fit_transform(X_train)
# X_test_scaled = scaler.transform(X_test)
#
# # Define SVM with Kernel
# svm_kernel = SVC(kernel='rbf', random_state=42)
#
# # Training SVM
# start_time = time.time()
# svm_kernel.fit(X_train_scaled, y_train)
# training_time = time.time() - start_time
# print(f"Training time: {training_time:.2f} seconds")
#
# # Save the trained SVM model
# with open('svm_model.pkl', 'wb') as f:
#     pickle.dump(svm_kernel, f)

# Define the path to your text file
file_path = "/media/rj/New Volume/Northeastern University/Semester-2/DS 5220 - SMLT/Proj
ects/Final Project/Code/SVMResults.txt"

# Open the text file in read mode
with open(file_path, 'r') as file:
    # Read all lines from the file
    file_contents = file.readlines()

# Print the contents of the file
for line in file_contents:
    print(line.strip())  # Use strip() to remove any leading/trailing whitespace
```

```
## Training time: 4452.97 seconds
```

Code-2: Loading the SVM model to determine loss and accuracy

The reason to comment out this code is that it will take to long to Knit the document since the SVM model takes very long to make the training and test set predictions to compute the accuracy and loss.

```
# import numpy as np
# import matplotlib.pyplot as plt
# from sklearn.svm import SVC
# from sklearn.metrics import accuracy_score, log_loss
# from sklearn.model_selection import train_test_split
# from sklearn.preprocessing import StandardScaler
# import pickle
# import time
# import csv
#
# # Load CIFAR-100 dataset from local file
# def load_cifar100(file_path):
#     with open(file_path, 'rb') as f:
#         cifar_data = pickle.load(f, encoding='bytes')
#     return cifar_data
#
# # Load the trained SVM model
# with open('svm_model.pkl', 'rb') as f:
#     svm_kernel = pickle.load(f)
#
# directory_path = '/media/rj/New Volume/Northeastern University/Semester-2/DS 5220 - SM
LT/Projects/Final Project/Code/'
# train_file_path = directory_path + 'cifar-100-python/train'
# cifar100_train_data = load_cifar100(train_file_path)
#
# # Get the mapping between numerical labels and superclass names
# train_label_names = {}
# with open(directory_path + 'train_label_names.csv', 'r') as f:
#     reader = csv.reader(f)
#     next(reader)  # Skip header row
#     for row in reader:
#         fine_label_id = int(row[0])
#         coarse_label_id = int(row[1])
#         coarse_label_name = row[2]
#         train_label_names[fine_label_id] = (coarse_label_id, coarse_label_name)
#
# # Convert fine labels to coarse (superclass) labels
# X_train = cifar100_train_data[b'data'].astype(np.float32) / 255.0  # Normalize pixel v
alues
# train_coarse_labels = np.array([train_label_names[label][0] for label in cifar100_trai
n_data[b'fine_labels']])
# y_train = train_coarse_labels
#
# test_label_names = {}
# with open(directory_path + 'test_label_names.csv', 'r') as f:
#     reader = csv.reader(f)
#     next(reader)  # Skip header row
#     for row in reader:
#         fine_label_id = int(row[0])
#         coarse_label_id = int(row[1])
#         coarse_label_name = row[2]
#         test_label_names[fine_label_id] = (coarse_label_id, coarse_label_name)
```

```
#
# # Load CIFAR-100 testing dataset
# test_file_path = directory_path + 'cifar-100-python/test'
# cifar100_test_data = load_cifar100(test_file_path)
# X_test = cifar100_test_data[b'data'].astype(np.float32) / 255.0  # Normalize pixel val
ues
# test_coarse_labels = np.array([test_label_names[label][0] for label in cifar100_test_d
ata[b'fine_labels']])
# y_test = test_coarse_labels
#
# # Standardize features by removing the mean and scaling to unit variance
# scaler = StandardScaler()
# X_train_scaled = scaler.fit_transform(X_train)
# X_test_scaled = scaler.transform(X_test)
#
# # Predictions
# batch_size = 1024
# n_batches = int(np.ceil(X_train_scaled.shape[0] / batch_size))
# train_pred = []
#
# for batch in range(n_batches):
#     start = batch * batch_size
#     end = min(start + batch_size, X_train_scaled.shape[0])
#     X_batch = X_train_scaled[start:end]
#     pred_batch = svm_kernel.predict(X_batch)
#     train_pred.extend(pred_batch)
#
# train_pred = np.array(train_pred)
#
# # Predictions
# batch_size_test = 2048
# n_batches_test = int(np.ceil(X_test_scaled.shape[0] / batch_size_test))
# test_pred = []
#
# for batch in range(n_batches_test):
#     start = batch * batch_size_test
#     end = min(start + batch_size_test, X_test_scaled.shape[0])
#     X_batch = X_test_scaled[start:end]
#     pred_batch = svm_kernel.predict(X_batch)
#     test_pred.extend(pred_batch)
#
# test_pred = np.array(test_pred)
#
# # Calculate accuracy
# train_accuracy = accuracy_score(y_train, train_pred)
# test_accuracy = accuracy_score(y_test, test_pred)
#
# # Calculate loss (using decision function values for SVM)
# train_loss = -svm_kernel.decision_function(X_train_scaled).mean()
# test_loss = -svm_kernel.decision_function(X_test_scaled).mean()
#
# # Get decision function values
```

```
# decision_values = svm_kernel.decision_function(X_train_scaled)
#
# # Plot results
# plt.figure(figsize=(12, 6))
#
# # Training Loss
# plt.subplot(1, 2, 1)
# plt.plot([train_loss]*len(X_train_scaled), label='Train')
# plt.plot([test_loss]*len(X_test_scaled), label='Test')
# plt.title('Loss')
# plt.xlabel('Iterations')
# plt.ylabel('Loss')
# plt.legend()
#
# # Training Accuracy
# plt.subplot(1, 2, 2)
# plt.plot(train_accuracy, label='Train')
# plt.plot(test_accuracy, label='Test')
# plt.title('Accuracy')
# plt.xlabel('Iterations')
# plt.ylabel('Accuracy')
# plt.legend()
#
# plt.tight_layout()
# plt.show()
#
# print("Convergence speed (change in decision function values):", np.mean(np.diff(decis
ion_values)))
```

Code-3: Mapping the CIFAR-100 labels to the 20 Superclasses

The reason to comment out this code is that it is printing a very long list of index values when I knit the document. The output document will be unnecessarily too lengthy.

```
# import pickle
# import csv
#
# def load_cifar100(file_path):
#     with open(file_path, 'rb') as f:
#         cifar_data = pickle.load(f, encoding='bytes')
#     return cifar_data
#
# # Specify the path to the CIFAR-100 dataset directory
# directory_path = '/media/rj/New Volume/Northeastern University/Semester-2/DS 5220 - SM
LT/Projects/Final Project/Code/cifar-100-python/'
#
# # Load the meta file
# meta_path = directory_path + 'meta'
# meta_data = load_cifar100(meta_path)
#
# # Get the coarse label names
# coarse_label_names = [label.decode('utf-8') for label in meta_data[b'coarse_label_name
s']]
#
# # Load the train file to get the mapping between fine and coarse labels
# train_file_path = directory_path + 'test'
# train_data = load_cifar100(train_file_path)
# coarse_labels = train_data[b'coarse_labels']
#
# # Get the fine label IDs by iterating over the length of the data
# num_examples = len(train_data[b'data'])
# fine_label_ids = list(range(num_examples))
#
# # Construct the label_names dictionary
# label_names = {}
# for fine_id, coarse_id in zip(fine_label_ids, coarse_labels):
#     coarse_name = coarse_label_names[coarse_id]
#     label_names[fine_id] = (coarse_id, coarse_name)
#
# # Save the label_names dictionary to a CSV file
# with open('test_label_names.csv', 'w', newline='') as csvfile:
#     fieldnames = ['fine_label_id', 'coarse_label_id', 'coarse_label_name']
#     writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
#
#     writer.writeheader()
#     for fine_id, (coarse_id, coarse_name) in label_names.items():
#         writer.writerow({'fine_label_id': fine_id, 'coarse_label_id': coarse_id, 'coar
se_label_name': coarse_name})
```

Code-4: Loading the model to classify 9 images in

the test dataset

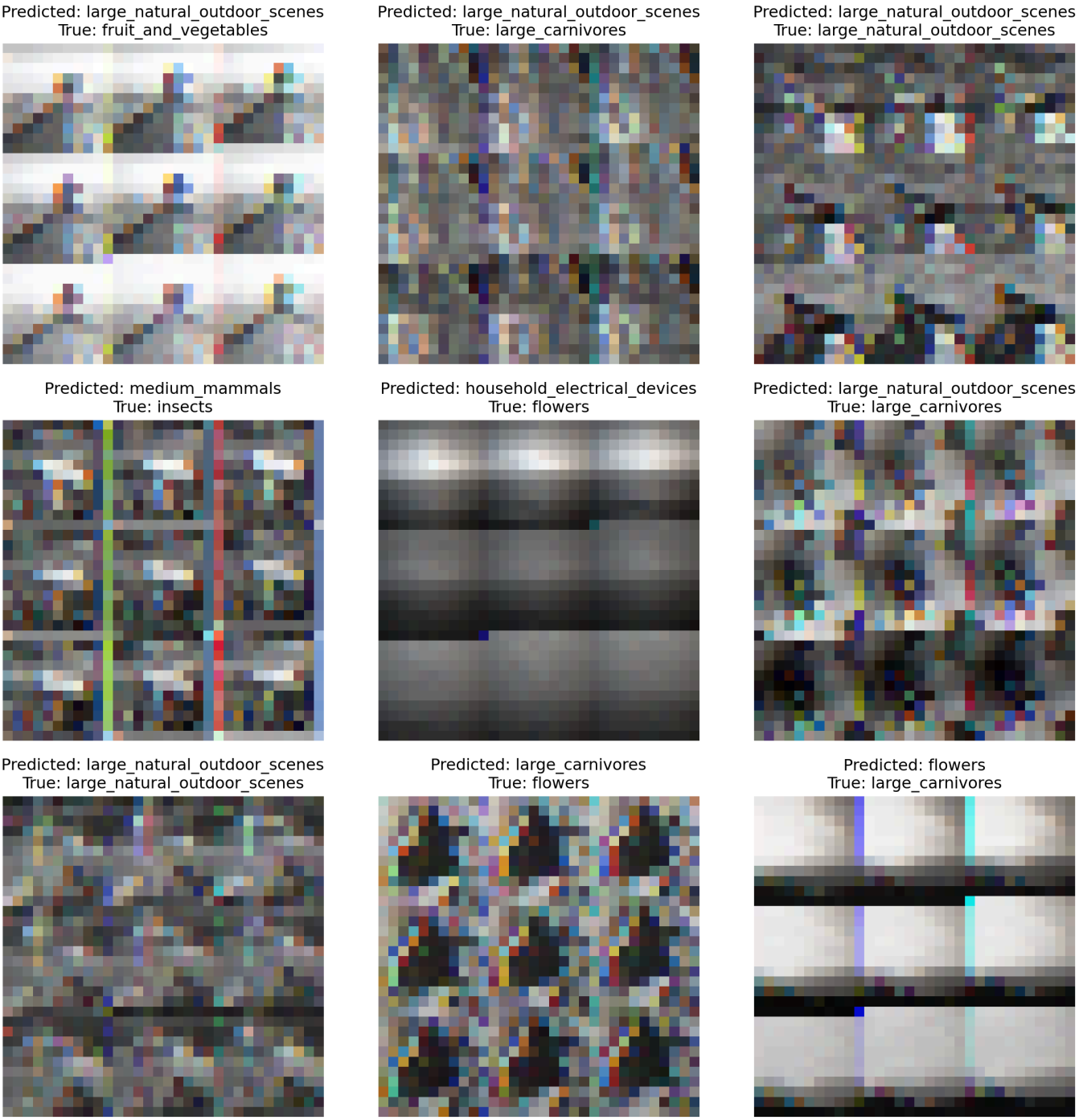
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVC
import pickle
from sklearn.preprocessing import StandardScaler
import csv

def load_cifar100(file_path):
    with open(file_path, 'rb') as f:
        cifar_data = pickle.load(f, encoding='bytes')
    return cifar_data

# Specify the path to the CIFAR-100 dataset directory
directory_path = '/media/rj/New Volume/Northeastern University/Semester-2/DS 5220 - SML
T/Projects/Final Project/Code/'

# Load the trained SVM model
with open('svm_model.pkl', 'rb') as f:
    svm_kernel = pickle.load(f)

# Load CIFAR-100 testing dataset
test_file_path = directory_path + 'cifar-100-python/test'
cifar100_test_data = load_cifar100(test_file_path)
X_test = cifar100_test_data[b'data'].astype(np.float32) / 255.0  # Normalize pixel value
s
y_test = np.array(cifar100_test_data[b'fine_labels'])

# Get the mapping between numerical labels and superclass names
# Load the label names from the csv file
label_names = {}
with open(directory_path + 'test_label_names.csv', 'r') as f:
    reader = csv.reader(f)
    next(reader)  # Skip header row
    for row in reader:
        fine_label_id = int(row[0])
        coarse_label_id = int(row[1])
        coarse_label_name = row[2]
        label_names[fine_label_id] = (coarse_label_id, coarse_label_name)
```

```
## ['fine_label_id', 'coarse_label_id', 'coarse_label_name']
```

```
# Convert fine labels to coarse (superclass) labels
coarse_labels = np.array([label_names[label][0] for label in y_test])

# Standardize features
scaler = StandardScaler()
X_test_scaled = scaler.fit_transform(X_test)

# Make predictions on the 9 images from the test set
test_pred = svm_kernel.predict(X_test_scaled[:9])

# Display the predicted label, true label, and the image
fig, axs = plt.subplots(3, 3, figsize=(12, 12))
for i, ax in enumerate(axs.flat):
    ax.imshow(X_test[i].reshape(32, 32, 3))
    predicted_label = label_names[test_pred[i]][1]
    true_label = label_names[coarse_labels[i]][1]
    ax.set_title(f"Predicted: {predicted_label}\nTrue: {true_label}")
    ax.axis('off')
```

```
## <matplotlib.image.AxesImage object at 0x7f45ae584bb0>
## Text(0.5, 1.0, 'Predicted: large_natural_outdoor_scenes\nTrue: fruit_and_vegetables')
## (-0.5, 31.5, 31.5, -0.5)
## <matplotlib.image.AxesImage object at 0x7f45ae965070>
## Text(0.5, 1.0, 'Predicted: large_natural_outdoor_scenes\nTrue: large_carnivores')
## (-0.5, 31.5, 31.5, -0.5)
## <matplotlib.image.AxesImage object at 0x7f45ae744c70>
## Text(0.5, 1.0, 'Predicted: large_natural_outdoor_scenes\nTrue: large_natural_outdoor_
scenes')
## (-0.5, 31.5, 31.5, -0.5)
## <matplotlib.image.AxesImage object at 0x7f45ae702a90>
## Text(0.5, 1.0, 'Predicted: medium_mammals\nTrue: insects')
## (-0.5, 31.5, 31.5, -0.5)
## <matplotlib.image.AxesImage object at 0x7f45ae6bf8b0>
## Text(0.5, 1.0, 'Predicted: household_electrical_devices\nTrue: flowers')
## (-0.5, 31.5, 31.5, -0.5)
## <matplotlib.image.AxesImage object at 0x7f45ae67d700>
## Text(0.5, 1.0, 'Predicted: large_natural_outdoor_scenes\nTrue: large_carnivores')
## (-0.5, 31.5, 31.5, -0.5)
## <matplotlib.image.AxesImage object at 0x7f45ae63c1f0>
## Text(0.5, 1.0, 'Predicted: large_natural_outdoor_scenes\nTrue: large_natural_outdoor_
scenes')
## (-0.5, 31.5, 31.5, -0.5)
## <matplotlib.image.AxesImage object at 0x7f45ae678130>
## Text(0.5, 1.0, 'Predicted: large_carnivores\nTrue: flowers')
## (-0.5, 31.5, 31.5, -0.5)
## <matplotlib.image.AxesImage object at 0x7f45ae634040>
## Text(0.5, 1.0, 'Predicted: flowers\nTrue: large_carnivores')
## (-0.5, 31.5, 31.5, -0.5)
```

```
plt.tight_layout()
plt.show()
```



Analysis of results

The accuracy of the trained model on 9 images of the test dataset (displayed above) is 22.22%. Same as in CNNs, we need extensive testing in order to truly comment about the performance of the model to unseen data.

The model training time of approximately 4453 seconds (about 74 minutes) for the SVM with an RBF kernel on the CIFAR-100 dataset is quite significant. There are many reasons for this:

- **Dataset Size:** The CIFAR-100 dataset is relatively large, containing 60,000 images for training and 10,000 images for testing. Each image is represented by a 32x32 RGB pixel array.
- **Feature Standardization:** Before training the SVM, the features (pixel values) are standardized using StandardScaler(). This step involves computing the mean and standard deviation of each feature across the dataset, which can be time-consuming for large datasets.
- **Model Complexity:** The choice of an SVM with an RBF (Radial Basis Function) kernel can contribute to longer training times, especially as the dataset size increases. The RBF kernel computes the similarity between pairs of samples in a high-dimensional space, which can be computationally intensive.

Comparison: CNN V/S SVM Kernel

Performance Metric	Convolutional Neural Network	SVM with Kernel
Accuracy	The accuracy on the test dataset is 33.33%	The accuracy on the test dataset is 22.22%
Convergence Speed	The convergence speed of the network is relatively fast, as evident from the smooth and consistent decline in training loss across epochs. The training accuracy also exhibits a continuous increase, reaching 91% within 50 epochs. However, the test metrics stabilize after around 32-38 epochs, indicating that the model's generalization ability reaches a plateau. The average training time per epoch is 68.23 seconds, which is reasonable for the given model complexity and dataset size.	The SVM model exhibits a very slow convergence speed, as indicated by the total training time for the SVM model, which is 4452.97 seconds (1 hour 14 minutes 12 seconds).

The above comparison between the Convolutional Neural Network (CNN) and the Support Vector Machine (SVM) with a kernel highlights key differences in their performance and convergence characteristics. While the CNN achieves a higher accuracy of 33.33% on the test dataset compared to the SVM's 22.22%, it also exhibits a significantly faster convergence speed. The CNN's training loss decreases smoothly, and its training accuracy reaches 91% within 50 epochs, with an average training time of 68.23 seconds per epoch. In contrast, the SVM model exhibits a very slow convergence speed, taking 74 minutes to train, which may be attributed to the choice of kernel function and the complexity of the dataset. The CNN's faster convergence and higher accuracy can be attributed to its ability to automatically learn hierarchical representations from the input data, making it more effective for complex tasks compared to the SVM, which relies on manually engineered features and kernel functions.

To conclude, I would choose the Convolutional Neural Networks over Support Vector Machines for complex tasks like image classfication involving large datasets like CIFAR-100. The performance of CNNs in terms of accuracy, generalization and convergence is far superior than that of SVMs.