class-imbalancing-mlp-on-cifar

May 27, 2024

1 Class Imbalancing in Multilayer perceptrons

This code will:

1.Load the CIFAR-10 dataset.
2.Create an imbalanced training set.
3.Apply SMOTE to balance the dataset.
4.Train a CNN on the CIFAR-10 dataset.
5.Extract features from the output layer of the trained CNN.
6.Train an MLP using these extracted features.
7.Test the MLP on the CIFAR-10 test dataset and print the accuracy.

[11]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, Subset
from imblearn.over_sampling import SMOTE
import numpy as np
```

1.0.1 Step 1: Load CIFAR 10 dataset

[12]:
```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])
train_dataset = datasets.CIFAR10('data', train=True, download=True,␣
 ↪transform=transform)
test_dataset = datasets.CIFAR10('data', train=False, transform=transform)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
data/cifar-10-python.tar.gz

100%|      | 170498071/170498071 [00:02<00:00, 84772974.95it/s]

Extracting data/cifar-10-python.tar.gz to data
```

1.0.2 Step 2: Apply imbalancing function for class imbalance

[13]:
```
# Function to create an imbalanced dataset
def get_imbalanced_data(dataset, num_sample_per_class, shuffle=False,␣
 ↪random_seed=0):
    """
    Return a list of imbalanced indices from a dataset.
    Input: A dataset (e.g., CIFAR-10), num_sample_per_class: list of integers
    Output: imbalanced_list
    """
    length = len(dataset)
    num_sample_per_class = list(num_sample_per_class)
    selected_list = []
    indices = list(range(0, length))

    if shuffle:
        np.random.seed(random_seed)
        np.random.shuffle(indices)

    for i in range(0, length):
        index = indices[i]
        _, label = dataset[index]
        if num_sample_per_class[label] > 0:
            selected_list.append(index)
            num_sample_per_class[label] -= 1
    return selected_list

# Create imbalance in the dataset
num_samples = [5000, 3000, 2000, 1000, 500, 200, 100, 50, 20, 10]
imbalanced_indices = get_imbalanced_data(train_dataset, num_samples)
train_dataset = Subset(train_dataset, imbalanced_indices)
```

1.0.3 Step 3: Apply SMOTE for balancing

[14]:
```
# Note: SMOTE is usually applied on feature vectors, so we'll need to flatten␣
 ↪the images
# Convert images to numpy arrays for SMOTE
train_images = [img.numpy().flatten() for img, _ in train_dataset]
train_labels = [label for _, label in train_dataset]
smote = SMOTE(random_state=42)
train_images_resampled, train_labels_resampled = smote.
 ↪fit_resample(train_images, train_labels)
train_images_resampled = torch.Tensor(np.array(train_images_resampled)).
 ↪view(-1, 3, 32, 32)
train_labels_resampled = torch.LongTensor(train_labels_resampled)
```

[15]:
```
# # Store flattened vectors for train data
# X_train = train_images_resampled.view(-1, 28 * 28)
# Y_train = train_labels_resampled
```

1.0.4 Step 4: Build CNN & Train

[16]:
```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3)
        self.fc1 = nn.Linear(64 * 6 * 6, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2(x), 2))
        x = x.view(-1, 64 * 6 * 6)
        x = F.relu(self.fc1(x))
        logits = self.fc2(x)
        return logits, x  # Return logits and features from the penultimate␣
 ↪layer

cnn = CNN()
# Define optimizer and loss function
optimizer_cnn = optim.Adam(cnn.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()

print("Training the CNN ... ")

# Train the CNN
def train_cnn(model, train_loader, optimizer, criterion, epochs=10):
    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        for data, target in train_loader:
            optimizer.zero_grad()
            logits, _ = model(data)
            loss = criterion(logits, target)
            loss.backward()
            optimizer.step()
            running_loss += loss.item() * data.size(0)
        print(f"Epoch {epoch+1}/{epochs}, Loss: {running_loss/len(train_loader.
 ↪dataset)}")

train_loader_cnn = DataLoader(list(zip(train_images_resampled,␣
 ↪train_labels_resampled)), batch_size=64, shuffle=True)
```

```
train_cnn(cnn, train_loader_cnn, optimizer_cnn, criterion)
```

```
Training the CNN …
Epoch 1/10, Loss: 0.681249429860115
Epoch 2/10, Loss: 0.25750763940811155
Epoch 3/10, Loss: 0.14672491289377212
Epoch 4/10, Loss: 0.09565989250063896
Epoch 5/10, Loss: 0.06459150330781936
Epoch 6/10, Loss: 0.04818558646805585
Epoch 7/10, Loss: 0.03930638539776206
Epoch 8/10, Loss: 0.03287420496612787
Epoch 9/10, Loss: 0.029538248270601034
Epoch 10/10, Loss: 0.017137879557819105
```

[17]:
```
# # Store flattened vectors for test data
# X_test = test_images.view(-1, 28 * 28)
# Y_test = test_labels
```

1.0.5 Step 5: Build a Multi-layer Perceptron

[18]:
```
#Extract features from the CNN to feed into the MLP
cnn.eval()
with torch.no_grad():
    train_features = []
    for data, _ in train_loader_cnn:
        _, features = cnn(data)
        train_features.append(features)
    train_features = torch.cat(train_features)

#Build a Multi-layer Perceptron
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(128, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

mlp = MLP()
# Define optimizer and loss function
optimizer_mlp = optim.Adam(mlp.parameters(), lr=0.001)
criterion_mlp = nn.CrossEntropyLoss()
```

```
print("Training the MLP ... ")
```

```
Training the MLP …
```

1.0.6 Step 6: Apply cross-entropy loss for loss calculation

[19]:
```
# Training the MLP
def train_mlp(model, train_loader, optimizer, criterion, epochs=10):
    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        for data, target in train_loader:
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            running_loss += loss.item() * data.size(0)
        print(f"Epoch {epoch+1}/{epochs}, Loss: {running_loss/len(train_loader.
 ↪dataset)}")

train_loader_mlp = DataLoader(list(zip(train_features,␣
 ↪train_labels_resampled)), batch_size=64, shuffle=True)
train_mlp(mlp, train_loader_mlp, optimizer_mlp, criterion_mlp)
```

```
Epoch 1/10, Loss: 2.308327042312622
Epoch 2/10, Loss: 2.302695669631958
Epoch 3/10, Loss: 2.302870717163086
Epoch 4/10, Loss: 2.301990905075073
Epoch 5/10, Loss: 2.301452420425415
Epoch 6/10, Loss: 2.3004353280639647
Epoch 7/10, Loss: 2.2984436138916013
Epoch 8/10, Loss: 2.2964931101226806
Epoch 9/10, Loss: 2.292941941986084
Epoch 10/10, Loss: 2.288054637680054
```

1.0.7 Step 8: Calculate final accuracy

[20]:
```
def test(model, test_loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for data, target in test_loader:
            output = model(data)
            _, predicted = torch.max(output.data, 1)
```

```
            total += target.size(0)
            correct += (predicted == target).sum().item()
    accuracy = correct / total
    return accuracy

test_loader_cnn = DataLoader(test_dataset, batch_size=64, shuffle=False)
test_features = []
test_labels = []
cnn.eval()
with torch.no_grad():
    for data, labels in test_loader_cnn:
        _, features = cnn(data)
        test_features.append(features)
        test_labels.append(labels)
test_features = torch.cat(test_features)
test_labels = torch.cat(test_labels)

test_loader_mlp = DataLoader(list(zip(test_features, test_labels)),␣
 ↪batch_size=64, shuffle=False)
accuracy = test(mlp, test_loader_mlp)

print(f"MLP Test Accuracy: {accuracy*100}")
```

```
MLP Test Accuracy: 9.2
```