mlp-on-mnist

May 25, 2024

1 Class Imbalancing in Multilayer perceptrons

[13]:
```python
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 MNIST dataset

[14]:
```python
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

train_dataset = datasets.MNIST('data', train=True, download=True,␣
 ↪transform=transform)
test_dataset = datasets.MNIST('data', train=False, transform=transform)
```

1.0.2 Step 2: Apply imbalancing function for class imbalance

[15]:
```python
# 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

[16]:
```
# 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, 1, 28, 28)
train_labels_resampled = torch.LongTensor(train_labels_resampled)
```

[17]:
```
# 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

[18]:
```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3)
```

```
        self.fc1 = nn.Linear(64 * 5 * 5, 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 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

cnn = CNN()

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

print(" ")
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()
            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 = DataLoader(list(zip(train_images_resampled,␣
 ↪train_labels_resampled)), batch_size=64, shuffle=True)
train_cnn(cnn, train_loader, optimizer, criterion)

test_images = [img.numpy().flatten() for img, _ in test_dataset]
test_labels = [label for _, label in test_dataset]

# Convert numpy arrays to a single numpy array for test data
test_images = torch.Tensor(np.array(test_images)).view(-1, 1, 28, 28)
test_labels = torch.LongTensor(test_labels)
```

```
Training the CNN ...
Epoch 1/10, Loss: 0.08041252863250672
Epoch 2/10, Loss: 0.008443046596841886
Epoch 3/10, Loss: 0.005568267855094746
Epoch 4/10, Loss: 0.0030876084698003253
Epoch 5/10, Loss: 0.0035641780086456856
Epoch 6/10, Loss: 0.002844212014042423
Epoch 7/10, Loss: 0.0007680557021931691
Epoch 8/10, Loss: 0.0003714925435681198
Epoch 9/10, Loss: 0.004911134818668215
Epoch 10/10, Loss: 0.0014430014268705508
```

[19]:

```
# 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

[20]:

```
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(28 * 28, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(-1, 28 * 28)
        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(" ")
print("Training the MLP ... ")
```

```
Training the MLP ...
```

1.0.6 Step 7: Apply cross-entropy loss for loss calculation

[21]:
```
# 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_images_resampled,␣
↪train_labels_resampled)), batch_size=64, shuffle=True)
train_mlp(mlp, train_loader_mlp, optimizer_mlp, criterion_mlp)
```

```
Epoch 1/10, Loss: 0.09466223964802921
Epoch 2/10, Loss: 0.013806616121723783
Epoch 3/10, Loss: 0.008401836839234456
Epoch 4/10, Loss: 0.008601906947477256
Epoch 5/10, Loss: 0.004532016910217936
Epoch 6/10, Loss: 0.005527359090443788
Epoch 7/10, Loss: 0.0052575174327231435
Epoch 8/10, Loss: 0.00412335330583388
Epoch 9/10, Loss: 0.0024656502844864463
Epoch 10/10, Loss: 0.0008559695080666643
```

1.0.7 Step 8: Calculate final accuracy

[22]:
```
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_mlp = DataLoader(list(zip(test_images, test_labels)),␣
    ↪batch_size=64, shuffle=False)
accuracy = test(mlp, test_loader_mlp)
print(" ")
print(f"MLP Test Accuracy: {accuracy*100}")
```

```
MLP Test Accuracy: 80.25999999999999
```