train_model_sch_lr_es_strat

May 27, 2024

[1]:
```python
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import DataLoader, random_split
from torch.utils.tensorboard import SummaryWriter
from bird_song_dataset import BirdSongDataset, DataPaths, DeviceManager
from torchvision.transforms import ToTensor
from torch.optim.lr_scheduler import StepLR
from sklearn.model_selection import train_test_split
from datetime import datetime
```

[2]:
```python
class SimpleCNN(nn.Module):
    """
    CNN model for image classification

    This network consists of two convolutional layers followed by two fully␣
↪connected layers
    The network uses ReLU activation functions for non-linearity and max␣
↪pooling for down-sampling

    Nueral net architecture:
        - conv1:
            The first convolutional layer holds 16 filters, a kernel size of 3,␣
↪stride of 1, and padding of 1
        - conv2:
            The second convolutional layer holds 32 filters, a kernel size of␣
↪3, stride of 1, and padding of 1
        - fc1:
            The first fully connected layer that maps from the flattened output␣
↪of the last pooling layer to 512 features
        - fc2:
            The second fully connected layer that maps the 512 features to the␣
↪number of classes

    The forward method defines the data flow through the network, applying␣
↪layers sequentially with ReLU activation functions and pooling operations
```

```
    """

    def __init__(self, num_classes=5):
        super(SimpleCNN, self).__init__()
        # First convolutional layer with 16 filters
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1)
        # Second convolutional layer with 32 filters
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        # First fully connected layer, transforming the feature map from␣
↪convolutional layers into a 512-dimensional vector
        self.fc1 = nn.Linear(32768, 512)
        # Final fully connected layer that outputs probability distribution␣
↪across the classes
        self.fc2 = nn.Linear(512, num_classes)

    def forward(self, x):
        # Apply the first convolutional layer followed by ReLU activation and␣
↪max pooling
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2)
        # Apply the second convolutional layer followed by ReLU activation and␣
↪another max pooling
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2)
        # Flatten the output from the convolutional layers to prepare for the␣
↪fully connected layer
        x = torch.flatten(x, 1)
        # Apply the first fully connected layer with ReLU activation
        x = F.relu(self.fc1(x))
        # Output layer that maps to the number of classes
        x = self.fc2(x)

        return x
```

[3]:
```
# Get dynamic paths
data_paths = DataPaths()
paths = data_paths.get_paths()
print(paths.keys())
```

```
dict_keys(['csv_file_path', 'wav_files_dir', 'models_dir', 'results_dir',
'runs_dir'])
```

[4]:
```
# Instantiate dataset class
bird_dataset = BirdSongDataset(csv_file=paths['csv_file_path'],␣
↪root_dir=paths['wav_files_dir'])
print(f"Dataset size: {len(bird_dataset)}")
```

```
Dataset size: 5422
```

[5]:
```
train_size = int(0.7 * len(bird_dataset))
val_size = int(0.15 * len(bird_dataset))
test_size = len(bird_dataset) - train_size - val_size

print(f'Data split sizes for train, val, and test: {train_size, val_size,␣
↪test_size}')
```

```
Data split sizes for train, val, and test: (3795, 813, 814)
```

[6]:
```
# Get the labels from the dataset for stratification
labels = bird_dataset.labels
labels
```

[6]:
```
array([1, 1, 1, …, 2, 2, 2])
```

[7]:
```
# Stratified split
train_indices, temp_indices, train_strat_labels, temp_strat_labels = \
    train_test_split(range(len(bird_dataset)), labels, stratify=labels,␣
↪test_size=val_size + test_size, random_state=42)

val_indices, test_indices, val_strat_labels, test_strat_labels = \
    train_test_split(temp_indices, temp_strat_labels,␣
↪stratify=temp_strat_labels, test_size=test_size / (val_size + test_size),␣
↪random_state=42)

# Create DataLoaders for the stratified subsets
train_loader = DataLoader(torch.utils.data.Subset(bird_dataset, train_indices),␣
↪batch_size=64, shuffle=True)
val_loader = DataLoader(torch.utils.data.Subset(bird_dataset, val_indices),␣
↪batch_size=64, shuffle=False)
test_loader = DataLoader(torch.utils.data.Subset(bird_dataset, test_indices),␣
↪batch_size=64, shuffle=False)

# Determine accelerator device
device_manager = DeviceManager()
device = device_manager.device
print(device)
```

```
Using MPS (Apple Silicon GPU)
mps
```

[8]:
```
# Define the model, loss function, optimizer, and learning rate scheduler
model = SimpleCNN(num_classes=5).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```
scheduler = StepLR(optimizer, step_size=3, gamma=0.1)

# Initialize TensorBoard writer for logging
writer = SummaryWriter(f"{paths['runs_dir']}/
↪bird_song_experiment_with_scheduler_early_stopping_stratified_splits")

# Initialize variables for early stopping mechanism
patience = 3
best_val_loss = float('inf')
epochs_no_improve = 0
early_stop = False

# Set number of epochs for training
num_epochs = 25
for epoch in range(num_epochs):
    # Set model to training mode and initialize running loss
    model.train()
    running_loss = 0.0

    # Loop over batches in the training dataset
    for batch in train_loader:
        inputs, labels = batch['spectrogram'].to(device), batch['label'].
↪to(device)
        # Zero the gradients
        optimizer.zero_grad()
        # Forward pass
        outputs = model(inputs)
        # Compute loss
        loss = criterion(outputs, labels)
        # Backward pass
        loss.backward()
        # Update parameters
        optimizer.step()
        # Accumulate the loss
        running_loss += loss.item()

    # Compute and log training loss
    training_loss = running_loss / len(train_loader)
    writer.add_scalar('Loss/train', training_loss, epoch)

    # Set model to evaluation mode and compute validation loss
    model.eval()
    val_loss = 0.0
    with torch.no_grad():
        # Loop over batches in the validation dataset
        for batch in val_loader:
            # Extract inputs and labels from the batch
```

```python
            inputs, labels = batch['spectrogram'].to(device), batch['label'].
↪to(device)
            # Forward pass: compute model output
            outputs = model(inputs)
            # Compute loss
            loss = criterion(outputs, labels)
            # Accumulate the validation loss over all of the batches
            val_loss += loss.item()

    # Logging the validation loss
    validation_loss = val_loss / len(val_loader)
    writer.add_scalar('Loss/validation', validation_loss, epoch)

    # Check for early stopping
    if validation_loss < best_val_loss:
        best_val_loss = validation_loss
        epochs_no_improve = 0

        # Format the current date and time as a string
        timestamp = datetime.now().strftime('%Y-%m-%d_%H-%M-%S')

        # Save the model with the timestamp in the filename
        torch.save(model.state_dict(), f"{paths['models_dir']}/
↪model_sch_lr_es_start_{timestamp}.pth")

    else:
        epochs_no_improve += 1
        if epochs_no_improve >= patience:
            print(f'Early stopping triggered after {epoch + 1} epochs!')
            early_stop = True
            break

    # Step the scheduler for learning rate adjustment
    scheduler.step()

    # Log training progress and learning rate
    current_lr = scheduler.get_last_lr()[0]
    writer.add_scalar('Learning Rate', current_lr, epoch)
    print(f"Epoch {epoch+1}/{num_epochs}, Training Loss: {training_loss},␣
↪Validation Loss: {validation_loss}")
    print("-" * 75)

# Check if training stopped early and close TensorBoard writer
if not early_stop:
    print(f"Training completed after {num_epochs} epochs.")
writer.close()
```

```
Epoch 1/25, Training Loss: 39.100919245680174, Validation Loss:
0.9216846181796148
---------------------------------------------------------------------------
Epoch 2/25, Training Loss: 0.72219214985768, Validation Loss: 0.7543656780169561
---------------------------------------------------------------------------
Epoch 3/25, Training Loss: 0.4131690027813117, Validation Loss:
0.6712680481947385
---------------------------------------------------------------------------
Epoch 4/25, Training Loss: 0.19353991510967414, Validation Loss:
0.6298494889185979
---------------------------------------------------------------------------
Epoch 5/25, Training Loss: 0.15486942554513614, Validation Loss:
0.6233967130000775
---------------------------------------------------------------------------
Epoch 6/25, Training Loss: 0.13023976398011047, Validation Loss:
0.6203022255347326
---------------------------------------------------------------------------
Epoch 7/25, Training Loss: 0.11128499197463194, Validation Loss:
0.6215839638159826
---------------------------------------------------------------------------
Epoch 8/25, Training Loss: 0.10903202394644419, Validation Loss:
0.6240699703876789
---------------------------------------------------------------------------
Early stopping triggered after 9 epochs!
```