train_model_fixed_lr

May 27, 2024

[1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import DataLoader
from bird_song_dataset import BirdSongDataset, DataPaths, DeviceManager
from torch.utils.tensorboard import SummaryWriter
import os
```

[2]:
```
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):
```

```python
        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]:
```python
# 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]: # Data split sizes for train, val, and test
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]: # Random split
train_dataset, val_dataset, test_dataset = torch.utils.data.
 ↪random_split(bird_dataset, [train_size, val_size, test_size])

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

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

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

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

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

# Initialize the best validation loss to a high value
best_val_loss = float('inf')

# Set number of epochs for training
num_epochs = 10
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
            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 if this is the best model so far
    if validation_loss < best_val_loss:
        best_val_loss = validation_loss
        # Save the model
```

```
        torch.save(model.state_dict(), f"{paths['models_dir']}/model_fixed_lr/
↪model_best.pth")

    print(f"Epoch {epoch+1}/{num_epochs}, Training Loss: {training_loss},␣
↪Validation Loss: {validation_loss}")
    print("-" * 75)

# Closing TensorBoard writer
writer.close()
```

```
Epoch 1/10, Training Loss: 29.40092813372612, Validation Loss:
1.0550424869243915
---------------------------------------------------------------------------
Epoch 2/10, Training Loss: 0.8446318248907725, Validation Loss:
0.8775509870969332
---------------------------------------------------------------------------
Epoch 3/10, Training Loss: 0.5641969790061315, Validation Loss:
0.8226675391197205
---------------------------------------------------------------------------
Epoch 4/10, Training Loss: 0.3868511237204075, Validation Loss:
0.8405184975037208
---------------------------------------------------------------------------
Epoch 5/10, Training Loss: 0.2856514650086562, Validation Loss:
0.8824616991556608
---------------------------------------------------------------------------
Epoch 6/10, Training Loss: 0.22764844683309396, Validation Loss:
0.8985623854857224
---------------------------------------------------------------------------
Epoch 7/10, Training Loss: 0.14086748684446018, Validation Loss:
0.9348215598326463
---------------------------------------------------------------------------
Epoch 8/10, Training Loss: 0.07668051312988003, Validation Loss:
1.011623501777649
---------------------------------------------------------------------------
Epoch 9/10, Training Loss: 0.06212049775446455, Validation Loss:
1.0507894937808697
---------------------------------------------------------------------------
Epoch 10/10, Training Loss: 0.03388605825603008, Validation Loss:
1.2411040709568903
---------------------------------------------------------------------------
```