lab8q2

March 11, 2025

[2]:

```
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader, random_split
from torch.nn.utils.rnn import pad_sequence, pack_padded_sequence,␣
 ↪pad_packed_sequence

from tqdm.notebook import tqdm

import glob
```

[25]:

```
class NamesDataset(Dataset):
    def __init__(self, device='cpu'):
        self.inputs = []
        self.targets = []
        self.device = device
        self.ttoi = {}  # Target-to-index mapping
        len_targets = 0
        self.input_size = 128  # ASCII character range

        file_list = glob.glob('/home/student/Desktop/220962049_aiml/names/names/
 ↪*')
        if not file_list:
            raise ValueError("No files found. Check dataset path.")

        for filename in file_list:
            target = filename.split('/')[-1].split('.')[0]

            if target not in self.ttoi:
                self.ttoi[target] = len_targets
                len_targets += 1

            with open(filename, 'r') as f:
                names = f.read().strip().lower()

            if not names:
                print(f"Warning: {filename} is empty.")
                continue
            for name in names.split('\n'):
                name = name.strip()
                if not name: continue

                # Convert name to one-hot encoding
                name_tensor = self.name_to_onehot(name)

                self.inputs.append(name_tensor)
                self.targets.append(torch.tensor(self.ttoi[target]))

        if not self.inputs:
            raise ValueError("No valid data found. Check dataset files.")

    def name_to_onehot(self, name):
        """Convert a name to a one-hot encoded tensor (sequence_length, 128)"""
        name_tensor = torch.zeros(len(name), self.input_size, dtype=torch.
 ↪float32)
        for i, char in enumerate(name):
            ascii_val = ord(char)
            if ascii_val < 128:
                name_tensor[i][ascii_val] = 1.0  # One-hot encoding
        return name_tensor

    def __len__(self):
        return len(self.targets)

    def __getitem__(self, idx):
        return self.inputs[idx].to(self.device), self.targets[idx].to(self.
 ↪device)
```

[26]:
```
dataset = NamesDataset(device="cpu")
print(f"Dataset size: {len(dataset)}")
print("Sample input:", dataset[0])
print("Languages mapping:", dataset.ttoi)
```

```
Dataset size: 20074
Sample input: (tensor([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0.]]), tensor(0))
Languages mapping: {'French': 0, 'Russian': 1, 'Scottish': 2, 'Chinese': 3,
'Czech': 4, 'Arabic': 5, 'Korean': 6, 'Japanese': 7, 'Vietnamese': 8, 'Dutch':
9, 'Greek': 10, 'Spanish': 11, 'Irish': 12, 'German': 13, 'Italian': 14,
'Polish': 15, 'Portuguese': 16, 'English': 17}
```

[29]:

```python
from torch.utils.data import DataLoader
from torch.nn.utils.rnn import pad_sequence

# Custom collate function for padding
def custom_collate_fn(batch):
    inputs, targets = zip(*batch)
    inputs_padded = pad_sequence(inputs, batch_first=True, padding_value=0)  #␣
 ↪Pad to same length
    targets = torch.tensor(targets, dtype=torch.int64)
    return inputs_padded, targets

# Create DataLoaders
train_size = int(0.8 * len(dataset))
test_size = len(dataset) - train_size
train_dataset, test_dataset = torch.utils.data.random_split(dataset,␣
 ↪[train_size, test_size])

train_loader = DataLoader(train_dataset, batch_size=256, shuffle=True,␣
 ↪collate_fn=custom_collate_fn)
```

```
test_loader = DataLoader(test_dataset, batch_size=1, shuffle=False,␣
 ↪collate_fn=custom_collate_fn)
```

[30]:

```
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        _, hidden = self.rnn(x)  # Get final hidden state
        return self.fc(hidden.squeeze(0))  # Pass through linear layer
```

[31]:

```
device = "cuda" if torch.cuda.is_available() else "cpu"

input_size = 128  # ASCII character range
hidden_size = 128
output_size = len(dataset.ttoi)

model = RNN(input_size, hidden_size, output_size).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.005)
```

[35]:

```
num_epochs = 120

for epoch in range(num_epochs):
    for batch, targets in train_loader:
        batch, targets = batch.to(device, dtype=torch.float32), targets.
 ↪to(device)

        optimizer.zero_grad()
        outputs = model(batch)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()

    if epoch % 10 == 0:
        print(f"Epoch {epoch}, Loss: {loss.item()}")
```

```
Epoch 0, Loss: 1.7537944316864014
Epoch 10, Loss: 1.7654621601104736
Epoch 20, Loss: 1.7910007238388062
Epoch 30, Loss: 1.7074795961380005
Epoch 40, Loss: 1.9959580898284912
Epoch 50, Loss: 1.6526007652282715
Epoch 60, Loss: 1.8315571546554565
Epoch 70, Loss: 2.0479135513305664
Epoch 80, Loss: 1.786002278327942
Epoch 90, Loss: 1.8503060340881348
Epoch 100, Loss: 1.9190142154693604
Epoch 110, Loss: 1.9395415782928467
```

[36]:

```
correct = 0
total = 0

with torch.no_grad():
    for batch, targets in test_loader:
        batch, targets = batch.to(device, dtype=torch.float32), targets.
 ↪to(device)
        outputs = model(batch)
        _, predicted = torch.max(outputs, 1)
        correct += (predicted == targets).sum().item()
        total += targets.size(0)

print(f"Accuracy: {correct / total:.2%}")
```

```
Accuracy: 47.27%
```