Problem_2

March 17, 2024

```
[ ]: '''
     Patrick Ballou
     ID: 801130521
     ECGR 4106
     Homework 3
     Problem 2
     '''
```

```
[ ]: '\nPatrick Ballou\nID: 801130521\nECGR 4106\nHomework 3\nProblem 2\n'
```

```
[ ]: import torch
     import torch.nn as nn
     import torch.optim as optim
     import time
     import requests
     from torch import cuda
     from torchvision import datasets, transforms
     from torch.utils.data import DataLoader, Dataset
     import numpy as np
     import pandas as pd
     from sklearn import metrics
     from sklearn.preprocessing import StandardScaler as SS
     from sklearn.model_selection import train_test_split
```

```
[ ]: #check if GPU is available and set the device accordingly
     #device = 'torch.device("cuda:0" if torch.cuda.is_available() else "cpu")'
     device = 'cuda'
     print("Using GPU: ", cuda.get_device_name())

     gpu_info = !nvidia-smi
     gpu_info = '\n'.join(gpu_info)
     if gpu_info.find('failed') >= 0:
       print('Not connected to a GPU')
     else:
       print(gpu_info)
```

```
Using GPU:  Quadro T2000
Sun Mar 17 19:44:57 2024
```

```
+-------------------------------------------------------------------------------
----------+
| NVIDIA-SMI 551.23                 Driver Version: 551.23         CUDA Version:
12.4     |
|-----------------------------------------+------------------------+------------
----------+
| GPU  Name                     TCC/WDDM  | Bus-Id          Disp.A | Volatile
Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util
Compute M. |
|                                         |                        |
MIG M. |
|=========================================+========================+============
==========|
|   0  Quadro T2000                 WDDM  |   00000000:01:00.0  On |
N/A |
| N/A   55C    P8              5W /   30W |     575MiB /   4096MiB |     16%
Default |
|                                         |                        |
N/A |
+-----------------------------------------+------------------------+------------
----------+

+-------------------------------------------------------------------------------
----------+
| Processes:
|
|  GPU   GI   CI        PID   Type   Process name
GPU Memory |
|        ID   ID
Usage      |
|===============================================================================
==========|
|    0   N/A  N/A      1788    C+G   …b3d8bbwe\Microsoft.Media.Player.exe
N/A      |
|    0   N/A  N/A      5084    C+G   …siveControlPanel\SystemSettings.exe
N/A      |
|    0   N/A  N/A      7208    C+G   …41.0_x64__8wekyb3d8bbwe\GameBar.exe
N/A      |
|    0   N/A  N/A     10352    C+G   …e Stream\88.0.0.0\GoogleDriveFS.exe
N/A      |
|    0   N/A  N/A     12968    C+G   …t.LockApp_cw5n1h2txyewy\LockApp.exe
N/A      |
|    0   N/A  N/A     14388    C+G   C:\Windows\explorer.exe
N/A      |
|    0   N/A  N/A     14988    C+G   …ekyb3d8bbwe\PhoneExperienceHost.exe
N/A      |
|    0   N/A  N/A     15592    C+G   …Programs\Microsoft VS Code\Code.exe
```

```
N/A      |
|    0   N/A  N/A     15932    C+G   …Search_cw5n1h2txyewy\SearchApp.exe
N/A      |
|    0   N/A  N/A     17808    C+G   …1300.0_x64__8j3eq9eme6ctt\IGCC.exe
N/A      |
|    0   N/A  N/A     18876    C+G   …CBS_cw5n1h2txyewy\TextInputHost.exe
N/A      |
|    0   N/A  N/A     19076    C+G   …ta\Local\Programs\Notion\Notion.exe
N/A      |
|    0   N/A  N/A     19888    C+G   …Brave-Browser\Application\brave.exe
N/A      |
|    0   N/A  N/A     20148    C+G   …aam7r\AcrobatNotificationClient.exe
N/A      |
|    0   N/A  N/A     21240    C+G   …les\Microsoft OneDrive\OneDrive.exe
N/A      |
|    0   N/A  N/A     21896    C+G   …5n1h2txyewy\ShellExperienceHost.exe
N/A      |
+-------------------------------------------------------------------------------
----------+
```

[ ]:

```python
# Pred dataset for training
url = "https://raw.githubusercontent.com/karpathy/char-rnn/master/data/
 ↪tinyshakespeare/input.txt"
response = requests.get(url)
text = response.text  # This is the entire text data

chars = sorted(list(set(text)))
char_to_int = {ch: i for i, ch in enumerate(chars)}
int_to_char = {i: ch for i, ch in enumerate(chars)}

# Encode the text into integers
encoded_text = [char_to_int[ch] for ch in text]
```

1 Problem 2A: LSTM(20, 30, 50)

[ ]:

```python
sequence_length = 20

# Create sequences and targets
sequences = []
targets = []
for i in range(0, len(encoded_text) - sequence_length):
    seq = encoded_text[i:i+sequence_length]
    target = encoded_text[i+sequence_length]
    sequences.append(seq)
    targets.append(target)

# Convert lists to PyTorch tensors
```

```
sequences = torch.tensor(sequences, dtype=torch.long)
targets = torch.tensor(targets, dtype=torch.long)

# Dataset class
class CharDataset(Dataset):
    def __init__(self, sequences, targets):
        self.sequences = sequences
        self.targets = targets

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

    def __getitem__(self, index):
        return self.sequences[index], self.targets[index]

# Instantiate the dataset
dataset = CharDataset(sequences, targets)

# Create data loaders
batch_size = 128
train_size = int(len(dataset) * 0.8)
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, shuffle=True, batch_size=batch_size)
test_loader = DataLoader(test_dataset, shuffle=False, batch_size=batch_size)
```

[ ]:

```
# Defining the LSTM model
class CharLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(CharLSTM, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(input_size, hidden_size)
        self.lstm = nn.LSTM(hidden_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x, hidden):
        embedded = self.embedding(x)
        output, hidden = self.lstm(embedded, hidden)
        output = self.fc(output[:, -1, :])
        return output, hidden

    def init_hidden(self, batch_size):
        return (torch.zeros(1, batch_size, self.hidden_size, device=device),
                torch.zeros(1, batch_size, self.hidden_size, device=device))
```

```
# Hyperparameters
input_size = len(chars)
hidden_size = 256
output_size = len(chars)
learning_rate = 0.001

# Model, loss, and optimizer
model = CharLSTM(input_size, hidden_size, output_size).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

epochs = 25

init_time = time.time()
print("20 sequence LSTM results:")

# Training the model
for epoch in range(epochs):
    model.train()
    running_loss = 0.0
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        hidden = model.init_hidden(data.size(0))
        output, hidden = model(data, hidden)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    print(f"Epoch {epoch+1}, Training Loss: {running_loss / len(train_loader)}")

print(f"Training time: {(time.time() - init_time)/60} minutes")

# Validation
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for data, target in test_loader:
        data, target = data.to(device), target.to(device)
        hidden = model.init_hidden(data.size(0))
        output, hidden = model(data, hidden)
        _, predicted = torch.max(output.data, 1)
        total += target.size(0)
        correct += (predicted == target).sum().item()
```

```
print(f"Accuracy of test set: {100 * correct / total}%")

#torch.save(model.state_dict(), '../../Models/hw3_2a_20.pth')
```

```
20 sequence LSTM results:
Epoch 1, Training Loss: 1.7039286458348653
Epoch 2, Training Loss: 1.4828566941152137
Epoch 3, Training Loss: 1.4228907150060912
Epoch 4, Training Loss: 1.3878118183188102
Epoch 5, Training Loss: 1.361920929226941
Epoch 6, Training Loss: 1.3428072792254024
Epoch 7, Training Loss: 1.3274340351217448
Epoch 8, Training Loss: 1.3143731060636092
Epoch 9, Training Loss: 1.303149068528253
Epoch 10, Training Loss: 1.2939258513953997
Epoch 11, Training Loss: 1.2853256974986964
Epoch 12, Training Loss: 1.2785495665657471
Epoch 13, Training Loss: 1.2721631698955835
Epoch 14, Training Loss: 1.2667568558447675
Epoch 15, Training Loss: 1.2605822635873323
Epoch 16, Training Loss: 1.2563257013852065
Epoch 17, Training Loss: 1.2524168876417054
Epoch 18, Training Loss: 1.2479410454827482
Epoch 19, Training Loss: 1.2448831394066842
Epoch 20, Training Loss: 1.2421532968594573
Epoch 21, Training Loss: 1.2385113516280655
Epoch 22, Training Loss: 1.2367871534068626
Epoch 23, Training Loss: 1.2343798212224015
Epoch 24, Training Loss: 1.232003951672843
Epoch 25, Training Loss: 1.2305857209350894
Training time: 10.025756839911143 minutes
Accuracy of test set: 58.42608988008517%
```

[ ]:

```
sequence_length = 30

# Create sequences and targets
sequences = []
targets = []
for i in range(0, len(encoded_text) - sequence_length):
    seq = encoded_text[i:i+sequence_length]
    target = encoded_text[i+sequence_length]
    sequences.append(seq)
    targets.append(target)

# Convert lists to PyTorch tensors
sequences = torch.tensor(sequences, dtype=torch.long)
targets = torch.tensor(targets, dtype=torch.long)
```

```
# Dataset class
class CharDataset(Dataset):
    def __init__(self, sequences, targets):
        self.sequences = sequences
        self.targets = targets

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

    def __getitem__(self, index):
        return self.sequences[index], self.targets[index]

# Instantiate the dataset
dataset = CharDataset(sequences, targets)

# Create data loaders
batch_size = 128
train_size = int(len(dataset) * 0.8)
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, shuffle=True, batch_size=batch_size)
test_loader = DataLoader(test_dataset, shuffle=False, batch_size=batch_size)
```

[ ]:

```
# Defining the LSTM model
class CharLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(CharLSTM, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(input_size, hidden_size)
        self.lstm = nn.LSTM(hidden_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x, hidden):
        embedded = self.embedding(x)
        output, hidden = self.lstm(embedded, hidden)
        output = self.fc(output[:, -1, :])
        return output, hidden

    def init_hidden(self, batch_size):
        return (torch.zeros(1, batch_size, self.hidden_size, device=device),
                torch.zeros(1, batch_size, self.hidden_size, device=device))

# Hyperparameters
input_size = len(chars)
```

```
hidden_size = 256
output_size = len(chars)
learning_rate = 0.001

# Model, loss, and optimizer
model = CharLSTM(input_size, hidden_size, output_size).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

epochs = 25

init_time = time.time()
print("30 sequence LSTM results:")

# Training the model
for epoch in range(epochs):
    model.train()
    running_loss = 0.0
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        hidden = model.init_hidden(data.size(0))
        output, hidden = model(data, hidden)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    print(f"Epoch {epoch+1}, Training Loss: {running_loss / len(train_loader)}")

print(f"Training time: {(time.time() - init_time)/60} minutes")

# Validation
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for data, target in test_loader:
        data, target = data.to(device), target.to(device)
        hidden = model.init_hidden(data.size(0))
        output, hidden = model(data, hidden)
        _, predicted = torch.max(output.data, 1)
        total += target.size(0)
        correct += (predicted == target).sum().item()

print(f"Accuracy of test set: {100 * correct / total}%")
```

```
#torch.save(model.state_dict(), '../../Models/hw3_2a_30.pth')
```

```
30 sequence LSTM results:
Epoch 1, Training Loss: 1.6993088538251386
Epoch 2, Training Loss: 1.474142562501812
Epoch 3, Training Loss: 1.4137238254679803
Epoch 4, Training Loss: 1.3783982019963377
Epoch 5, Training Loss: 1.3537912187788996
Epoch 6, Training Loss: 1.3340486319531535
Epoch 7, Training Loss: 1.3190894767094876
Epoch 8, Training Loss: 1.3062398333403822
Epoch 9, Training Loss: 1.2956803129516794
Epoch 10, Training Loss: 1.2862978480538279
Epoch 11, Training Loss: 1.278085498514356
Epoch 12, Training Loss: 1.270350139226916
Epoch 13, Training Loss: 1.263484604259161
Epoch 14, Training Loss: 1.2579474826260537
Epoch 15, Training Loss: 1.252641338822969
Epoch 16, Training Loss: 1.2477584728480617
Epoch 17, Training Loss: 1.2438362084085477
Epoch 18, Training Loss: 1.2396418672574367
Epoch 19, Training Loss: 1.235707599415358
Epoch 20, Training Loss: 1.2322622239076741
Epoch 21, Training Loss: 1.2298383069188856
Epoch 22, Training Loss: 1.2269100274525184
Epoch 23, Training Loss: 1.225325421208427
Epoch 24, Training Loss: 1.2233063099255044
Epoch 25, Training Loss: 1.2213229424075975
Training time: 11.263746031125386 minutes
Accuracy of test set: 58.58575443912979%
```

[ ]:

```
sequence_length = 50

# Create sequences and targets
sequences = []
targets = []
for i in range(0, len(encoded_text) - sequence_length):
    seq = encoded_text[i:i+sequence_length]
    target = encoded_text[i+sequence_length]
    sequences.append(seq)
    targets.append(target)

# Convert lists to PyTorch tensors
sequences = torch.tensor(sequences, dtype=torch.long)
targets = torch.tensor(targets, dtype=torch.long)

# Dataset class
```

```
class CharDataset(Dataset):
    def __init__(self, sequences, targets):
        self.sequences = sequences
        self.targets = targets

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

    def __getitem__(self, index):
        return self.sequences[index], self.targets[index]

# Instantiate the dataset
dataset = CharDataset(sequences, targets)

# Create data loaders
batch_size = 128
train_size = int(len(dataset) * 0.8)
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, shuffle=True, batch_size=batch_size)
test_loader = DataLoader(test_dataset, shuffle=False, batch_size=batch_size)
```

[ ]:

```
# Defining the LSTM model
class CharLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(CharLSTM, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(input_size, hidden_size)
        self.lstm = nn.LSTM(hidden_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x, hidden):
        embedded = self.embedding(x)
        output, hidden = self.lstm(embedded, hidden)
        output = self.fc(output[:, -1, :])
        return output, hidden

    def init_hidden(self, batch_size):
        return (torch.zeros(1, batch_size, self.hidden_size, device=device),
                torch.zeros(1, batch_size, self.hidden_size, device=device))

# Hyperparameters
input_size = len(chars)
hidden_size = 256
output_size = len(chars)
```

```python
learning_rate = 0.001

# Model, loss, and optimizer
model = CharLSTM(input_size, hidden_size, output_size).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

epochs = 25

init_time = time.time()
print("50 sequence LSTM results:")

# Training the model
for epoch in range(epochs):
    model.train()
    running_loss = 0.0
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        hidden = model.init_hidden(data.size(0))
        output, hidden = model(data, hidden)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    print(f"Epoch {epoch+1}, Training Loss: {running_loss / len(train_loader)}")

print(f"Training time: {(time.time() - init_time)/60} minutes")

# Validation
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for data, target in test_loader:
        data, target = data.to(device), target.to(device)
        hidden = model.init_hidden(data.size(0))
        output, hidden = model(data, hidden)
        _, predicted = torch.max(output.data, 1)
        total += target.size(0)
        correct += (predicted == target).sum().item()

print(f"Accuracy of test set: {100 * correct / total}%")

#torch.save(model.state_dict(), '../../Models/hw3_2a_50.pth')
```

```
50 sequence LSTM results:
```

```
Epoch 1, Training Loss: 1.7033157834293955
Epoch 2, Training Loss: 1.472816352770021
Epoch 3, Training Loss: 1.4091637182092072
Epoch 4, Training Loss: 1.3726160892830093
Epoch 5, Training Loss: 1.346235753165959
Epoch 6, Training Loss: 1.325980530674737
Epoch 7, Training Loss: 1.3088446769271789
Epoch 8, Training Loss: 1.2956975393650298
Epoch 9, Training Loss: 1.284235503269634
Epoch 10, Training Loss: 1.2747833096490258
Epoch 11, Training Loss: 1.2659264016383998
Epoch 12, Training Loss: 1.2586697097950559
Epoch 13, Training Loss: 1.2507322769119893
Epoch 14, Training Loss: 1.245463993572946
Epoch 15, Training Loss: 1.240341373383272
Epoch 16, Training Loss: 1.234599030893411
Epoch 17, Training Loss: 1.2303823452566331
Epoch 18, Training Loss: 1.227314035949945
Epoch 19, Training Loss: 1.2234475778517602
Epoch 20, Training Loss: 1.220022144413049
Epoch 21, Training Loss: 1.217116185521279
Epoch 22, Training Loss: 1.21419994343885
Epoch 23, Training Loss: 1.212461400247513
Epoch 24, Training Loss: 1.2099621501052384
Epoch 25, Training Loss: 1.2077432776297752
Training time: 12.90316569407781 minutes
Accuracy of test set: 59.213965185660044%
```

2 Problem 2B: GRU(20, 30, 50)

[ ]:

```python
sequence_length = 20

# Create sequences and targets
sequences = []
targets = []
for i in range(0, len(encoded_text) - sequence_length):
    seq = encoded_text[i:i+sequence_length]
    target = encoded_text[i+sequence_length]
    sequences.append(seq)
    targets.append(target)

# Convert lists to PyTorch tensors
sequences = torch.tensor(sequences, dtype=torch.long)
targets = torch.tensor(targets, dtype=torch.long)

# Dataset class
class CharDataset(Dataset):
    def __init__(self, sequences, targets):
        self.sequences = sequences
        self.targets = targets

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

    def __getitem__(self, index):
        return self.sequences[index], self.targets[index]

# Instantiate the dataset
dataset = CharDataset(sequences, targets)

# Create data loaders
batch_size = 128
train_size = int(len(dataset) * 0.8)
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, shuffle=True, batch_size=batch_size)
test_loader = DataLoader(test_dataset, shuffle=False, batch_size=batch_size)
```

[ ]:

```
# Defining the GRU model
class CharGRU(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(CharGRU, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(input_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x, hidden):
        embedded = self.embedding(x)
        output, hidden = self.gru(embedded, hidden)
        output = self.fc(output[:, -1, :])
        return output, hidden

    def init_hidden(self, batch_size):
        return (torch.zeros(1, batch_size, self.hidden_size, device=device))

# Hyperparameters
input_size = len(chars)
hidden_size = 256
output_size = len(chars)
learning_rate = 0.001
```

```
# Model, loss, and optimizer
model = CharGRU(input_size, hidden_size, output_size).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

epochs = 25

init_time = time.time()
print("20 sequence GRU results:")

# Training the model
for epoch in range(epochs):
    model.train()
    running_loss = 0.0
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        hidden = model.init_hidden(data.size(0))
        output, hidden = model(data, hidden)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    print(f"Epoch {epoch+1}, Training Loss: {running_loss / len(train_loader)}")

print(f"Training time: {(time.time() - init_time)/60} minutes")

# Validation
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for data, target in test_loader:
        data, target = data.to(device), target.to(device)
        hidden = model.init_hidden(data.size(0))
        output, hidden = model(data, hidden)
        _, predicted = torch.max(output.data, 1)
        total += target.size(0)
        correct += (predicted == target).sum().item()

print(f"Accuracy of test set: {100 * correct / total}%")

#torch.save(model.state_dict(), '../../Models/hw3_2b_20.pth')
```

```
20 sequence GRU results:
Epoch 1, Training Loss: 1.6904839627782435
Epoch 2, Training Loss: 1.497342645540637
Epoch 3, Training Loss: 1.4507262527993952
Epoch 4, Training Loss: 1.4255671261270635
Epoch 5, Training Loss: 1.4101337558807345
Epoch 6, Training Loss: 1.3995701468373878
Epoch 7, Training Loss: 1.3912026316047195
Epoch 8, Training Loss: 1.3853752609391494
Epoch 9, Training Loss: 1.3809091519399819
Epoch 10, Training Loss: 1.376863594574622
Epoch 11, Training Loss: 1.374466078185994
Epoch 12, Training Loss: 1.3727443141288889
Epoch 13, Training Loss: 1.3720990438794378
Epoch 14, Training Loss: 1.3719647168873783
Epoch 15, Training Loss: 1.3736919506110477
Epoch 16, Training Loss: 1.3745446196675506
Epoch 17, Training Loss: 1.3749985953411423
Epoch 18, Training Loss: 1.3745096339723946
Epoch 19, Training Loss: 1.3774950651348288
Epoch 20, Training Loss: 1.3789241221090602
Epoch 21, Training Loss: 1.3827220115573393
Epoch 22, Training Loss: 1.3876126805342823
Epoch 23, Training Loss: 1.3884858456720788
Epoch 24, Training Loss: 1.3935831162396752
Epoch 25, Training Loss: 1.3947730505100557
Training time: 9.919338961442312 minutes
Accuracy of test set: 55.9645859016026%
```

[ ]:

```python
sequence_length = 30

# Create sequences and targets
sequences = []
targets = []
for i in range(0, len(encoded_text) - sequence_length):
    seq = encoded_text[i:i+sequence_length]
    target = encoded_text[i+sequence_length]
    sequences.append(seq)
    targets.append(target)

# Convert lists to PyTorch tensors
sequences = torch.tensor(sequences, dtype=torch.long)
targets = torch.tensor(targets, dtype=torch.long)

# Dataset class
class CharDataset(Dataset):
    def __init__(self, sequences, targets):
        self.sequences = sequences
        self.targets = targets
    def __len__(self):
        return len(self.sequences)

    def __getitem__(self, index):
        return self.sequences[index], self.targets[index]

# Instantiate the dataset
dataset = CharDataset(sequences, targets)

# Create data loaders
batch_size = 128
train_size = int(len(dataset) * 0.8)
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, shuffle=True, batch_size=batch_size)
test_loader = DataLoader(test_dataset, shuffle=False, batch_size=batch_size)
```

[ ]:

```
# Defining the GRU model
class CharGRU(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(CharGRU, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(input_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x, hidden):
        embedded = self.embedding(x)
        output, hidden = self.gru(embedded, hidden)
        output = self.fc(output[:, -1, :])
        return output, hidden

    def init_hidden(self, batch_size):
        return (torch.zeros(1, batch_size, self.hidden_size, device=device))

# Hyperparameters
input_size = len(chars)
hidden_size = 256
output_size = len(chars)
learning_rate = 0.001

# Model, loss, and optimizer
model = CharGRU(input_size, hidden_size, output_size).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

```python
epochs = 25

init_time = time.time()
print("30 sequence GRU results:")

# Training the model
for epoch in range(epochs):
    model.train()
    running_loss = 0.0
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        hidden = model.init_hidden(data.size(0))
        output, hidden = model(data, hidden)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    print(f"Epoch {epoch+1}, Training Loss: {running_loss / len(train_loader)}")

print(f"Training time: {(time.time() - init_time)/60} minutes")

# Validation
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for data, target in test_loader:
        data, target = data.to(device), target.to(device)
        hidden = model.init_hidden(data.size(0))
        output, hidden = model(data, hidden)
        _, predicted = torch.max(output.data, 1)
        total += target.size(0)
        correct += (predicted == target).sum().item()

print(f"Accuracy of test set: {100 * correct / total}%")

#torch.save(model.state_dict(), '../../Models/hw3_2b_30.pth')
```

```
30 sequence GRU results:
Epoch 1, Training Loss: 1.688761063437317
Epoch 2, Training Loss: 1.4947071062188277
Epoch 3, Training Loss: 1.446230951023389
Epoch 4, Training Loss: 1.4202568271740783
Epoch 5, Training Loss: 1.403517935940129
Epoch 6, Training Loss: 1.3898934693649043
Epoch 7, Training Loss: 1.3804380048538718
Epoch 8, Training Loss: 1.3759228813104498
Epoch 9, Training Loss: 1.369669947578725
Epoch 10, Training Loss: 1.367014287098413
Epoch 11, Training Loss: 1.3641738207156946
Epoch 12, Training Loss: 1.3612514548067791
Epoch 13, Training Loss: 1.3606549097300944
Epoch 14, Training Loss: 1.3582993330229853
Epoch 15, Training Loss: 1.3583210253093136
Epoch 16, Training Loss: 1.3583726284665345
Epoch 17, Training Loss: 1.3611586110060474
Epoch 18, Training Loss: 1.3605015400115335
Epoch 19, Training Loss: 1.3630748972835312
Epoch 20, Training Loss: 1.3640652118369347
Epoch 21, Training Loss: 1.3667174100756165
Epoch 22, Training Loss: 1.3682563400993522
Epoch 23, Training Loss: 1.3699411446174123
Epoch 24, Training Loss: 1.373385976365224
Epoch 25, Training Loss: 1.3744303944445313
Training time: 11.177410554885864 minutes
Accuracy of test set: 56.34747369695122%
```

[ ]:

```
sequence_length = 50

# Create sequences and targets
sequences = []
targets = []
for i in range(0, len(encoded_text) - sequence_length):
    seq = encoded_text[i:i+sequence_length]
    target = encoded_text[i+sequence_length]
    sequences.append(seq)
    targets.append(target)

# Convert lists to PyTorch tensors
sequences = torch.tensor(sequences, dtype=torch.long)
targets = torch.tensor(targets, dtype=torch.long)

# Dataset class
class CharDataset(Dataset):
    def __init__(self, sequences, targets):
        self.sequences = sequences
        self.targets = targets

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

    def __getitem__(self, index):
        return self.sequences[index], self.targets[index]

# Instantiate the dataset
dataset = CharDataset(sequences, targets)

# Create data loaders
batch_size = 128
train_size = int(len(dataset) * 0.8)
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, shuffle=True, batch_size=batch_size)
test_loader = DataLoader(test_dataset, shuffle=False, batch_size=batch_size)
```

[ ]:

```
# Defining the GRU model
class CharGRU(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(CharGRU, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(input_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x, hidden):
        embedded = self.embedding(x)
        output, hidden = self.gru(embedded, hidden)
        output = self.fc(output[:, -1, :])
        return output, hidden

    def init_hidden(self, batch_size):
        return (torch.zeros(1, batch_size, self.hidden_size, device=device))

# Hyperparameters
input_size = len(chars)
hidden_size = 256
output_size = len(chars)
learning_rate = 0.001

# Model, loss, and optimizer
model = CharGRU(input_size, hidden_size, output_size).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

epochs = 25

init_time = time.time()
```

```python
print("50 sequence GRU results:")

# Training the model
for epoch in range(epochs):
    model.train()
    running_loss = 0.0
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        hidden = model.init_hidden(data.size(0))
        output, hidden = model(data, hidden)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    print(f"Epoch {epoch+1}, Training Loss: {running_loss / len(train_loader)}")

print(f"Training time: {(time.time() - init_time)/60} minutes")

# Validation
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for data, target in test_loader:
        data, target = data.to(device), target.to(device)
        hidden = model.init_hidden(data.size(0))
        output, hidden = model(data, hidden)
        _, predicted = torch.max(output.data, 1)
        total += target.size(0)
        correct += (predicted == target).sum().item()

print(f"Accuracy of test set: {100 * correct / total}%")

#torch.save(model.state_dict(), '../../Models/hw3_2b_50.pth')
```

```
50 sequence GRU results:
Epoch 1, Training Loss: 1.6804511425160316
Epoch 2, Training Loss: 1.4817302748678474
Epoch 3, Training Loss: 1.435106966865232
Epoch 4, Training Loss: 1.4087642558830709
Epoch 5, Training Loss: 1.3914834176548379
Epoch 6, Training Loss: 1.3802862838049597
Epoch 7, Training Loss: 1.3710225728881391
Epoch 8, Training Loss: 1.3649185802036712
Epoch 9, Training Loss: 1.3613395527092838
Epoch 10, Training Loss: 1.356445742951389
Epoch 11, Training Loss: 1.353771461371352
Epoch 12, Training Loss: 1.353007037043486
Epoch 13, Training Loss: 1.3511901474925885
Epoch 14, Training Loss: 1.3503325411740206
Epoch 15, Training Loss: 1.349965271555083
Epoch 16, Training Loss: 1.3516816538403869
Epoch 17, Training Loss: 1.351813011686902
Epoch 18, Training Loss: 1.3515537315419819
Epoch 19, Training Loss: 1.3527359983668301
Epoch 20, Training Loss: 1.3570245269842562
Epoch 21, Training Loss: 1.3570453758784233
Epoch 22, Training Loss: 1.3602206598924027
Epoch 23, Training Loss: 1.3637807675848526
Epoch 24, Training Loss: 1.365017493560177
Epoch 25, Training Loss: 1.368766647631153
Training time: 12.810324263572692 minutes
Accuracy of test set: 56.79632759370419%
```