Problem_1

March 17, 2024

[ ]:
```
'''
Patrick Ballou
ID: 801130521
ECGR 4106
Homework 3
Problem 1
'''
```

[ ]: '\nPatrick Ballou\nID: 801130521\nECGR 4106\nHomework 3\nProblem 1\n'

[ ]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import time
from torch import cuda
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, TensorDataset
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
Thu Mar 14 18:12:53 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   44C    P8              6W /   60W |    1230MiB /   4096MiB |     33%
Default |
|                                         |                        |
N/A |
+-----------------------------------------+------------------------+------------
----------+

+-------------------------------------------------------------------------------
----------+
| Processes:
|
|  GPU   GI   CI        PID   Type   Process name
GPU Memory |
|        ID   ID
Usage      |
|===============================================================================
==========|
|    0   N/A  N/A      1168    C+G   …ta\Local\Programs\Notion\Notion.exe
N/A      |
|    0   N/A  N/A      2832    C+G   …siveControlPanel\SystemSettings.exe
N/A      |
|    0   N/A  N/A      6664    C+G   …Programs\Microsoft VS Code\Code.exe
N/A      |
|    0   N/A  N/A      7804    C+G   …b3d8bbwe\Microsoft.Media.Player.exe
N/A      |
|    0   N/A  N/A     12632    C+G   …Search_cw5n1h2txyewy\SearchApp.exe
N/A      |
|    0   N/A  N/A     14636    C+G   …aam7r\AcrobatNotificationClient.exe
N/A      |
|    0   N/A  N/A     14748    C+G   …AppData\Roaming\Spotify\Spotify.exe
N/A      |
|    0   N/A  N/A     14840    C+G   C:\Windows\explorer.exe
N/A      |
```

```
|    0   N/A  N/A     15140    C+G   …CBS_cw5n1h2txyewy\TextInputHost.exe
N/A      |
|    0   N/A  N/A     16296    C+G   …t.LockApp_cw5n1h2txyewy\LockApp.exe
N/A      |
|    0   N/A  N/A     17264    C+G   …1300.0_x64__8j3eq9eme6ctt\IGCC.exe
N/A      |
|    0   N/A  N/A     17840    C+G   …5n1h2txyewy\ShellExperienceHost.exe
N/A      |
|    0   N/A  N/A     19340    C+G   …ekyb3d8bbwe\PhoneExperienceHost.exe
N/A      |
|    0   N/A  N/A     19600    C+G   …les\Microsoft OneDrive\OneDrive.exe
N/A      |
|    0   N/A  N/A     20568    C+G   …Brave-Browser\Application\brave.exe
N/A      |
|    0   N/A  N/A     21900    C+G   …e Stream\87.0.2.0\GoogleDriveFS.exe
N/A      |
+--------------------------------------------------------------------------------
----------+
```

[ ]:
```
text = """Next character prediction is a fundamental task in the field of natural language processing (NLP) that involves predicting the next character in a sequence of text based on the characters that precede it.
       This task is essential for various applications, including text auto-completion, spell checking, and even in the development of sophisticated AI models capable of generating human-like text.
       At its core, next character prediction relies on statistical models or deep learning algorithms to analyze a given sequence of text and predict which character is most likely to follow.
       These predictions are based on patterns and relationships learned from large datasets of text during the training phase of the model.
       One of the most popular approaches to next character prediction involves the use of Recurrent Neural Networks (RNNs), and more specifically, a variant called Long Short-Term Memory (LSTM) networks.
       RNNs are particularly well-suited for sequential data like text, as they can maintain information in 'memory' about previous characters to inform the prediction of the next character.
       LSTM networks enhance this capability by being able to remember long-term dependencies, making them even more effective for next character prediction tasks.
       Training a model for next character prediction involves feeding it large amounts of text data, allowing it to learn the probability of each character's appearance following a sequence of characters.
       During this training process, the model adjusts its parameters to minimize the difference between its predictions and the actual outcomes, thus improving its predictive accuracy over time.
```

```
        Once trained, the model can be used to predict the next character in a␣
↪given piece of text by considering the sequence of characters that precede␣
↪it.
        This can enhance user experience in text editing software, improve␣
↪efficiency in coding environments with auto-completion features, and enable␣
↪more natural interactions with AI-based chatbots and virtual assistants.
        In summary, next character prediction plays a crucial role in enhancing␣
↪the capabilities of various NLP applications, making text-based interactions␣
↪more efficient, accurate, and human-like.
        Through the use of advanced machine learning models like RNNs and␣
↪LSTMs, next character prediction continues to evolve, opening new␣
↪possibilities for the future of text-based technology."""
```

[ ]:
```
# Creating character vocabulary
# part of the data preprocessing step for a character-level text modeling task.
# Create mappings between characters in the text and numerical indices

#set(text): Creates a set of unique characters found in the text. The set␣
↪function removes any duplicate characters.
#list(set(text)): Converts the set back into a list so that it can be sorted.
# sorted(list(set(text))): Sorts the list of unique characters.
chars = sorted(list(set(text)))
#This line creates a dictionary that maps each character to a unique index␣
↪(integer)."
ix_to_char = {i: ch for i, ch in enumerate(chars)}
#Similar to the previous line, but in reverse. This line creates a dictionary␣
↪that maps each unique index (integer) back to its corresponding character.
char_to_ix = {ch: i for i, ch in enumerate(chars)}
chars = sorted(list(set(text)))
```

1 Problem 1A: RNN(10, 20, 30)

[ ]:
```
# Preparing the dataset
max_length = 10  # Maximum length of input sequences
X = []
y = []
for i in range(len(text) - max_length):
    sequence = text[i:i + max_length]
    label = text[i + max_length]
    X.append([char_to_ix[char] for char in sequence])
    y.append(char_to_ix[label])

X = np.array(X)
y = np.array(y)

# Splitting the dataset into training and validation sets
```

```
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)

# Converting data to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.long)
y_train = torch.tensor(y_train, dtype=torch.long)
X_val = torch.tensor(X_val, dtype=torch.long)
y_val = torch.tensor(y_val, dtype=torch.long)

# Defining the RNN model
class CharRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(CharRNN, self).__init__()
        self.hidden_size = hidden_size
        #This line takes the input tensor x, which contains indices of␣
↪characters, and passes it through an embedding layer (self.embedding).
        #The embedding layer converts these indices into dense vectors of fixed␣
↪size.
        #These vectors are learned during training and can capture semantic␣
↪similarities between characters.
        #The result is a higher-dimensional representation of the input␣
↪sequence, where each character index is replaced by its corresponding␣
↪embedding vector.
        self.embedding = nn.Embedding(input_size, hidden_size)
        self.rnn = nn.RNN(hidden_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        embedded = self.embedding(x)
        #The RNN layer returns two outputs:
        #1- the output tensor containing the output of the RNN at each time␣
↪step for each sequence in the batch,
        #2-the hidden state (_) of the last time step (which is not used in␣
↪this line, hence the underscore).
        output, _ = self.rnn(embedded)
        #The RNN's output contains the outputs for every time step,
        #but for this task, we're only interested in the output of the last␣
↪time step because we're predicting the next character after the sequence.
        #output[:, -1, :] selects the last time step's output for every␣
↪sequence in the batch (-1 indexes the last item in Python).
        output = self.fc(output[:, -1, :])  # Get the output of the last RNN␣
↪cell
        return output
```

[ ]:
```
# Hyperparameters
hidden_size = 128
```

```python
learning_rate = 0.005
epochs = 100

# Model, loss, and optimizer
model = CharRNN(len(chars), hidden_size, len(chars))
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

init_time = time.time()
print("10 sequence RNN results:")

# Training the model
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()
    output = model(X_train)
    loss = criterion(output, y_train)
    loss.backward()
    optimizer.step()

    # Validation
    model.eval()
    with torch.no_grad():
        val_output = model(X_val)
        val_loss = criterion(val_output, y_val)
        #The use of the underscore _ is a common Python convention to indicate that the actual maximum values returned by torch.max are not needed and can be disregarded.
        #What we are interested in is the indices of these maximum values, which are captured by the variable predicted. These indices represent the model's predictions for each example in the validation set.
        _, predicted = torch.max(val_output, 1)
        val_accuracy = (predicted == y_val).float().mean()

    if (epoch+1) % 20 == 0:
        print(f'Epoch {epoch+1}, Loss: {loss.item()}, Validation Loss: {val_loss.item()}, Validation Accuracy: {val_accuracy.item()}')

print(f"Training time: {time.time() - init_time} seconds")
torch.save(model.state_dict(), '../../Models/hw3_1a_10.pth')
```

```
10 sequence RNN results:
Epoch 20, Loss: 1.7982101440429688, Validation Loss: 2.1622025966644287,
Validation Accuracy: 0.42052313685417175
Epoch 40, Loss: 1.1528027057647705, Validation Loss: 2.0086846351623535,
Validation Accuracy: 0.4828973710536957
Epoch 60, Loss: 0.6619546413421631, Validation Loss: 2.0594570636749268,
```

```
Validation Accuracy: 0.4909456670284271
Epoch 80, Loss: 0.31352028250694275, Validation Loss: 2.291659355163574,
Validation Accuracy: 0.49295774102211
Epoch 100, Loss: 0.13182944059371948, Validation Loss: 2.569873094558716,
Validation Accuracy: 0.5030180811882019
Training time: 4.494206190109253 seconds
```

[ ]:
```
# Preparing the dataset
max_length = 20  # Maximum length of input sequences
X = []
y = []
for i in range(len(text) - max_length):
    sequence = text[i:i + max_length]
    label = text[i + max_length]
    X.append([char_to_ix[char] for char in sequence])
    y.append(char_to_ix[label])

X = np.array(X)
y = np.array(y)

# Splitting the dataset into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)

# Converting data to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.long)
y_train = torch.tensor(y_train, dtype=torch.long)
X_val = torch.tensor(X_val, dtype=torch.long)
y_val = torch.tensor(y_val, dtype=torch.long)

# Defining the RNN model
class CharRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(CharRNN, self).__init__()
        self.hidden_size = hidden_size
        #This line takes the input tensor x, which contains indices of␣
↪characters, and passes it through an embedding layer (self.embedding).
        #The embedding layer converts these indices into dense vectors of fixed␣
↪size.
        #These vectors are learned during training and can capture semantic␣
↪similarities between characters.
        #The result is a higher-dimensional representation of the input␣
↪sequence, where each character index is replaced by its corresponding␣
↪embedding vector.
        self.embedding = nn.Embedding(input_size, hidden_size)
        self.rnn = nn.RNN(hidden_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)
```

```
    def forward(self, x):
        embedded = self.embedding(x)
        #The RNN layer returns two outputs:
        #1- the output tensor containing the output of the RNN at each time␣
↪step for each sequence in the batch,
        #2-the hidden state (_) of the last time step (which is not used in␣
↪this line, hence the underscore).
        output, _ = self.rnn(embedded)
        #The RNN's output contains the outputs for every time step,
        #but for this task, we're only interested in the output of the last␣
↪time step because we're predicting the next character after the sequence.
        #output[:, -1, :] selects the last time step's output for every␣
↪sequence in the batch (-1 indexes the last item in Python).
        output = self.fc(output[:, -1, :])  # Get the output of the last RNN␣
↪cell
        return output
```

[ ]:
```
# Hyperparameters
hidden_size = 128
learning_rate = 0.005
epochs = 100

# Model, loss, and optimizer
model = CharRNN(len(chars), hidden_size, len(chars))
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

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

# Training the model
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()
    output = model(X_train)
    loss = criterion(output, y_train)
    loss.backward()
    optimizer.step()

    # Validation
    model.eval()
    with torch.no_grad():
        val_output = model(X_val)
        val_loss = criterion(val_output, y_val)
```

```
        #The use of the underscore _ is a common Python convention to indicate␣
↪that the actual maximum values returned by torch.max are not needed and can␣
↪be disregarded.
        #What we are interested in is the indices of these maximum values,␣
↪which are captured by the variable predicted. These indices represent the␣
↪model's predictions for each example in the validation set.
        _, predicted = torch.max(val_output, 1)
        val_accuracy = (predicted == y_val).float().mean()

    if (epoch+1) % 20 == 0:
        print(f'Epoch {epoch+1}, Loss: {loss.item()}, Validation Loss:␣
↪{val_loss.item()}, Validation Accuracy: {val_accuracy.item()}')

print(f"Training time: {time.time() - init_time} seconds")
torch.save(model.state_dict(), '../../Models/hw3_1a_20.pth')
```

```
20 sequence RNN results:
Epoch 20, Loss: 1.7767621278762817, Validation Loss: 2.131897211074829,
Validation Accuracy: 0.4161616265773773
Epoch 40, Loss: 1.115134596824646, Validation Loss: 1.9793099164962769,
Validation Accuracy: 0.49696969985961914
Epoch 60, Loss: 0.6132451891899109, Validation Loss: 2.0395753383636475,
Validation Accuracy: 0.5151515007019043
Epoch 80, Loss: 0.27225732803344727, Validation Loss: 2.242558240890503,
Validation Accuracy: 0.5191919207572937
Epoch 100, Loss: 0.12532006204128265, Validation Loss: 2.4666197299957275,
Validation Accuracy: 0.521212100982666
Training time: 8.47878646850586 seconds
```

[ ]:
```
# Preparing the dataset
max_length = 30  # Maximum length of input sequences
X = []
y = []
for i in range(len(text) - max_length):
    sequence = text[i:i + max_length]
    label = text[i + max_length]
    X.append([char_to_ix[char] for char in sequence])
    y.append(char_to_ix[label])

X = np.array(X)
y = np.array(y)

# Splitting the dataset into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)

# Converting data to PyTorch tensors
```

```
X_train = torch.tensor(X_train, dtype=torch.long)
y_train = torch.tensor(y_train, dtype=torch.long)
X_val = torch.tensor(X_val, dtype=torch.long)
y_val = torch.tensor(y_val, dtype=torch.long)

# Defining the RNN model
class CharRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(CharRNN, self).__init__()
        self.hidden_size = hidden_size
        #This line takes the input tensor x, which contains indices of␣
↪characters, and passes it through an embedding layer (self.embedding).
        #The embedding layer converts these indices into dense vectors of fixed␣
↪size.
        #These vectors are learned during training and can capture semantic␣
↪similarities between characters.
        #The result is a higher-dimensional representation of the input␣
↪sequence, where each character index is replaced by its corresponding␣
↪embedding vector.
        self.embedding = nn.Embedding(input_size, hidden_size)
        self.rnn = nn.RNN(hidden_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        embedded = self.embedding(x)
        #The RNN layer returns two outputs:
        #1- the output tensor containing the output of the RNN at each time␣
↪step for each sequence in the batch,
        #2-the hidden state (_) of the last time step (which is not used in␣
↪this line, hence the underscore).
        output, _ = self.rnn(embedded)
        #The RNN's output contains the outputs for every time step,
        #but for this task, we're only interested in the output of the last␣
↪time step because we're predicting the next character after the sequence.
        #output[:, -1, :] selects the last time step's output for every␣
↪sequence in the batch (-1 indexes the last item in Python).
        output = self.fc(output[:, -1, :])  # Get the output of the last RNN␣
↪cell
        return output
```

[ ]:
```
# Hyperparameters
hidden_size = 128
learning_rate = 0.005
epochs = 100

# Model, loss, and optimizer
```

```
model = CharRNN(len(chars), hidden_size, len(chars))
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

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

# Training the model
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()
    output = model(X_train)
    loss = criterion(output, y_train)
    loss.backward()
    optimizer.step()

    # Validation
    model.eval()
    with torch.no_grad():
        val_output = model(X_val)
        val_loss = criterion(val_output, y_val)
        #The use of the underscore _ is a common Python convention to indicate␣
↪that the actual maximum values returned by torch.max are not needed and can␣
↪be disregarded.
        #What we are interested in is the indices of these maximum values,␣
↪which are captured by the variable predicted. These indices represent the␣
↪model's predictions for each example in the validation set.
        _, predicted = torch.max(val_output, 1)
        val_accuracy = (predicted == y_val).float().mean()

    if (epoch+1) % 20 == 0:
        print(f'Epoch {epoch+1}, Loss: {loss.item()}, Validation Loss:␣
↪{val_loss.item()}, Validation Accuracy: {val_accuracy.item()}')

print(f"Training time: {time.time() - init_time} seconds")
torch.save(model.state_dict(), '../../Models/hw3_1a_30.pth')
```

```
30 sequence RNN results:
Epoch 20, Loss: 1.7936835289001465, Validation Loss: 2.113086462020874,
Validation Accuracy: 0.4482758641242981
Epoch 40, Loss: 1.1175235509872437, Validation Loss: 1.9179291725158691,
Validation Accuracy: 0.4868154227733612
Epoch 60, Loss: 0.6131270527839661, Validation Loss: 1.9983218908309937,
Validation Accuracy: 0.5091277956962585
Epoch 80, Loss: 0.30790334939956665, Validation Loss: 2.21238112449646,
Validation Accuracy: 0.5253549814224243
Epoch 100, Loss: 0.1290741115808487, Validation Loss: 2.45947265625, Validation
```

```
Accuracy: 0.529411792755127
Training time: 11.126161336898804 seconds
```

2 Problem 1B: LSTM(10, 20, 30)

```
[ ]: # Preparing the dataset
max_length = 10  # Maximum length of input sequences
X = []
y = []
for i in range(len(text) - max_length):
    sequence = text[i:i + max_length]
    label = text[i + max_length]
    X.append([char_to_ix[char] for char in sequence])
    y.append(char_to_ix[label])

X = np.array(X)
y = np.array(y)

# Splitting the dataset into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)

# Converting data to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.long)
y_train = torch.tensor(y_train, dtype=torch.long)
X_val = torch.tensor(X_val, dtype=torch.long)
y_val = torch.tensor(y_val, dtype=torch.long)

# 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
        #This line takes the input tensor x, which contains indices of␣
↪characters, and passes it through an embedding layer (self.embedding).
        #The embedding layer converts these indices into dense vectors of fixed␣
↪size.
        #These vectors are learned during training and can capture semantic␣
↪similarities between characters.
        #The result is a higher-dimensional representation of the input␣
↪sequence, where each character index is replaced by its corresponding␣
↪embedding vector.
        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):
        embedded = self.embedding(x)
        #The LSTM layer returns two outputs:
        #1- the output tensor containing the output of the LSTM at each time␣
↪step for each sequence in the batch,
        #2-the hidden state (_) of the last time step (which is not used in␣
↪this line, hence the underscore).
        output, _ = self.lstm(embedded)
        #The LSTM's output contains the outputs for every time step,
        #but for this task, we're only interested in the output of the last␣
↪time step because we're predicting the next character after the sequence.
        #output[:, -1, :] selects the last time step's output for every␣
↪sequence in the batch (-1 indexes the last item in Python).
        output = self.fc(output[:, -1, :])  # Get the output of the last LSTM␣
↪cell
        return output
```

[ ]:
```
# Hyperparameters
hidden_size = 128
learning_rate = 0.005
epochs = 100

# Model, loss, and optimizer
model = CharRNN(len(chars), hidden_size, len(chars))
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

init_time = time.time()
print("10 sequence LSTM results:")

# Training the model
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()
    output = model(X_train)
    loss = criterion(output, y_train)
    loss.backward()
    optimizer.step()

    # Validation
    model.eval()
    with torch.no_grad():
        val_output = model(X_val)
        val_loss = criterion(val_output, y_val)
        #The use of the underscore _ is a common Python convention to indicate␣
↪that the actual maximum values returned by torch.max are not needed and can␣
↪be disregarded.
```

```
        #What we are interested in is the indices of these maximum values,␣
↪which are captured by the variable predicted. These indices represent the␣
↪model's predictions for each example in the validation set.
        _, predicted = torch.max(val_output, 1)
        val_accuracy = (predicted == y_val).float().mean()

    if (epoch+1) % 20 == 0:
        print(f'Epoch {epoch+1}, Loss: {loss.item()}, Validation Loss:␣
↪{val_loss.item()}, Validation Accuracy: {val_accuracy.item()}')

print(f"Training time: {time.time() - init_time} seconds")
torch.save(model.state_dict(), '../../Models/hw3_1b_10.pth')
```

```
10 sequence LSTM results:
Epoch 20, Loss: 1.773175835609436, Validation Loss: 2.2055509090423584,
Validation Accuracy: 0.4004024267196655
Epoch 40, Loss: 1.1387425661087036, Validation Loss: 2.077422618865967,
Validation Accuracy: 0.47283703088760376
Epoch 60, Loss: 0.6284953355789185, Validation Loss: 2.1856610774993896,
Validation Accuracy: 0.4969818890094757
Epoch 80, Loss: 0.3015962541103363, Validation Loss: 2.4754509925842285,
Validation Accuracy: 0.5050301551818848
Epoch 100, Loss: 0.12140171229839325, Validation Loss: 2.7687766551971436,
Validation Accuracy: 0.5050301551818848
Training time: 3.934805154800415 seconds
```

[ ]:
```
# Preparing the dataset
max_length = 20  # Maximum length of input sequences
X = []
y = []
for i in range(len(text) - max_length):
    sequence = text[i:i + max_length]
    label = text[i + max_length]
    X.append([char_to_ix[char] for char in sequence])
    y.append(char_to_ix[label])

X = np.array(X)
y = np.array(y)

# Splitting the dataset into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)

# Converting data to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.long)
y_train = torch.tensor(y_train, dtype=torch.long)
X_val = torch.tensor(X_val, dtype=torch.long)
```

```
y_val = torch.tensor(y_val, dtype=torch.long)

# 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
        #This line takes the input tensor x, which contains indices of␣
↪characters, and passes it through an embedding layer (self.embedding).
        #The embedding layer converts these indices into dense vectors of fixed␣
↪size.
        #These vectors are learned during training and can capture semantic␣
↪similarities between characters.
        #The result is a higher-dimensional representation of the input␣
↪sequence, where each character index is replaced by its corresponding␣
↪embedding vector.
        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):
        embedded = self.embedding(x)
        #The LSTM layer returns two outputs:
        #1- the output tensor containing the output of the LSTM at each time␣
↪step for each sequence in the batch,
        #2-the hidden state (_) of the last time step (which is not used in␣
↪this line, hence the underscore).
        output, _ = self.lstm(embedded)
        #The LSTM's output contains the outputs for every time step,
        #but for this task, we're only interested in the output of the last␣
↪time step because we're predicting the next character after the sequence.
        #output[:, -1, :] selects the last time step's output for every␣
↪sequence in the batch (-1 indexes the last item in Python).
        output = self.fc(output[:, -1, :])  # Get the output of the last LSTM␣
↪cell
        return output
```

[ ]:
```
# Hyperparameters
hidden_size = 128
learning_rate = 0.005
epochs = 100

# Model, loss, and optimizer
model = CharRNN(len(chars), hidden_size, len(chars))
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

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

# Training the model
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()
    output = model(X_train)
    loss = criterion(output, y_train)
    loss.backward()
    optimizer.step()

    # Validation
    model.eval()
    with torch.no_grad():
        val_output = model(X_val)
        val_loss = criterion(val_output, y_val)
        #The use of the underscore _ is a common Python convention to indicate␣
 ↪that the actual maximum values returned by torch.max are not needed and can␣
 ↪be disregarded.
        #What we are interested in is the indices of these maximum values,␣
 ↪which are captured by the variable predicted. These indices represent the␣
 ↪model's predictions for each example in the validation set.
        _, predicted = torch.max(val_output, 1)
        val_accuracy = (predicted == y_val).float().mean()

    if (epoch+1) % 20 == 0:
        print(f'Epoch {epoch+1}, Loss: {loss.item()}, Validation Loss:␣
 ↪{val_loss.item()}, Validation Accuracy: {val_accuracy.item()}')

print(f"Training time: {time.time() - init_time} seconds")
torch.save(model.state_dict(), '../../Models/hw3_1b_20.pth')
```

```
20 sequence LSTM results:
Epoch 20, Loss: 1.7846384048461914, Validation Loss: 2.164747714996338,
Validation Accuracy: 0.4080808162689209
Epoch 40, Loss: 1.1258690357208252, Validation Loss: 1.9789587259292603,
Validation Accuracy: 0.4909090995788574
Epoch 60, Loss: 0.6051864624023438, Validation Loss: 2.0801331996917725,
Validation Accuracy: 0.513131320476532
Epoch 80, Loss: 0.2641662657260895, Validation Loss: 2.3448119163513184,
Validation Accuracy: 0.513131320476532
Epoch 100, Loss: 0.10667157918214798, Validation Loss: 2.6786410808563232,
Validation Accuracy: 0.5010101199150085
Training time: 7.357742071151733 seconds
```

[ ]:

```
# Preparing the dataset
max_length = 30  # Maximum length of input sequences
X = []
y = []
for i in range(len(text) - max_length):
    sequence = text[i:i + max_length]
    label = text[i + max_length]
    X.append([char_to_ix[char] for char in sequence])
    y.append(char_to_ix[label])

X = np.array(X)
y = np.array(y)

# Splitting the dataset into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)

# Converting data to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.long)
y_train = torch.tensor(y_train, dtype=torch.long)
X_val = torch.tensor(X_val, dtype=torch.long)
y_val = torch.tensor(y_val, dtype=torch.long)

# 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
        #This line takes the input tensor x, which contains indices of␣
↪characters, and passes it through an embedding layer (self.embedding).
        #The embedding layer converts these indices into dense vectors of fixed␣
↪size.
        #These vectors are learned during training and can capture semantic␣
↪similarities between characters.
        #The result is a higher-dimensional representation of the input␣
↪sequence, where each character index is replaced by its corresponding␣
↪embedding vector.
        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):
        embedded = self.embedding(x)
        #The LSTM layer returns two outputs:
        #1- the output tensor containing the output of the LSTM at each time␣
↪step for each sequence in the batch,
```

```
        #2-the hidden state (_) of the last time step (which is not used in␣
↪this line, hence the underscore).
        output, _ = self.lstm(embedded)
        #The LSTM's output contains the outputs for every time step,
        #but for this task, we're only interested in the output of the last␣
↪time step because we're predicting the next character after the sequence.
        #output[:, -1, :] selects the last time step's output for every␣
↪sequence in the batch (-1 indexes the last item in Python).
        output = self.fc(output[:, -1, :])  # Get the output of the last LSTM␣
↪cell
        return output
```

[ ]:
```
# Hyperparameters
hidden_size = 128
learning_rate = 0.005
epochs = 100

# Model, loss, and optimizer
model = CharRNN(len(chars), hidden_size, len(chars))
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

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

# Training the model
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()
    output = model(X_train)
    loss = criterion(output, y_train)
    loss.backward()
    optimizer.step()

    # Validation
    model.eval()
    with torch.no_grad():
        val_output = model(X_val)
        val_loss = criterion(val_output, y_val)
        #The use of the underscore _ is a common Python convention to indicate␣
↪that the actual maximum values returned by torch.max are not needed and can␣
↪be disregarded.
        #What we are interested in is the indices of these maximum values,␣
↪which are captured by the variable predicted. These indices represent the␣
↪model's predictions for each example in the validation set.
        _, predicted = torch.max(val_output, 1)
        val_accuracy = (predicted == y_val).float().mean()
```

```
    if (epoch+1) % 20 == 0:
        print(f'Epoch {epoch+1}, Loss: {loss.item()}, Validation Loss:␣
↪{val_loss.item()}, Validation Accuracy: {val_accuracy.item()}')

print(f"Training time: {time.time() - init_time} seconds")
torch.save(model.state_dict(), '../../Models/hw3_1b_30.pth')
```

```
30 sequence LSTM results:
Epoch 20, Loss: 1.7847687005996704, Validation Loss: 2.092292547225952,
Validation Accuracy: 0.4482758641242981
Epoch 40, Loss: 1.1175408363342285, Validation Loss: 1.9323941469192505,
Validation Accuracy: 0.49087220430374146
Epoch 60, Loss: 0.6051943898200989, Validation Loss: 2.069451332092285,
Validation Accuracy: 0.5050709843635559
Epoch 80, Loss: 0.2756587862968445, Validation Loss: 2.357543706893921,
Validation Accuracy: 0.4949290156364441
Epoch 100, Loss: 0.13489699363708496, Validation Loss: 2.616608142852783,
Validation Accuracy: 0.4929006099700928
Training time: 11.018937110900879 seconds
```

3 Problem 1C: GRU(10, 20, 30)

[ ]:
```
# Preparing the dataset
max_length = 10  # Maximum length of input sequences
X = []
y = []
for i in range(len(text) - max_length):
    sequence = text[i:i + max_length]
    label = text[i + max_length]
    X.append([char_to_ix[char] for char in sequence])
    y.append(char_to_ix[label])

X = np.array(X)
y = np.array(y)

# Splitting the dataset into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)

# Converting data to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.long)
y_train = torch.tensor(y_train, dtype=torch.long)
X_val = torch.tensor(X_val, dtype=torch.long)
y_val = torch.tensor(y_val, dtype=torch.long)

# Defining the GRU model
```

```python
class CharGRU(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(CharGRU, self).__init__()
        self.hidden_size = hidden_size
        #This line takes the input tensor x, which contains indices of characters, and passes it through an embedding layer (self.embedding).
        #The embedding layer converts these indices into dense vectors of fixed size.
        #These vectors are learned during training and can capture semantic similarities between characters.
        #The result is a higher-dimensional representation of the input sequence, where each character index is replaced by its corresponding embedding vector.
        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):
        embedded = self.embedding(x)
        #The GRU layer returns two outputs:
        #1- the output tensor containing the output of the GRU at each time step for each sequence in the batch,
        #2-the hidden state (_) of the last time step (which is not used in this line, hence the underscore).
        output, _ = self.gru(embedded)
        #The GRU's output contains the outputs for every time step,
        #but for this task, we're only interested in the output of the last time step because we're predicting the next character after the sequence.
        #output[:, -1, :] selects the last time step's output for every sequence in the batch (-1 indexes the last item in Python).
        output = self.fc(output[:, -1, :])  # Get the output of the last GRU cell
        return output
```

[ ]:

```python
# Hyperparameters
hidden_size = 128
learning_rate = 0.005
epochs = 100

# Model, loss, and optimizer
model = CharRNN(len(chars), hidden_size, len(chars))
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

init_time = time.time()
print("10 sequence GRU results:")
```

```
# Training the model
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()
    output = model(X_train)
    loss = criterion(output, y_train)
    loss.backward()
    optimizer.step()

    # Validation
    model.eval()
    with torch.no_grad():
        val_output = model(X_val)
        val_loss = criterion(val_output, y_val)
        #The use of the underscore _ is a common Python convention to indicate␣
 ↪that the actual maximum values returned by torch.max are not needed and can␣
 ↪be disregarded.
        #What we are interested in is the indices of these maximum values,␣
 ↪which are captured by the variable predicted. These indices represent the␣
 ↪model's predictions for each example in the validation set.
        _, predicted = torch.max(val_output, 1)
        val_accuracy = (predicted == y_val).float().mean()

    if (epoch+1) % 20 == 0:
        print(f'Epoch {epoch+1}, Loss: {loss.item()}, Validation Loss:␣
 ↪{val_loss.item()}, Validation Accuracy: {val_accuracy.item()}')

print(f"Training time: {time.time() - init_time} seconds")
torch.save(model.state_dict(), '../../Models/hw3_1c_10.pth')
```

```
10 sequence GRU results:
Epoch 20, Loss: 1.7582334280014038, Validation Loss: 2.2070350646972656,
Validation Accuracy: 0.4124748408794403
Epoch 40, Loss: 1.121056318283081, Validation Loss: 2.0890400409698486,
Validation Accuracy: 0.46277666091918945
Epoch 60, Loss: 0.6207669377326965, Validation Loss: 2.198493242263794,
Validation Accuracy: 0.49899396300315857
Epoch 80, Loss: 0.2994094491004944, Validation Loss: 2.490537643432617,
Validation Accuracy: 0.5090543031692505
Epoch 100, Loss: 0.1344083696603775, Validation Loss: 2.7802088260650635,
Validation Accuracy: 0.501006007194519
Training time: 4.025479316711426 seconds
```

[ ]:
```
# Preparing the dataset
max_length = 20  # Maximum length of input sequences
X = []
```

```
y = []
for i in range(len(text) - max_length):
    sequence = text[i:i + max_length]
    label = text[i + max_length]
    X.append([char_to_ix[char] for char in sequence])
    y.append(char_to_ix[label])

X = np.array(X)
y = np.array(y)

# Splitting the dataset into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

# Converting data to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.long)
y_train = torch.tensor(y_train, dtype=torch.long)
X_val = torch.tensor(X_val, dtype=torch.long)
y_val = torch.tensor(y_val, dtype=torch.long)

# 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
        #This line takes the input tensor x, which contains indices of characters, and passes it through an embedding layer (self.embedding).
        #The embedding layer converts these indices into dense vectors of fixed size.
        #These vectors are learned during training and can capture semantic similarities between characters.
        #The result is a higher-dimensional representation of the input sequence, where each character index is replaced by its corresponding embedding vector.
        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):
        embedded = self.embedding(x)
        #The GRU layer returns two outputs:
        #1- the output tensor containing the output of the GRU at each time step for each sequence in the batch,
        #2-the hidden state (_) of the last time step (which is not used in this line, hence the underscore).
        output, _ = self.gru(embedded)
```

```python
        #The GRU's output contains the outputs for every time step,
        #but for this task, we're only interested in the output of the last time step because we're predicting the next character after the sequence.
        #output[:, -1, :] selects the last time step's output for every sequence in the batch (-1 indexes the last item in Python).
        output = self.fc(output[:, -1, :])  # Get the output of the last GRU cell
        return output
```

[ ]:
```python
# Hyperparameters
hidden_size = 128
learning_rate = 0.005
epochs = 100

# Model, loss, and optimizer
model = CharRNN(len(chars), hidden_size, len(chars))
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

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

# Training the model
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()
    output = model(X_train)
    loss = criterion(output, y_train)
    loss.backward()
    optimizer.step()

    # Validation
    model.eval()
    with torch.no_grad():
        val_output = model(X_val)
        val_loss = criterion(val_output, y_val)
        #The use of the underscore _ is a common Python convention to indicate that the actual maximum values returned by torch.max are not needed and can be disregarded.
        #What we are interested in is the indices of these maximum values, which are captured by the variable predicted. These indices represent the model's predictions for each example in the validation set.
        _, predicted = torch.max(val_output, 1)
        val_accuracy = (predicted == y_val).float().mean()

    if (epoch+1) % 20 == 0:
```

```
        print(f'Epoch {epoch+1}, Loss: {loss.item()}, Validation Loss:␣
↪{val_loss.item()}, Validation Accuracy: {val_accuracy.item()}')

print(f"Training time: {time.time() - init_time} seconds")
torch.save(model.state_dict(), '../../Models/hw3_1c_20.pth')
```

```
20 sequence GRU results:
Epoch 20, Loss: 1.8342708349227905, Validation Loss: 2.169351816177368,
Validation Accuracy: 0.4121212065219879
Epoch 40, Loss: 1.1893103122711182, Validation Loss: 1.9785033464431763,
Validation Accuracy: 0.4747474789619446
Epoch 60, Loss: 0.6928154230117798, Validation Loss: 1.9990766048431396,
Validation Accuracy: 0.5111111402511597
Epoch 80, Loss: 0.33933183550834656, Validation Loss: 2.166846513748169,
Validation Accuracy: 0.5272727012634277
Epoch 100, Loss: 0.1809617131948471, Validation Loss: 2.403912305831909,
Validation Accuracy: 0.5252525210380554
Training time: 7.495378017425537 seconds
```

[ ]:
```
# Preparing the dataset
max_length = 30  # Maximum length of input sequences
X = []
y = []
for i in range(len(text) - max_length):
    sequence = text[i:i + max_length]
    label = text[i + max_length]
    X.append([char_to_ix[char] for char in sequence])
    y.append(char_to_ix[label])

X = np.array(X)
y = np.array(y)

# Splitting the dataset into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)

# Converting data to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.long)
y_train = torch.tensor(y_train, dtype=torch.long)
X_val = torch.tensor(X_val, dtype=torch.long)
y_val = torch.tensor(y_val, dtype=torch.long)

# 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
```

```
        #This line takes the input tensor x, which contains indices of␣
↪characters, and passes it through an embedding layer (self.embedding).
        #The embedding layer converts these indices into dense vectors of fixed␣
↪size.
        #These vectors are learned during training and can capture semantic␣
↪similarities between characters.
        #The result is a higher-dimensional representation of the input␣
↪sequence, where each character index is replaced by its corresponding␣
↪embedding vector.
        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):
        embedded = self.embedding(x)
        #The GRU layer returns two outputs:
        #1- the output tensor containing the output of the GRU at each time␣
↪step for each sequence in the batch,
        #2-the hidden state (_) of the last time step (which is not used in␣
↪this line, hence the underscore).
        output, _ = self.gru(embedded)
        #The GRU's output contains the outputs for every time step,
        #but for this task, we're only interested in the output of the last␣
↪time step because we're predicting the next character after the sequence.
        #output[:, -1, :] selects the last time step's output for every␣
↪sequence in the batch (-1 indexes the last item in Python).
        output = self.fc(output[:, -1, :])  # Get the output of the last GRU␣
↪cell
        return output
```

[ ]:
```
# Hyperparameters
hidden_size = 128
learning_rate = 0.005
epochs = 100

# Model, loss, and optimizer
model = CharRNN(len(chars), hidden_size, len(chars))
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

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

# Training the model
for epoch in range(epochs):
    model.train()
```

```
    optimizer.zero_grad()
    output = model(X_train)
    loss = criterion(output, y_train)
    loss.backward()
    optimizer.step()

    # Validation
    model.eval()
    with torch.no_grad():
        val_output = model(X_val)
        val_loss = criterion(val_output, y_val)
        #The use of the underscore _ is a common Python convention to indicate␣
↪that the actual maximum values returned by torch.max are not needed and can␣
↪be disregarded.
        #What we are interested in is the indices of these maximum values,␣
↪which are captured by the variable predicted. These indices represent the␣
↪model's predictions for each example in the validation set.
        _, predicted = torch.max(val_output, 1)
        val_accuracy = (predicted == y_val).float().mean()

    if (epoch+1) % 20 == 0:
        print(f'Epoch {epoch+1}, Loss: {loss.item()}, Validation Loss:␣
↪{val_loss.item()}, Validation Accuracy: {val_accuracy.item()}')

print(f"Training time: {time.time() - init_time} seconds")
torch.save(model.state_dict(), '../../Models/hw3_1c_30.pth')
```

```
30 sequence GRU results:
Epoch 20, Loss: 1.8461445569992065, Validation Loss: 2.1086063385009766,
Validation Accuracy: 0.4279918968677521
Epoch 40, Loss: 1.1958062648773193, Validation Loss: 1.879054069519043,
Validation Accuracy: 0.5192697644233704
Epoch 60, Loss: 0.7067962884902954, Validation Loss: 1.8773630857467651,
Validation Accuracy: 0.5375253558158875
Epoch 80, Loss: 0.3689146935939789, Validation Loss: 2.057988405227661,
Validation Accuracy: 0.5192697644233704
Epoch 100, Loss: 0.17575804889202118, Validation Loss: 2.311189651489258,
Validation Accuracy: 0.5131846070289612
Training time: 10.908366203308105 seconds
```