hw5q1

April 4, 2025

[1]:
```
#sequence length 10

import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import time
import math
from sklearn.model_selection import train_test_split
import torchinfo
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f'Using device: {device}')

# Sample text
```

```python
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.'''

# Preparing the dataset for sequence prediction
max_length = 10  # Maximum length of input sequences
sequences = [text[i:i + max_length] for i in range(len(text) - max_length)]
labels = [text[i + max_length] for i in range(len(text) - max_length)]

# Creating character vocabulary
chars = sorted(list(set(text)))
char_to_ix = {ch: i for i, ch in enumerate(chars)}

# Convert sequences and labels to tensors
X = torch.tensor([[char_to_ix[ch] for ch in seq] for seq in sequences], dtype=torch.long).to(device)
```

```python
y = torch.tensor([char_to_ix[label] for label in labels], dtype=torch.long).
↪to(device)

# 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)

# Define Transformer model
class CharTransformer(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers, nhead):
        super(CharTransformer, self).__init__()
        self.embedding = nn.Embedding(input_size, hidden_size)
        encoder_layers = nn.TransformerEncoderLayer(hidden_size, nhead)
        self.transformer_encoder = nn.TransformerEncoder(encoder_layers,␣
↪num_layers)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        embedded = self.embedding(x)
        transformer_output = self.transformer_encoder(embedded)
        output = self.fc(transformer_output[:, -1, :])  # Get output of last␣
↪Transformer block
        return output

# Hyperparameters
hidden_size = 128
num_layers = 3
nhead = 2
learning_rate = 0.001
epochs = 50

# Model, loss, and optimizer
model = CharTransformer(len(chars), hidden_size, len(chars), num_layers, nhead).
↪to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Display model summary
summary = torchinfo.summary(model, input_data=X_train)
print(summary)

# Training the model
total_start_time = time.time()
for epoch in range(epochs):
    start_time = time.time()
    model.train()
    optimizer.zero_grad()
```

```python
    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)
        _, predicted = torch.max(val_output, 1)
        val_accuracy = (predicted == y_val).float().mean()

    if (epoch+1) % 5 == 0:
        end_time = time.time()
        execution_time = end_time - start_time
        print(f'Epoch {epoch+1}, Loss: {loss.item()}, Validation Loss: 
 ↪{val_loss.item()}, Validation Accuracy: {val_accuracy.item()}, Execution 
 ↪Time: {execution_time} seconds')

total_end_time = time.time()
total_execution_time = total_end_time - total_start_time
print(f'Total Execution Time: {total_execution_time} seconds')
```

```
Using device: cuda
```

```
/home/dman/.venv/master/lib/python3.12/site-
packages/torch/nn/modules/transformer.py:385: UserWarning: enable_nested_tensor
is True, but self.use_nested_tensor is False because
encoder_layer.self_attn.batch_first was not True(use batch_first for better
inference performance)
  warnings.warn(
```

```
================================================================================
===============
Layer (type:depth-idx)                        Output Shape              Param #
================================================================================
===============
CharTransformer                               [1900, 44]                --
 Embedding: 1-1                               [1900, 10, 128]           5,632
 TransformerEncoder: 1-2                      [1900, 10, 128]           --
     ModuleList: 2-1                          --                        --
         TransformerEncoderLayer: 3-1         [1900, 10, 128]           593,024
         TransformerEncoderLayer: 3-2         [1900, 10, 128]           593,024
         TransformerEncoderLayer: 3-3         [1900, 10, 128]           593,024
 Linear: 1-3                                  [1900, 44]                5,676
================================================================================
===============
Total params: 1,790,380
```

```
Trainable params: 1,790,380
Non-trainable params: 0
Total mult-adds (Units.GIGABYTES): 3.03
================================================================================
===============
Input size (MB): 0.15
Forward/backward pass size (MB): 1129.12
Params size (MB): 6.37
Estimated Total Size (MB): 1135.64
================================================================================
===============
Epoch 5, Loss: 2.8971612453460693, Validation Loss: 2.762009859085083,
Validation Accuracy: 0.21848741173744202, Execution Time: 0.08399343490600586
seconds
Epoch 10, Loss: 2.608732223510742, Validation Loss: 2.5518441200256348,
Validation Accuracy: 0.2920168340206146, Execution Time: 0.13433384895324707
seconds
Epoch 15, Loss: 2.476253032684326, Validation Loss: 2.4380640983581543,
Validation Accuracy: 0.2899159789085388, Execution Time: 0.12561368942260742
seconds
Epoch 20, Loss: 2.403218984603882, Validation Loss: 2.411163330078125,
Validation Accuracy: 0.28151261806488037, Execution Time: 0.1250901222229004
seconds
Epoch 25, Loss: 2.3485896587371826, Validation Loss: 2.371657371520996,
Validation Accuracy: 0.2878151535987854, Execution Time: 0.12447786331176758
seconds
Epoch 30, Loss: 2.3031558990478516, Validation Loss: 2.3365249633789062,
Validation Accuracy: 0.2878151535987854, Execution Time: 0.12671923637390137
seconds
Epoch 35, Loss: 2.279742956161499, Validation Loss: 2.321427583694458,
Validation Accuracy: 0.2899159789085388, Execution Time: 0.12458181381225586
seconds
Epoch 40, Loss: 2.2605137825012207, Validation Loss: 2.316436529159546,
Validation Accuracy: 0.2857142984867096, Execution Time: 0.12621021270751953
seconds
Epoch 45, Loss: 2.2435638904571533, Validation Loss: 2.311800956726074,
Validation Accuracy: 0.2878151535987854, Execution Time: 0.12342119216918945
seconds
Epoch 50, Loss: 2.237259864807129, Validation Loss: 2.3116962909698486,
Validation Accuracy: 0.2878151535987854, Execution Time: 0.12564873695373535
seconds
Total Execution Time: 6.421080589294434 seconds
```

[2]:
```
#sequence length 20
# Sample text
```

```python
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.'''

# Preparing the dataset for sequence prediction
max_length = 20  # Maximum length of input sequences
sequences = [text[i:i + max_length] for i in range(len(text) - max_length)]
labels = [text[i + max_length] for i in range(len(text) - max_length)]

# Creating character vocabulary
chars = sorted(list(set(text)))
char_to_ix = {ch: i for i, ch in enumerate(chars)}

# Convert sequences and labels to tensors
X = torch.tensor([[char_to_ix[ch] for ch in seq] for seq in sequences], dtype=torch.long).to(device)
```

```
y = torch.tensor([char_to_ix[label] for label in labels], dtype=torch.long).
 ↪to(device)

# 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)

# Define Transformer model
class CharTransformer(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers, nhead):
        super(CharTransformer, self).__init__()
        self.embedding = nn.Embedding(input_size, hidden_size)
        encoder_layers = nn.TransformerEncoderLayer(hidden_size, nhead)
        self.transformer_encoder = nn.TransformerEncoder(encoder_layers,␣
 ↪num_layers)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        embedded = self.embedding(x)
        transformer_output = self.transformer_encoder(embedded)
        output = self.fc(transformer_output[:, -1, :])  # Get output of last␣
 ↪Transformer block
        return output

# Hyperparameters
hidden_size = 128
num_layers = 3
nhead = 2
learning_rate = 0.001
epochs = 50

# Model, loss, and optimizer
model = CharTransformer(len(chars), hidden_size, len(chars), num_layers, nhead).
 ↪to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Display model summary
summary = torchinfo.summary(model, input_data=X_train)
print(summary)

# Training the model
total_start_time = time.time()
for epoch in range(epochs):
    start_time = time.time()
    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)
        _, predicted = torch.max(val_output, 1)
        val_accuracy = (predicted == y_val).float().mean()

    if (epoch+1) % 5 == 0:
        end_time = time.time()
        execution_time = end_time - start_time
        print(f'Epoch {epoch+1}, Loss: {loss.item()}, Validation Loss:␣
↪{val_loss.item()}, Validation Accuracy: {val_accuracy.item()}, Execution␣
↪Time: {execution_time} seconds')

total_end_time = time.time()
total_execution_time = total_end_time - total_start_time
print(f'Total Execution Time: {total_execution_time} seconds')
```

```
================================================================================
===============
Layer (type:depth-idx)                        Output Shape              Param #
================================================================================
===============
CharTransformer                               [1892, 44]                --
 Embedding: 1-1                               [1892, 20, 128]           5,632
 TransformerEncoder: 1-2                      [1892, 20, 128]           --
     ModuleList: 2-1                          --                        --
         TransformerEncoderLayer: 3-1         [1892, 20, 128]           593,024
         TransformerEncoderLayer: 3-2         [1892, 20, 128]           593,024
         TransformerEncoderLayer: 3-3         [1892, 20, 128]           593,024
 Linear: 1-3                                  [1892, 44]                5,676
================================================================================
===============
Total params: 1,790,380
Trainable params: 1,790,380
Non-trainable params: 0
Total mult-adds (Units.GIGABYTES): 3.01
================================================================================
===============
Input size (MB): 0.30
Forward/backward pass size (MB): 2248.06
Params size (MB): 6.37
```

```
Estimated Total Size (MB): 2254.73
================================================================================
===============

/home/dman/.venv/master/lib/python3.12/site-
packages/torch/nn/modules/transformer.py:385: UserWarning: enable_nested_tensor
is True, but self.use_nested_tensor is False because
encoder_layer.self_attn.batch_first was not True(use batch_first for better
inference performance)
  warnings.warn(

Epoch 5, Loss: 2.8855316638946533, Validation Loss: 2.79542875289917, Validation
Accuracy: 0.2109704613685608, Execution Time: 0.21418094635009766 seconds
Epoch 10, Loss: 2.5777204036712646, Validation Loss: 2.5451254844665527,
Validation Accuracy: 0.23839661478996277, Execution Time: 0.24049806594848633
seconds
Epoch 15, Loss: 2.4539241790771484, Validation Loss: 2.4622461795806885,
Validation Accuracy: 0.2805907130241394, Execution Time: 0.24182915687561035
seconds
Epoch 20, Loss: 2.3868677616119385, Validation Loss: 2.4474551677703857,
Validation Accuracy: 0.2763713002204895, Execution Time: 0.24112629890441895
seconds
Epoch 25, Loss: 2.3527097702026367, Validation Loss: 2.416934013366699,
Validation Accuracy: 0.24894513189792633, Execution Time: 0.24055862426757812
seconds
Epoch 30, Loss: 2.310042142868042, Validation Loss: 2.386185646057129,
Validation Accuracy: 0.25738394260406494, Execution Time: 0.2419743537902832
seconds
Epoch 35, Loss: 2.2733840942382812, Validation Loss: 2.382976531982422,
Validation Accuracy: 0.25316452980041504, Execution Time: 0.2410719394683838
seconds
Epoch 40, Loss: 2.264220714569092, Validation Loss: 2.361839532852173,
Validation Accuracy: 0.25316452980041504, Execution Time: 0.24018406867980957
seconds
Epoch 45, Loss: 2.2494311332702637, Validation Loss: 2.3602750301361084,
Validation Accuracy: 0.2679324746131897, Execution Time: 0.23980093002319336
seconds
Epoch 50, Loss: 2.2300937175750732, Validation Loss: 2.364194869995117,
Validation Accuracy: 0.27426159381866455, Execution Time: 0.24025893211364746
seconds
Total Execution Time: 11.98580002784729 seconds
```

[3]:

```
#sequence length 30

# Sample text
```

```
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.'''

# Preparing the dataset for sequence prediction
max_length = 30  # Maximum length of input sequences
sequences = [text[i:i + max_length] for i in range(len(text) - max_length)]
labels = [text[i + max_length] for i in range(len(text) - max_length)]

# Creating character vocabulary
chars = sorted(list(set(text)))
char_to_ix = {ch: i for i, ch in enumerate(chars)}

# Convert sequences and labels to tensors
X = torch.tensor([[char_to_ix[ch] for ch in seq] for seq in sequences],␣
 ↪dtype=torch.long).to(device)
```

```
y = torch.tensor([char_to_ix[label] for label in labels], dtype=torch.long).
 ↪to(device)

# 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)

# Define Transformer model
class CharTransformer(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers, nhead):
        super(CharTransformer, self).__init__()
        self.embedding = nn.Embedding(input_size, hidden_size)
        encoder_layers = nn.TransformerEncoderLayer(hidden_size, nhead)
        self.transformer_encoder = nn.TransformerEncoder(encoder_layers,␣
 ↪num_layers)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        embedded = self.embedding(x)
        transformer_output = self.transformer_encoder(embedded)
        output = self.fc(transformer_output[:, -1, :])  # Get output of last␣
 ↪Transformer block
        return output

# Hyperparameters
hidden_size = 128
num_layers = 3
nhead = 2
learning_rate = 0.001
epochs = 50

# Model, loss, and optimizer
model = CharTransformer(len(chars), hidden_size, len(chars), num_layers, nhead).
 ↪to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Display model summary
summary = torchinfo.summary(model, input_data=X_train)
print(summary)

# Training the model
total_start_time = time.time()
for epoch in range(epochs):
    start_time = time.time()
    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)
        _, predicted = torch.max(val_output, 1)
        val_accuracy = (predicted == y_val).float().mean()

    if (epoch+1) % 5 == 0:
        end_time = time.time()
        execution_time = end_time - start_time
        print(f'Epoch {epoch+1}, Loss: {loss.item()}, Validation Loss:␣
↪{val_loss.item()}, Validation Accuracy: {val_accuracy.item()}, Execution␣
↪Time: {execution_time} seconds')

total_end_time = time.time()
total_execution_time = total_end_time - total_start_time
print(f'Total Execution Time: {total_execution_time} seconds')
```

```
================================================================================
===============
Layer (type:depth-idx)                        Output Shape              Param #
================================================================================
===============
CharTransformer                                [1884, 44]                --
 Embedding: 1-1                               [1884, 30, 128]           5,632
 TransformerEncoder: 1-2                      [1884, 30, 128]           --
      ModuleList: 2-1                         --                        --
           TransformerEncoderLayer: 3-1      [1884, 30, 128]           593,024
           TransformerEncoderLayer: 3-2      [1884, 30, 128]           593,024
           TransformerEncoderLayer: 3-3      [1884, 30, 128]           593,024
 Linear: 1-3                                  [1884, 44]                5,676
================================================================================
===============
Total params: 1,790,380
Trainable params: 1,790,380
Non-trainable params: 0
Total mult-adds (Units.GIGABYTES): 3.00
================================================================================
===============
Input size (MB): 0.45
Forward/backward pass size (MB): 3357.50
Params size (MB): 6.37
```

```
Estimated Total Size (MB): 3364.32
================================================================================
===============
Epoch 5, Loss: 2.9286999702453613, Validation Loss: 2.8957574367523193,
Validation Accuracy: 0.17796610295772552, Execution Time: 0.3450007438659668
seconds
Epoch 10, Loss: 2.6100516319274902, Validation Loss: 2.6570611000061035,
Validation Accuracy: 0.24152542650699615, Execution Time: 0.3161318302154541
seconds
Epoch 15, Loss: 2.4706196784973145, Validation Loss: 2.5679800510406494,
Validation Accuracy: 0.2330508530139923, Execution Time: 0.2823753356933594
seconds
Epoch 20, Loss: 2.394347906112671, Validation Loss: 2.5069236755371094,
Validation Accuracy: 0.24576270580291748, Execution Time: 0.28311777114868164
seconds
Epoch 25, Loss: 2.3387489318847656, Validation Loss: 2.484586477279663,
Validation Accuracy: 0.24788135290145874, Execution Time: 0.2780444622039795
seconds
Epoch 30, Loss: 2.2942371368408203, Validation Loss: 2.457167625427246,
Validation Accuracy: 0.24576270580291748, Execution Time: 0.28285980224609375
seconds
Epoch 35, Loss: 2.26870059967041, Validation Loss: 2.449058771133423, Validation
Accuracy: 0.25, Execution Time: 0.28238797187805176 seconds
Epoch 40, Loss: 2.2444820404052734, Validation Loss: 2.4435513019561768,
Validation Accuracy: 0.24788135290145874, Execution Time: 0.31443238258361816
seconds
Epoch 45, Loss: 2.232463836669922, Validation Loss: 2.4325082302093506,
Validation Accuracy: 0.24788135290145874, Execution Time: 0.2831428050994873
seconds
Epoch 50, Loss: 2.2242465019226074, Validation Loss: 2.435275077819824,
Validation Accuracy: 0.24788135290145874, Execution Time: 0.27874326705932617
seconds
Total Execution Time: 14.616542339324951 seconds
```