Tecnológico
de Monterrey

TC 5033

Text Generation

Team Members:

- A01200230 - Armando Bringas Corpus

Activity 4: Building a Simple LSTM Text Generator using WikiText-2

- Objective:
  - Gain a fundamental understanding of Long Short-Term Memory (LSTM) networks.
  - Develop hands-on experience with sequence data processing and text generation in PyTorch. Given the simplicity of the model, amount of data, and computer resources, the text you generate will not replace ChatGPT, and results must likely will not make a lot of sense. Its only purpose is academic and to understand the text generation using RNNs.
  - Enhance code comprehension and documentation skills by commenting on provided starter code.

- Instructions:
  - Code Understanding: Begin by thoroughly reading and understanding the code. Comment each section/block of the provided code to demonstrate your understanding. For this, you are encouraged to add cells with experiments to improve your understanding

    - Model Overview: The starter code includes an LSTM model setup for sequence data processing. Familiarize yourself with the model architecture and its components. Once you are familiar with the provided model, feel free to change the model to experiment.

    - Training Function: Implement a function to train the LSTM model on the WikiText-2 dataset. This function should feed the training data into the model and perform backpropagation.

    - Text Generation Function: Create a function that accepts starting text (seed text) and a specified total number of words to generate. The function should use the trained model to generate a continuation of the input text.

    - Code Commenting: Ensure that all the provided starter code is well-commented. Explain the purpose and functionality of each section, indicating your understanding.

    - Submission: Submit your Jupyter Notebook with all sections completed and commented. Include a markdown cell with the full names of all contributing team members at the beginning of the notebook.

- Evaluation Criteria:
    - Code Commenting (60%): The clarity, accuracy, and thoroughness of comments explaining the provided code. You are suggested to use markdown cells for your explanations.

    - Training Function Implementation (20%): The correct implementation of the training function, which should effectively train the model.

    - Text Generation Functionality (10%): A working function is provided in comments. You are free to use it as long as you make sure to uderstand it, you may as well improve it as you see fit. The minimum expected is to provide comments for the given function.

    - Conclusions (10%): Provide some final remarks specifying the differences you notice between this model and the one used for classification tasks. Also comment on changes you made to the model, hyperparameters, and any other information you consider relevant. Also, please provide 3 examples of generated texts.

Import libraries

In [1]:
```
import numpy as np
#PyTorch Libraries
import torch
import torchtext
from torchtext.datasets import WikiText2
# Dataloader library
from torch.utils.data import DataLoader, TensorDataset
from torch.utils.data.dataset import random_split
# Libraries to prepare the data
from torchtext.data.utils import get_tokenizer
from torchtext.vocab import build_vocab_from_iterator
from torchtext.data.functional import to_map_style_dataset
# neural layers
from torch import nn
from torch.nn import functional as F
import torch.optim as optim
from tqdm import tqdm

import random

# Added libraries
import math
import time
```

In [2]:
```
# Check torch version
torch.__version__
```

Out[2]:
```
'2.1.0'
```

In [3]:
```
# Use GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)
```

```
cuda
```

In [4]:
```
if torch.cuda.is_available():
    print(torch.cuda.get_device_name(0))
    print(torch.cuda.get_device_capability(0))
    print(torch.cuda.get_device_properties(0))
else:
    print("No GPU available")
```

```
NVIDIA GeForce GTX 1650
(7, 5)
_CudaDeviceProperties(name='NVIDIA GeForce GTX 1650', major=7, minor=5, total_memory
=4095MB, multi_processor_count=14)
```

Get the train and the test datasets and dataloaders

In [5]:
```
train_dataset, val_dataset, test_dataset = WikiText2()
```

In [6]:
```
tokeniser = get_tokenizer('basic_english')
def yield_tokens(data):
    for text in data:
        yield tokeniser(text)
```

In [7]:
```
# Build the vocabulary
vocab = build_vocab_from_iterator(yield_tokens(train_dataset), specials=["<unk>", "
#set unknown token at position 0
vocab.set_default_index(vocab["<unk>"])
```

In [8]:
```
seq_length = 50
def data_process(raw_text_iter, seq_length = 50):
    data = [torch.tensor(vocab(tokeniser(item)), dtype=torch.long) for item in raw_
    data = torch.cat(tuple(filter(lambda t: t.numel() > 0, data))) #remove empty te
#     target_data = torch.cat(d)
    return (data[:-(data.size(0)%seq_length)].view(-1, seq_length),
            data[1:-(data.size(0)%seq_length-1)].view(-1, seq_length))

# # Create tensors for the training set
x_train, y_train = data_process(train_dataset, seq_length)
x_val, y_val = data_process(val_dataset, seq_length)
x_test, y_test = data_process(test_dataset, seq_length)
```

In [9]:
```
train_dataset = TensorDataset(x_train, y_train)
val_dataset = TensorDataset(x_val, y_val)
test_dataset = TensorDataset(x_test, y_test)
```

In [10]:
```
batch_size = 128  # choose a batch size that fits your computation resources
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, drop_
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=True, drop_last
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True, drop_la
```

LSTM Model definition

LSTM class

This LSTM-based model is an architecture for handle various sequence processing tasks in NLP. It has a combination of embedding layers, LSTM cells, dropout, and fully connected layers, along with weight initialization. This LSTM-based model is suitable for text generation tasks.

In this case we added some initialization for the weights, we check through the PyTorch documentation for the type on initializators: https://pytorch.org/docs/stable/nn.init.html

In [11]:
```
# Define the LSTM model
# Feel free to experiment
class LSTMModel(nn.Module):
    def __init__(self, vocab_size, embed_size, hidden_size, num_layers, dropout_rat
        super(LSTMModel, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embed_size)
        self.embedding_size = embed_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        # As regularization added dropout in LSTM and after as a layer
        self.lstm = nn.LSTM(embed_size, hidden_size, num_layers, dropout=dropout_ra
        self.dropout = nn.Dropout(dropout_rate)
        self.fc = nn.Linear(hidden_size, vocab_size)
        self.init_weights() # Added weight layer
    def forward(self, text, hidden=None):
        embeddings = self.embeddings(text)
        output, hidden = self.lstm(embeddings, hidden)
        # As regularization added dropout in the output
        output = self.dropout(output)
        decoded = self.fc(output)
        return decoded, hidden

    def init_hidden(self, batch_size):
        return (torch.zeros(self.num_layers, batch_size, self.hidden_size).to(devic
                torch.zeros(self.num_layers, batch_size, self.hidden_size).to(devic

    # Added initiallization weights
    def init_weights(self):
        initrange = 0.1
        # Initialize embeddings
        self.embeddings.weight.data.uniform_(-initrange, initrange)
        # Initialize the linear layer
        self.fc.bias.data.zero_()
        self.fc.weight.data.uniform_(-initrange, initrange)
        # Initialization schemes for LSTM: https://pytorch.org/docs/stable/nn.init.
        for name, param in self.lstm.named_parameters():
            if 'weight_ih' in name:
                # Glorot - input-hidden weights
                nn.init.xavier_uniform_(param.data)
            elif 'weight_hh' in name:
                # Orthogonal - hidden-hidden weights
                nn.init.orthogonal_(param.data)
            elif 'bias' in name:
                # Bias
                param.data.fill_(0)
```

Training function

In [12]:
```
def train(model, epochs, optimiser):
    '''
    The following are possible instructions you may want to conside for this functi
    This is only a guide and you may change add or remove whatever you consider app
    as long as you train your model correctly.
        - loop through specified epochs [ok]
        - loop through dataloader [ok]
        - don't forget to zero grad! [ok]
        - place data (both input and target) in device [ok]
        - init hidden states e.g. hidden = model.init_hidden(batch_size) [ok]
        - run the model [ok]
        - compute the cost or loss [ok]
        - backpropagation [ok]
        - Update paratemers [ok]
        - Include print all the information you consider helpful

    '''

    # Assign model to current processing device
    model = model.to(device=device)

    # Start counting total training time
    total_start_time = time.time()

    # Iterate through epochs
    for epoch in range(epochs):
        # Start epoch timing
        start_time = time.time()

        # Put model on training mode
        model.train()

        total_loss = 0
        for i, (data, targets) in enumerate(train_loader):
            # Assign data to the correct device
            data = data.to(device=device, dtype=torch.int64)
            targets = targets.to(device=device, dtype=torch.int64)

            # Initialize hidden states
            batch_size = data.size(0)
            hidden = model.init_hidden(batch_size)
            hidden = tuple([each.data for each in hidden])

            # Calculate prediction scores, forward pass
            scores, hidden = model(data, hidden)

            # Calculate the loss
            scores = scores.view(-1, vocab_size)
            targets = targets.view(-1)
            loss = F.cross_entropy(scores, targets)
            total_loss += loss.item()

            # Perform backward pass and optimize
            optimiser.zero_grad()
            loss.backward()
            optimiser.step()

            # Print batch loss
            if i % 100 == 0:  # print every 100 batches
                print(f'Epoch {epoch}, Batch {i}, Loss {loss.item()}')

        # Calculate average loss and elapsed time for the epoch
        avg_loss = total_loss / len(train_loader)
        elapsed_time = time.time() - start_time

        # Print epoch loss and time
        print('\n' + '-' * 60)
        print(f'Epoch {epoch} completed with average loss {avg_loss:.4f} in {elapse
        print('-' * 60 + '\n')

    # Print total elapsed time for training
    total_elapsed_time = time.time() - total_start_time
    hours = total_elapsed_time // 3600
    minutes = (total_elapsed_time % 3600) // 60
    seconds = (total_elapsed_time % 3600) % 60

    # Print total time with hours, minutes and seconds
    print(f'Total training time: {int(hours)}h {int(minutes)}m {int(seconds)}s')
```

Training LSTM

Hyperparameters definition.

In [13]:
```
vocab_size = len(vocab) # vocabulary size
emb_size = 800 # embedding size
neurons = 512 # the dimension of the feedforward network model, i.e. # of neurons
num_layers = 2 # the number of nn.LSTM layers
dropout_rate = 0.65 # the regularization factor

loss_function = nn.CrossEntropyLoss() # loss function
lr = 0.001 # learning rate
epochs = 50 # epochs
```

Optimizer, Model definition & training

In [14]:
```
# Call the train function
model = LSTMModel(vocab_size, emb_size, neurons, num_layers, dropout_rate)
optimiser = optim.Adam(model.parameters(), lr=lr)
train(model, epochs, optimiser)
```

```
Epoch 0, Batch 0, Loss 10.267952919006348
Epoch 0, Batch 100, Loss 6.834283828735352
Epoch 0, Batch 200, Loss 6.512574672698975
Epoch 0, Batch 300, Loss 6.296056747436523

------------------------------------------------------------
Epoch 0 completed with average loss 6.7500 in 838.35s
------------------------------------------------------------

Epoch 1, Batch 0, Loss 6.237114906311035
Epoch 1, Batch 100, Loss 6.113154411315918
Epoch 1, Batch 200, Loss 5.999763011932373
Epoch 1, Batch 300, Loss 5.8418450355529785

------------------------------------------------------------
Epoch 1 completed with average loss 6.0200 in 842.76s
------------------------------------------------------------

Epoch 2, Batch 0, Loss 5.816137790679932
Epoch 2, Batch 100, Loss 5.7320122718811035
Epoch 2, Batch 200, Loss 5.7492876052856445
Epoch 2, Batch 300, Loss 5.564638137817383

------------------------------------------------------------
Epoch 2 completed with average loss 5.7374 in 862.74s
------------------------------------------------------------

Epoch 3, Batch 0, Loss 5.581632137298584
Epoch 3, Batch 100, Loss 5.557146072387695
Epoch 3, Batch 200, Loss 5.477694511413574
Epoch 3, Batch 300, Loss 5.489266395568848

------------------------------------------------------------
Epoch 3 completed with average loss 5.5353 in 820.89s
------------------------------------------------------------

Epoch 4, Batch 0, Loss 5.338476657867432
Epoch 4, Batch 100, Loss 5.383350372314453
Epoch 4, Batch 200, Loss 5.384471416473389
Epoch 4, Batch 300, Loss 5.3710174560546875

------------------------------------------------------------
Epoch 4 completed with average loss 5.3741 in 806.76s
------------------------------------------------------------

Epoch 5, Batch 0, Loss 5.262752056121826
Epoch 5, Batch 100, Loss 5.25981330871582
Epoch 5, Batch 200, Loss 5.179980278015137
Epoch 5, Batch 300, Loss 5.1806793212890625

------------------------------------------------------------
Epoch 5 completed with average loss 5.2423 in 802.60s
------------------------------------------------------------

Epoch 6, Batch 0, Loss 5.161752700805664
Epoch 6, Batch 100, Loss 5.0628252029418945
Epoch 6, Batch 200, Loss 5.113908290863037
Epoch 6, Batch 300, Loss 5.107614040374756

------------------------------------------------------------
Epoch 6 completed with average loss 5.1287 in 801.80s
------------------------------------------------------------

Epoch 7, Batch 0, Loss 4.99995231628418
Epoch 7, Batch 100, Loss 5.022647380828857
Epoch 7, Batch 200, Loss 4.963942050933838
Epoch 7, Batch 300, Loss 5.053882122039795

------------------------------------------------------------
Epoch 7 completed with average loss 5.0306 in 802.12s
------------------------------------------------------------

Epoch 8, Batch 0, Loss 4.9972405433654785
Epoch 8, Batch 100, Loss 4.8533101081848145
Epoch 8, Batch 200, Loss 5.002138614654541
Epoch 8, Batch 300, Loss 4.90820837020874

------------------------------------------------------------
Epoch 8 completed with average loss 4.9443 in 806.23s
------------------------------------------------------------

Epoch 9, Batch 0, Loss 4.737703800201416
Epoch 9, Batch 100, Loss 4.832128524780273
Epoch 9, Batch 200, Loss 4.884164333343506
Epoch 9, Batch 300, Loss 4.805817127227783

------------------------------------------------------------
Epoch 9 completed with average loss 4.8652 in 801.65s
------------------------------------------------------------

Epoch 10, Batch 0, Loss 4.735334873199463
Epoch 10, Batch 100, Loss 4.825525283813477
Epoch 10, Batch 200, Loss 4.815614223480225
Epoch 10, Batch 300, Loss 4.752321243286133

------------------------------------------------------------
Epoch 10 completed with average loss 4.7948 in 801.77s
------------------------------------------------------------

Epoch 11, Batch 0, Loss 4.685385227203369
Epoch 11, Batch 100, Loss 4.592047214508057
Epoch 11, Batch 200, Loss 4.7573018074035645
Epoch 11, Batch 300, Loss 4.764608383178711

------------------------------------------------------------
Epoch 11 completed with average loss 4.7268 in 802.43s
------------------------------------------------------------

Epoch 12, Batch 0, Loss 4.690743923187256
Epoch 12, Batch 100, Loss 4.6815996170043945
Epoch 12, Batch 200, Loss 4.666337490081787
Epoch 12, Batch 300, Loss 4.731950283050537
```

```
------------------------------------------------------------
Epoch 12 completed with average loss 4.6649 in 801.68s
------------------------------------------------------------

Epoch 13, Batch 0, Loss 4.582937717437744
Epoch 13, Batch 100, Loss 4.565176010131836
Epoch 13, Batch 200, Loss 4.633476734161377
Epoch 13, Batch 300, Loss 4.648616790771484

------------------------------------------------------------
Epoch 13 completed with average loss 4.6074 in 801.88s
------------------------------------------------------------

Epoch 14, Batch 0, Loss 4.569495677947998
Epoch 14, Batch 100, Loss 4.514218807220459
Epoch 14, Batch 200, Loss 4.593440532684326
Epoch 14, Batch 300, Loss 4.619258880615234

------------------------------------------------------------
Epoch 14 completed with average loss 4.5501 in 801.76s
------------------------------------------------------------

Epoch 15, Batch 0, Loss 4.426522731781006
Epoch 15, Batch 100, Loss 4.542745113372803
Epoch 15, Batch 200, Loss 4.467591762542725
Epoch 15, Batch 300, Loss 4.468790054321289

------------------------------------------------------------
Epoch 15 completed with average loss 4.4978 in 801.54s
------------------------------------------------------------

Epoch 16, Batch 0, Loss 4.39603853225708
Epoch 16, Batch 100, Loss 4.459595203399658
Epoch 16, Batch 200, Loss 4.434098720550537
Epoch 16, Batch 300, Loss 4.41222620010376

------------------------------------------------------------
Epoch 16 completed with average loss 4.4467 in 803.02s
------------------------------------------------------------

Epoch 17, Batch 0, Loss 4.277978420257568
Epoch 17, Batch 100, Loss 4.381706714630127
Epoch 17, Batch 200, Loss 4.446342945098877
Epoch 17, Batch 300, Loss 4.519172191619873

------------------------------------------------------------
Epoch 17 completed with average loss 4.3969 in 801.29s
------------------------------------------------------------

Epoch 18, Batch 0, Loss 4.328232765197754
Epoch 18, Batch 100, Loss 4.3408732414245605
Epoch 18, Batch 200, Loss 4.308963298797607
Epoch 18, Batch 300, Loss 4.388861656188965

------------------------------------------------------------
```

```
Epoch 18 completed with average loss 4.3506 in 802.73s
------------------------------------------------------------

Epoch 19, Batch 0, Loss 4.2198686599731445
Epoch 19, Batch 100, Loss 4.278253078460693
Epoch 19, Batch 200, Loss 4.325565338134766
Epoch 19, Batch 300, Loss 4.434560298919678

------------------------------------------------------------
Epoch 19 completed with average loss 4.3052 in 801.49s
------------------------------------------------------------

Epoch 20, Batch 0, Loss 4.153954029083252
Epoch 20, Batch 100, Loss 4.1899333000183105
Epoch 20, Batch 200, Loss 4.23619270324707
Epoch 20, Batch 300, Loss 4.235925674438477

------------------------------------------------------------
Epoch 20 completed with average loss 4.2602 in 801.93s
------------------------------------------------------------

Epoch 21, Batch 0, Loss 4.25771951675415
Epoch 21, Batch 100, Loss 4.229190826416016
Epoch 21, Batch 200, Loss 4.166518688201904
Epoch 21, Batch 300, Loss 4.2935404777526855

------------------------------------------------------------
Epoch 21 completed with average loss 4.2173 in 802.73s
------------------------------------------------------------

Epoch 22, Batch 0, Loss 4.101994037628174
Epoch 22, Batch 100, Loss 4.192526340484619
Epoch 22, Batch 200, Loss 4.2172346115112305
Epoch 22, Batch 300, Loss 4.196609973907471

------------------------------------------------------------
Epoch 22 completed with average loss 4.1762 in 801.80s
------------------------------------------------------------

Epoch 23, Batch 0, Loss 4.022215366363525
Epoch 23, Batch 100, Loss 4.1735615730285645
Epoch 23, Batch 200, Loss 4.181331157684326
Epoch 23, Batch 300, Loss 4.131230354309082

------------------------------------------------------------
Epoch 23 completed with average loss 4.1342 in 801.96s
------------------------------------------------------------

Epoch 24, Batch 0, Loss 3.9640164375305176
Epoch 24, Batch 100, Loss 4.088408946990967
Epoch 24, Batch 200, Loss 4.200056076049805
Epoch 24, Batch 300, Loss 4.156118392944336

------------------------------------------------------------
Epoch 24 completed with average loss 4.0960 in 801.26s
------------------------------------------------------------
```

```
Epoch 25, Batch 0, Loss 3.9424755573272705
Epoch 25, Batch 100, Loss 4.053533554077148
Epoch 25, Batch 200, Loss 4.033069133758545
Epoch 25, Batch 300, Loss 4.1308674812316895

------------------------------------------------------------
Epoch 25 completed with average loss 4.0578 in 802.33s
------------------------------------------------------------

Epoch 26, Batch 0, Loss 3.940448522567749
Epoch 26, Batch 100, Loss 3.9569079875946045
Epoch 26, Batch 200, Loss 3.9917783737182617
Epoch 26, Batch 300, Loss 4.041839122772217

------------------------------------------------------------
Epoch 26 completed with average loss 4.0212 in 802.05s
------------------------------------------------------------

Epoch 27, Batch 0, Loss 4.009219169616699
Epoch 27, Batch 100, Loss 4.043494701385498
Epoch 27, Batch 200, Loss 3.9369277954101562
Epoch 27, Batch 300, Loss 4.0703020095825195

------------------------------------------------------------
Epoch 27 completed with average loss 3.9848 in 802.79s
------------------------------------------------------------

Epoch 28, Batch 0, Loss 3.8649916648864746
Epoch 28, Batch 100, Loss 3.8571088314056396
Epoch 28, Batch 200, Loss 3.889878511428833
Epoch 28, Batch 300, Loss 3.91973876953125

------------------------------------------------------------
Epoch 28 completed with average loss 3.9503 in 802.23s
------------------------------------------------------------

Epoch 29, Batch 0, Loss 3.8586084842681885
Epoch 29, Batch 100, Loss 3.934353828430176
Epoch 29, Batch 200, Loss 4.0065836906433105
Epoch 29, Batch 300, Loss 4.006514072418213

------------------------------------------------------------
Epoch 29 completed with average loss 3.9176 in 802.35s
------------------------------------------------------------

Epoch 30, Batch 0, Loss 3.889730215072632
Epoch 30, Batch 100, Loss 3.903712749481201
Epoch 30, Batch 200, Loss 3.9084503650665283
Epoch 30, Batch 300, Loss 3.945368766784668

------------------------------------------------------------
Epoch 30 completed with average loss 3.8861 in 802.39s
------------------------------------------------------------

Epoch 31, Batch 0, Loss 3.796024799346924
Epoch 31, Batch 100, Loss 3.879132032394409
Epoch 31, Batch 200, Loss 3.9028306007385254
Epoch 31, Batch 300, Loss 3.911132574081421

------------------------------------------------------------
Epoch 31 completed with average loss 3.8536 in 801.32s
------------------------------------------------------------

Epoch 32, Batch 0, Loss 3.7424488067626953
Epoch 32, Batch 100, Loss 3.8054208755493164
Epoch 32, Batch 200, Loss 3.8263790607452393
Epoch 32, Batch 300, Loss 3.845512628555298

------------------------------------------------------------
Epoch 32 completed with average loss 3.8217 in 801.93s
------------------------------------------------------------

Epoch 33, Batch 0, Loss 3.7514727115631104
Epoch 33, Batch 100, Loss 3.8061022758483887
Epoch 33, Batch 200, Loss 3.736781597137451
Epoch 33, Batch 300, Loss 3.8150887489318848

------------------------------------------------------------
Epoch 33 completed with average loss 3.7936 in 801.93s
------------------------------------------------------------

Epoch 34, Batch 0, Loss 3.6780450344085693
Epoch 34, Batch 100, Loss 3.703847646713257
Epoch 34, Batch 200, Loss 3.759312629699707
Epoch 34, Batch 300, Loss 3.7719740867614746

------------------------------------------------------------
Epoch 34 completed with average loss 3.7634 in 801.23s
------------------------------------------------------------

Epoch 35, Batch 0, Loss 3.658438205718994
Epoch 35, Batch 100, Loss 3.703782081604004
Epoch 35, Batch 200, Loss 3.670518159866333
Epoch 35, Batch 300, Loss 3.7931385040283203

------------------------------------------------------------
Epoch 35 completed with average loss 3.7355 in 824.86s
------------------------------------------------------------

Epoch 36, Batch 0, Loss 3.649303913116455
Epoch 36, Batch 100, Loss 3.639050006866455
Epoch 36, Batch 200, Loss 3.754610538482666
Epoch 36, Batch 300, Loss 3.7921905517578125

------------------------------------------------------------
Epoch 36 completed with average loss 3.7087 in 874.41s
------------------------------------------------------------

Epoch 37, Batch 0, Loss 3.5893630981445312
Epoch 37, Batch 100, Loss 3.687694787979126
Epoch 37, Batch 200, Loss 3.6619317531585693
Epoch 37, Batch 300, Loss 3.6857733726501465

------------------------------------------------------------
Epoch 37 completed with average loss 3.6824 in 872.09s
------------------------------------------------------------

Epoch 38, Batch 0, Loss 3.641207695007324
Epoch 38, Batch 100, Loss 3.6234164237976074
Epoch 38, Batch 200, Loss 3.728297233581543
Epoch 38, Batch 300, Loss 3.6995999813079834

------------------------------------------------------------
Epoch 38 completed with average loss 3.6551 in 871.82s
------------------------------------------------------------

Epoch 39, Batch 0, Loss 3.584510087966919
Epoch 39, Batch 100, Loss 3.7361221313476562
Epoch 39, Batch 200, Loss 3.6356899738311768
Epoch 39, Batch 300, Loss 3.6664485931396484

------------------------------------------------------------
Epoch 39 completed with average loss 3.6325 in 4642.96s
------------------------------------------------------------

Epoch 40, Batch 0, Loss 3.5742011070251465
Epoch 40, Batch 100, Loss 3.570418119430542
Epoch 40, Batch 200, Loss 3.6597607135772705
Epoch 40, Batch 300, Loss 3.684755563735962

------------------------------------------------------------
Epoch 40 completed with average loss 3.6058 in 863.65s
------------------------------------------------------------

Epoch 41, Batch 0, Loss 3.4757680892944336
Epoch 41, Batch 100, Loss 3.449211359024048
Epoch 41, Batch 200, Loss 3.6067256927490234
Epoch 41, Batch 300, Loss 3.69199275970459

------------------------------------------------------------
Epoch 41 completed with average loss 3.5829 in 863.91s
------------------------------------------------------------

Epoch 42, Batch 0, Loss 3.4456000328063965
Epoch 42, Batch 100, Loss 3.5996456146240234
Epoch 42, Batch 200, Loss 3.6827785968780518
Epoch 42, Batch 300, Loss 3.5827133655548096

------------------------------------------------------------
Epoch 42 completed with average loss 3.5613 in 865.51s
------------------------------------------------------------

Epoch 43, Batch 0, Loss 3.4760053157806396
Epoch 43, Batch 100, Loss 3.490980625152588
Epoch 43, Batch 200, Loss 3.5125222206115723
Epoch 43, Batch 300, Loss 3.633470058441162
```

```
------------------------------------------------------------
Epoch 43 completed with average loss 3.5395 in 864.27s
------------------------------------------------------------

Epoch 44, Batch 0, Loss 3.424654245376587
Epoch 44, Batch 100, Loss 3.464897394180298
Epoch 44, Batch 200, Loss 3.5410289764404297
Epoch 44, Batch 300, Loss 3.525073289871216

------------------------------------------------------------
Epoch 44 completed with average loss 3.5151 in 864.28s
------------------------------------------------------------

Epoch 45, Batch 0, Loss 3.4438858032226562
Epoch 45, Batch 100, Loss 3.488055944442749
Epoch 45, Batch 200, Loss 3.5350828170776367
Epoch 45, Batch 300, Loss 3.5552256107330322

------------------------------------------------------------
Epoch 45 completed with average loss 3.4941 in 864.46s
------------------------------------------------------------

Epoch 46, Batch 0, Loss 3.396148920059204
Epoch 46, Batch 100, Loss 3.5242137908935547
Epoch 46, Batch 200, Loss 3.421900987625122
Epoch 46, Batch 300, Loss 3.4529242515563965

------------------------------------------------------------
Epoch 46 completed with average loss 3.4750 in 867.65s
------------------------------------------------------------

Epoch 47, Batch 0, Loss 3.391603469848633
Epoch 47, Batch 100, Loss 3.4138152599334717
Epoch 47, Batch 200, Loss 3.4773247241973877
Epoch 47, Batch 300, Loss 3.473972797393799

------------------------------------------------------------
Epoch 47 completed with average loss 3.4530 in 863.89s
------------------------------------------------------------

Epoch 48, Batch 0, Loss 3.3335628509521484
Epoch 48, Batch 100, Loss 3.4155097007751465
Epoch 48, Batch 200, Loss 3.4925928115844727
Epoch 48, Batch 300, Loss 3.56992244720459

------------------------------------------------------------
Epoch 48 completed with average loss 3.4345 in 865.67s
------------------------------------------------------------

Epoch 49, Batch 0, Loss 3.2505855560302734
Epoch 49, Batch 100, Loss 3.366615056991577
Epoch 49, Batch 200, Loss 3.3907852172851562
Epoch 49, Batch 300, Loss 3.4022743701934814

------------------------------------------------------------
Epoch 49 completed with average loss 3.4144 in 864.08s
```

```
------------------------------------------------------------

Total training time: 12h 29m 29s
```

Text Generation

In [15]:
```
def generate_text(model, start_text, num_words, temperature=1.0):
    model.eval()

    # Tokenize the starting text and initialize the hidden state
    words = tokeniser(start_text)
    hidden = model.init_hidden(1)

    # Generate words one by one
    for _ in range(num_words):
        # Input tensor preparation
        input_indices = [vocab[word] for word in words[-1:]]  # get indices for the
        x = torch.tensor([input_indices], dtype=torch.long, device=device)

        # Get predictions from model
        y_pred, hidden = model(x, hidden)

        # Vector of raw prediction score (logit), apply softmax with temperature
        last_word_logits = y_pred[0][-1]
        probabilities = F.softmax(last_word_logits / temperature, dim=0).detach().c

        # Sample a word index from the probability distribution, append the generat
        word_index = np.random.choice(len(vocab), p=probabilities)
        words.append(vocab.lookup_token(word_index))

    return ' '.join(words)
```

Text generations samples

Example 1

In [16]:
```
print(generate_text(model, start_text="I like", num_words=100))
```

```
i like lemon - 76 in birmingham . in the 1950s , thomas observed billy the new demon
called <unk> and <unk> love america ' s final direction for best story , to celebrat
e the devin townsend album , when i moved back on dangerously in love in the symphon
y for fame . the manga ' s album video performances = = = god of war iii film revolu
tion ( father @-@ green , love ) and john ( author william ii [ god ] having a major
<unk> ) to be regarded as the seventh to cultural influences . in the
```

Example 2

In [17]:
```
print(generate_text(model, start_text="A cat", num_words=125))
```

```
a cat in the midst of a buddhist shadow , with much meat and desired to direct seein
g whatever this offensive is in the history of king ' s war . this she was forced to
operate in that world . he said that the king was serious on creating my own rivals
( when he made the developing way ) , <unk> it was more careful to this concert rose
bery believes to become the person ' s challenging . despite eva perón ' s own tenur
e , a believe of a criminal family that renewed our imprisonment de <unk> <unk> his
things to build within a formula . edmund claims that applewhite and nettles conscio
usly learn , , the humans , said , and substance (
```

Example 3

In [18]:
```python
print(generate_text(model, start_text="She loves you because", num_words=25))
```

```
she loves you because the wing ' s pattern were handled . a large tertiary vessel wa
s discovered by group japanese commanders in the united states , where the
```

Additional examples making some variations with starting text and numer of words

In [19]:
```python
print(generate_text(model, start_text="I hope", num_words=50))
```

```
i hope , marking the first time to do there . he and <unk> did not figured on odaena
thus ' s name while calvert did not <unk> . <unk> in <unk> , the supreme court of sp
ain did ' s in keeping him as the 22nd century when they began to pursue
```

In [20]:
```python
print(generate_text(model, start_text="The Beatles are", num_words=250))
```

```
the beatles are able to obtain <unk> as complementary items have been increased . th
e same list for such <unk> <unk> has not been described as give little social <unk>
, this may have been <unk> between the help of a medium and nearest scientific figur
e . while by inari ii under conservation , latin @-@ americans had a group of differ
ent manifestation of it from blood @-@ material if <unk> . after finding more long @
-@ derived reactions that specific objects that done in the path , goddesses believe
d that amun was the most regions of his lives , and the reason of the eighteenth jai
n @-@ speaking processes . the gods lived in other parts of the history of djedkare
and isis , especially in such children , was only an important family of their issue
s . these objects were found in <unk> , combined with many other piedras phenomena ,
mostly or visitors to adapted for their myths . the eshmun statue was different , an
d they emerged in the most stone shiva on toniná . the sculpture begins by osiris ,
from <unk> , and a south section of an ancient temple held at the manor of maharasht
ra , a tiger , shows this a long mark to pure a <unk> <unk> . colonel <unk> <unk> a
<unk> , which equated with shiva by north @-@ eastern kings with the animal , suppor
ted by the wives ' s and personal deities . in magical texts , their only mastery
```

In [21]:
```python
print(generate_text(model, start_text="Super Mario is", num_words=175))
```

```
super mario is given the cap of the court around two @-@ thirds of shiva and his lef
t men in this desert . the shape of the facade is given the added walls in the town
, which is called <unk> ( the right <unk> ) and and the <unk> @-@ long ( two @-@ squ
are layer ) , at the inscription ( <unk> ) <unk> and the mosque , which contains a p
lantain meaning being ordered by the capitals . as the mirror is formed around the s
culpture of one <unk> , the piers supported the inscription at bath as a small lady
holds entrance to the east . each tower has parallel the window for the sky in the c
aves , which are built over the aisle . fig . 9 is number 66 . <unk> in australia co
ntain surrounded by two short @-@ style organs while simple classic . in addition to
each tower ( fig . 16 ) , m @-@ 44 meant the immediately original structure of the <
unk> ' building road .
```

Conclusion

In the initial LSTM class, we incorporated weight initialization to accelerate the model's convergence during training and to potentially enhance overall performance. The following methods were implemented:

- Xavier/Glorot initialization for input-hidden weights.
- Orthogonal initialization for LSTM hidden-hidden weights.
- Zero initialization for biases.

When comparing this LSTM, designed for Text Generation, with an RNN used for classification tasks, we observed notable differences. The RNN has a simpler structure focused primarily on prediction, where its performance is evaluated based on accuracy using a cross-entropy loss function. On the other hand, the LSTM, a specialized type of RNN architecture, is more complex. It features memory cells that regulate the flow of information. Unlike the RNN, LSTMs are trained to predict the next token in a sequence based on previous tokens, necessitating the maintenance of information over many time steps. In this case, we computed the cross-entropy loss function. However, for a more accurate evaluation of model performance, it would be appropriate to calculate perplexity.

We found this interesting page that very didactically explain the perplexity concept: https://lukesalamone.github.io/posts/perplexity/

$$\text{perplexity} = e^z$$

where

$$z = -\frac{1}{N}\sum_{i=0}^{N} \ln(P_n)$$

Hopefully in the future we can implement an improved model in which we can calculate perplexity.

About the temperature, we asked to our dear professor in the class and looks like it is effectively related with Softmax adding a variable theta that affects the softmax distribution, as professor mentioned it can be interpreted of how much entropy or noise you want to have to the output, the more it ism the more "creative", we have now the next question, does this parameter could affect the hallucination of the model?

$$\sigma(z_i) = \frac{e^{z_i\theta}}{\sum_{j=0}^{N} e^{z_j\theta}}$$

https://lukesalamone.github.io/posts/what-is-temperature/

For the weight initialization and hyperparameters selection we based as well on this Medium article: https://towardsdatascience.com/language-modeling-with-lstms-in-pytorch-381a26badcbf, for weight initialization they recommend to checked out this paper that has some studies to select different learning rates for regularization and optimization of LSTM models: https://arxiv.org/abs/1708.02182.

About the results of the generated text we tried to improve the hyperparameters by increasing the embedding size, neurons and epochs. In the first trial took us like 3 hours to have the model training but in the lasta attempt where we increase the embedding size the computing time creases by four and we didn't noticed relevant chances in the generated texts, we think that is due that this model is quite simple comparing with LLMs like GPT or BERT that incorporate transformers and attention mechanisms to deal with the context. As well, we trained with a very basic and modest setup and a relatively small dataset, the other models are trained with a huge quantity of data that used more demanding computational power. This excersice was very didactic, even we though a little bit dissapointing that we couln't improve over model quality of text generation was a good excersise to start getting familiarized with text generation and LLM models.