# Processo Seletivo Aluno Especial IA-024 1S2024 FEEC-UNICAMP

## Aluno: Fabio Grassiotto

## RA: 890441

#### **Respostas das Questões**

#### Seção I

##### I.1. Na célula de calcular o vocabulário, aproveite o laço sobre IMDB de treinamento e utilize um segundo contador para calcular o número de amostras positivas e amostras negativas. Calcule também o comprimento médio do texto em número de palavras dos textos das amostras.

Código implementado na seção I acima. Seguem os resultados obtidos:

Amostras positivas, negativas e totais: Counter({'total': 25000, 'pos': 12500, 'neg': 12500})

Comprimento médio do texto em palavras 270.68748

##### I.2 Mostre as cinco palavras mais frequentes do vocabulário e as cinco palavras menos frequentes.

(Utilizando o Tokenizador)

Cinco palavras mais frequentes: ['the', '.', ',', 'and', 'a']

Cinco palavras menos frequentes: ['voicing', 'hazard', 'lynda', 'gft', 'watergate']

##### Qual é o código do token que está sendo utilizado quando a palavra não está no vocabulário?

Na função de dicionário dict.get() o segundo parâmetro indica o valor default caso a palavra não seja encontrada no dicionário. Nesse caso o código do token usado é o número zero.

##### Número de tokens que não estão no vocabulário na base de treinamento:

174226

#### I.3.a) Qual é a razão pela qual o modelo preditivo conseguiu acertar 100% das amostras de teste do dataset selecionado com apenas as primeiras 200 amostras?

Ao reduzirmos a base de treinamento para apenas 200 amostras, a base se tornou totalmente desbalanceada. Como pudemos verificar, temos 200 amostras classificadas como negativas e nenhuma como positiva. Portanto a taxa de acurácia calculada sobre a classificação da base de testes depende unicamente da percentagem de amostras positivas ou negativas nesta base.

#### I.3.b) Modifique a forma de selecionar 200 amostras do dataset, porém garantindo que ele continue balanceado, isto é, aproximadamente 100 amostras positivas e 100 amostras negativas.

Para obtermos um dataset balanceado, usaremos uma função que seleciona amostras do dataset de acordo com a classificação e cria um dataset com a quantidade de amostras de cada classificação desejada conforme abaixo.

#### Seção II

##### II.1.a) Investigue o dataset criado na linha 24. Faça um código que aplique um laço sobre o dataset train\_data e calcule novamente quantas amostras positivas e negativas do dataset.

Seção do código implementado:

counter\_lbl = Counter({"pos": 0, "neg": 0, "total": 0})  
words\_encoded = 0  
for (oneHot, sentiment) in train\_data:  
  
 words = oneHot.tolist()  
 label = sentiment.item()  
  
 # Número de amostras positivas e negativas  
 if (label == 1):  
 counter\_lbl['neg'] += 1  
 else:  
 counter\_lbl['pos'] += 1  
 counter\_lbl['total'] += 1  
  
 hot\_encoded = sum(words[i] for i in range(len(words)) if words[i] != 0)  
 words\_encoded += hot\_encoded  
  
avg\_words\_enc = words\_encoded / counter\_lbl['total']

##### II.1.b) Calcule também o número médio de palavras codificadas em cada vetor one-hot.

Quantidade média de palavras codificadas em cada vetor one-hot 139.59268

#### Compare este valor com o comprimento médio de cada texto (contado em palavras), conforme calculado no exercício I.1.c. e explique a diferença.

No exercício I.1.c, o comprimento médio do texto em palavras depois de passar pelo tokenizador foi de cerca de 270 palavras. Essa diferença do vetor One-Hot se deve ao fato que o vetor one-hot só codifica as palavras que foram identificadas no dicionário, enquanto que o comprimento médio considera todas as palavras das sentenças. Ou seja, palavras que não foram codificadas no dicionário serão representadas por zeros.

##### II.2.a) Medição dos tempos de loop

Notamos que o tempo do passo do forward leva mais tempo que o passo de backward, conforme os dados obtidos abaixo para a primeira época do treinamento. Também notamos que a maior parte to tempo do loop de forward é gasto com a transferência dos dados da CPU para a GPU (97% no primeiro loop).

Loop # 1  
Tempo de loop = 0.048320770263671875  
Forward pass = 0.047322750091552734  
Gpu copy = 97.88851606662435 %  
Model processing = 2.1114839333756574 %  
Backward pass = 0.0009980201721191406  
  
Loop # 2  
Tempo de loop = 0.007141590118408203  
Forward pass = 0.005140781402587891  
Gpu copy = 80.50737408403673 %  
Model processing = 19.49262591596327 %  
Backward pass = 0.0020008087158203125

##### II.2.b) Trecho que precisa ser otimizado. (Esse é um problema mais difícil)

Para otimizarmos o loop, o carregamento dos dados em GPU pode ser realizado pelo Dataloader fora do loop de treinamento, para tanto alterando o método **init**() da classe IMDBDataset.

def \_\_init\_\_(self, split, vocab):  
 #self.data = list(IMDB(split=split))[:n\_samples]  
 self.data = list(balanced\_dataset(IMDB(split=split), n\_samples))   
 self.vocab = vocab

##### II.2.c) Otimize o código e explique aqui.

Substituimos então com a nova implementação, onde o dataset inteiro é pré-processado, codificado em forma One-Hot (uma vez que tensores não suportam strings) e movido para a GPU antes do processo de treinamento:

def \_\_init\_\_(self, split, vocab):  
   
 # II.2.b) Trecho que precisa ser otimizado. (Esse é um problema mais difícil)  
 self.data = list(balanced\_dataset(IMDB(split='train'), n\_samples))  
  
 if preload\_to\_gpu:   
 labels = [x[0] for x in self.data]  
 lines = [x[1] for x in self.data]  
  
 # One-Hot Encoding  
 self.labels\_enc = []  
 for l in labels:  
 l = 1 if l == 1 else 0  
 self.labels\_enc.append(l)  
 self.labels\_enc = torch.tensor(self.labels\_enc)  
 self.labels\_enc = self.labels\_enc.to(device)  
  
 self.lines\_enc = []  
 for l in lines:  
 X = torch.zeros(len(vocab) + 1)  
 for word in encode\_sentence(l, vocab):  
 X[word] = 1  
 self.lines\_enc.append(X)  
 self.lines\_enc = [tensor.to(device) for tensor in self.lines\_enc]  
  
 self.vocab = vocab

##### Comparação do tempo de treinamento com a otimização (GPU RTX2060 local):

Sem pre-load em GPU:

Epoch [1/5], Loss: 0.6911, Elapsed Time: 61.36 sec  
Epoch [2/5], Loss: 0.6929, Elapsed Time: 58.69 sec  
Epoch [3/5], Loss: 0.6984, Elapsed Time: 58.95 sec  
Epoch [4/5], Loss: 0.6792, Elapsed Time: 58.60 sec  
Epoch [5/5], Loss: 0.6874, Elapsed Time: 58.59 sec

Com pre-load em GPU (RTX2060)

Epoch [1/5], Loss: 0.6896, Elapsed Time: 3.81 sec  
Epoch [2/5], Loss: 0.6925, Elapsed Time: 0.58 sec  
Epoch [3/5], Loss: 0.6933, Elapsed Time: 0.64 sec  
Epoch [4/5], Loss: 0.6890, Elapsed Time: 0.58 sec  
Epoch [5/5], Loss: 0.6904, Elapsed Time: 0.57 sec

Notamos, no entanto, que o uso de mémória na GPU se torna muito maior, conforme pode ser visualizado abaixo (5Gb/6Gb total):

[venv:ml] $ nvidia-smi  
Mon Feb 12 08:23:42 2024  
+-----------------------------------------------------------------------------+  
| NVIDIA-SMI 516.94 Driver Version: 516.94 CUDA Version: 11.7 |  
|-------------------------------+----------------------+----------------------+  
| 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 NVIDIA GeForce ... WDDM | 00000000:01:00.0 On | N/A |  
| N/A 76C P8 12W / N/A | 5035MiB / 6144MiB | 1% Default |  
| | | N/A |  
+-------------------------------+----------------------+----------------------+

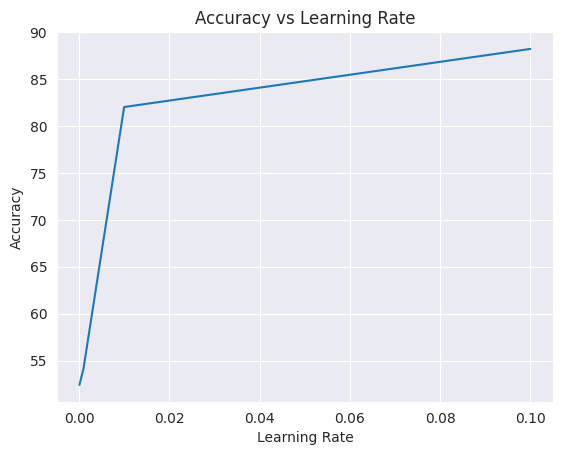
##### II.3 Faça a melhor escolha do LR, analisando o valor da acurácia no conjunto de teste, utilizando para cada valor de LR, a acurácia obtida. Faça um gráfico de Acurácia vs LR e escolha o LR que forneça a maior acurácia possível.

lr\_list = [0.0001, 0.001, 0.01, 0.1]  
acc\_list = []  
  
for lr in lr\_list:  
 print("LR = ", lr)  
 model = OneHotMLP(vocab\_size) # to reset weights  
 train\_mdl(model, lr)  
 acc\_list.append(eval\_mdl(model))  
 print()  
  
print(lr\_list)  
print(acc\_list)  
print()

LR = 0.0001  
Epoch [1/5], Loss: 0.6939, Elapsed Time: 0.37 sec, Loader Iterations: 196, Spls lst batch: 40 , R avg: 0.5000   
Epoch [2/5], Loss: 0.6937, Elapsed Time: 0.46 sec, Loader Iterations: 196, Spls lst batch: 40 , R avg: 0.5003   
Epoch [3/5], Loss: 0.6941, Elapsed Time: 0.47 sec, Loader Iterations: 196, Spls lst batch: 40 , R avg: 0.5002   
Epoch [4/5], Loss: 0.6942, Elapsed Time: 0.49 sec, Loader Iterations: 196, Spls lst batch: 40 , R avg: 0.5003   
Epoch [5/5], Loss: 0.6954, Elapsed Time: 0.51 sec, Loader Iterations: 196, Spls lst batch: 40 , R avg: 0.5002   
Test Accuracy: 52.412%  
  
LR = 0.001  
Epoch [1/5], Loss: 0.6956, Elapsed Time: 0.37 sec, Loader Iterations: 196, Spls lst batch: 40 , R avg: 0.4998   
Epoch [2/5], Loss: 0.6914, Elapsed Time: 0.37 sec, Loader Iterations: 196, Spls lst batch: 40 , R avg: 0.5000   
Epoch [3/5], Loss: 0.6853, Elapsed Time: 0.37 sec, Loader Iterations: 196, Spls lst batch: 40 , R avg: 0.5003   
Epoch [4/5], Loss: 0.6860, Elapsed Time: 0.36 sec, Loader Iterations: 196, Spls lst batch: 40 , R avg: 0.4999   
Epoch [5/5], Loss: 0.7005, Elapsed Time: 0.37 sec, Loader Iterations: 196, Spls lst batch: 40 , R avg: 0.4990   
Test Accuracy: 54.152%  
  
LR = 0.01  
Epoch [1/5], Loss: 0.6765, Elapsed Time: 0.36 sec, Loader Iterations: 196, Spls lst batch: 40 , R avg: 0.5000   
Epoch [2/5], Loss: 0.6476, Elapsed Time: 0.37 sec, Loader Iterations: 196, Spls lst batch: 40 , R avg: 0.4998   
Epoch [3/5], Loss: 0.5837, Elapsed Time: 0.37 sec, Loader Iterations: 196, Spls lst batch: 40 , R avg: 0.5001   
Epoch [4/5], Loss: 0.4483, Elapsed Time: 0.37 sec, Loader Iterations: 196, Spls lst batch: 40 , R avg: 0.5000   
Epoch [5/5], Loss: 0.5591, Elapsed Time: 0.37 sec, Loader Iterations: 196, Spls lst batch: 40 , R avg: 0.5001   
Test Accuracy: 82.02%  
  
LR = 0.1  
Epoch [1/5], Loss: 0.3548, Elapsed Time: 0.37 sec, Loader Iterations: 196, Spls lst batch: 40 , R avg: 0.5001   
Epoch [2/5], Loss: 0.2222, Elapsed Time: 0.36 sec, Loader Iterations: 196, Spls lst batch: 40 , R avg: 0.5007   
Epoch [3/5], Loss: 0.2340, Elapsed Time: 0.36 sec, Loader Iterations: 196, Spls lst batch: 40 , R avg: 0.5002   
Epoch [4/5], Loss: 0.3581, Elapsed Time: 0.37 sec, Loader Iterations: 196, Spls lst batch: 40 , R avg: 0.5002   
Epoch [5/5], Loss: 0.3277, Elapsed Time: 0.37 sec, Loader Iterations: 196, Spls lst batch: 40 , R avg: 0.5004   
Test Accuracy: 88.212%  
  
[0.0001, 0.001, 0.01, 0.1]  
[52.412, 54.152, 82.02, 88.212]

##### II.3.a) Gráfico Acurácia vs LR

import seaborn as sns  
import matplotlib.pyplot as plt  
  
sns.set\_style("darkgrid")  
sns.lineplot(x=lr\_list, y=acc\_list)  
  
# Add labels and title  
plt.xlabel("Learning Rate")  
plt.ylabel("Accuracy")  
plt.title("Accuracy vs Learning Rate")  
  
# Show the plot  
plt.show()



##### II.3.b) Valor ótimo do LR

Notamos que o valor ótimo para a Learning Rate foi de cerca de 0.1, com crescimento exponencial ao aumentá-la. Valores acima deste são grandes demais e não levam à otimização do modelo.

##### II.3.c) Mostre a equação utilizada no gradiente descendente e qual é o papel do LR no ajuste dos parâmetros (weights) do modelo da rede neural.

No processo de otimização de uma função, a fórmula utilizada para a estimativa do próximo valor da função é dada por:

valor atualizado = valor anterior - learning rate\*gradiente

Portanto o papel da LR é definir qual é o *tamanho* do passo a ser utilizado no processo de atualização.

##### II.4 Melhores a forma de tokenizar, isto é, pré-processar o dataset de modo que a codificação seja indiferente das palavras serem escritas com maiúsculas ou minúsculas e sejam pouco influenciadas pelas pontuações.

##### II.4.a) Mostre os trechos modificados para este novo tokenizador, tanto na seção I - Vocabulário, como na seção II - Dataset.

Na seção I - Vocabulário:

from torchtext.data import get\_tokenizer  
  
for (label, line) in list(IMDB(split='train'))[:n\_samples]:  
 if (use\_tokenizer):  
 tokenizer = get\_tokenizer('basic\_english')  
 # tokenize the sentence  
 line = tokenizer(line)  
 counter.update(line.split())  
  
 # Número de amostras positivas e negativas  
 if (label == 1):  
 counter\_lbl['neg'] += 1  
 else:  
 counter\_lbl['pos'] += 1  
 counter\_lbl['total'] += 1  
  
 # Comprimento médio do texto das reviews em palavras  
 tokenizer = get\_tokenizer('basic\_english')  
  
 # tokenize the sentence  
 tokens = tokenizer(line)  
  
 # count the number of words  
 total\_review\_len += len(tokens)

Na Seção II - Dataset: São apenas necessárias alterações no encoder da sentença, conforme abaixo.

def encode\_sentence(sentence, vocab, use\_tokenizer):  
 if (use\_tokenizer):  
 sentence = tokenizer(sentence)  
 return [vocab.get(word, 0) for word in sentence]  
 else:  
 return [vocab.get(word, 0) for word in sentence.split()] # 0 for OOV

##### II.4.b) Recalcule novamente os valores do exercício I.2.c - número de tokens unknown, e apresente uma tabela comparando os novos valores com os valores obtidos com o tokenizador original e justifique os resultados obtidos.

Sem o tokenizador:

566141

Com o tokenizador:

174226

Estes valores se justificam pelo fato que o tokenizador altera as palavras das sentenças, mantendo apenas radicais, de forma que menos palavras não serão encontradas na base do vocabulário.

##### II.4.c) Execute agora no notebook inteiro com o novo tokenizador e veja o novo valor da acurácia obtido com a melhoria do tokenizador.

Sem o tokenizador:

Test Accuracy: 73.45% (Para LR = 0.1)

Com o tokenizador

Test Accuracy: 88.47% (Para LR = 0.1)

O aumento da acurácia é justificado pelo fato que menos palavras de cada sentença não serão reconhecidas (OneHot encoding não terá tantos valores zerados)

##### Os dados obtidos estão resumidos na tabela abaixo.

from tabulate import tabulate  
  
# Sample data  
data = [  
 ['Sem Tokenizador', 566141, '73.45%'],  
 ['Com Tokenizador', 174226, '88.47%'],  
]  
  
# Headers  
headers = ['Uso do Tokenizador', 'Tokens Unknown', 'Test Accuracy']  
  
# Print the table  
print(tabulate(data, headers=headers))

Uso do Tokenizador Tokens Unknown Test Accuracy  
-------------------- ---------------- ---------------  
Sem Tokenizador 566141 73.45%  
Com Tokenizador 174226 88.47%

#### Seção III

##### Vamos estudar agora o Data Loader da seção III do notebook. Em primeiro lugar anote a acurácia do notebook com as melhorias de eficiência de rodar em GPU, com ajustes de LR e do tokenizador. Em seguida mude o parâmetro shuffle na construção do objeto train\_loader para False e execute novamente o notebook por completo e meça novamente a acurácia.

from tabulate import tabulate  
  
# Sample data  
data = [  
 ['Com Shuffle', '88.47%'],  
 ['Sem Shuffle', '50.00%']  
]  
  
# Headers  
headers = ['Shuffle dos dados de Treinamento', 'Test Accuracy']  
  
# Print the table  
print(tabulate(data, headers=headers))

Shuffle dos dados de Treinamento Test Accuracy  
---------------------------------- ---------------  
Com Shuffle 88.47%  
Sem Shuffle 50.00%

##### III.1.a) Explique as duas principais vantagens do uso de batch no treinamento de redes neurais.

O uso de lotes em treinamento é importante por causa do aumento da eficiência computacional e para aumentar a estabilidade do gradiente. A eficiência computacional é aumentada pois com lotes maiores a paralelização do processamento em GPUs é mais bem aproveitada, enquanto que a estabilidade do gradiente é aumentada durante o treinamento pois em cada iteração, o gradiente é calculado com base na função de perda para o lote inteiro reduzindo a variabilidade do gradiente em comparação com o cálculo individual para cada exemplo.

##### III.1.b) Explique por que é importante fazer o embaralhamento das amostras do batch em cada nova época.

O embarelhamento das amostras de batch do treinamento é essencial para aumentar a generabilidade do modelo. As razões para tanto são:

* Redução do viés das amostras ordenadas do início ao fim do dataset.
* Estabilização do gradiente (redução da oscilação causada por amostras ordenadas).
* Melhoria da convergência, pois amostras agrupadas de uma classe dificultam o processo de aprendizado da rede neural.

Em geral o embaralhamento de amostras de treinamento é um processo usual para a redução da generalização e a obtenção de um modelo de melhores características.

##### III.1.c) Se você alterar o shuffle=False no instanciamento do objeto test\_loader, por que o cálculo da acurácia não se altera?

A acurácia não se altera pois em tempo de inferência (ou seja, fase de teste) os pesos do modelo não são alterados mais. Portanto, a base de testes é usada apenas para verificar a capacidade de generalização do modelo.

##### III.2.a) Faça um laço no objeto train\_loader e meça quantas iterações o Loader tem. Mostre o código para calcular essas iterações. Explique o valor encontrado.

Modificações no código de treinamento (função train\_mdl()) acima:

for epoch in range(num\_epochs):  
 start\_time = time.time() # Start time of the epoch  
 model.train()  
  
 loop\_count = 0  
  
 train\_loader\_iterations = 0   
 for inputs, labels in train\_loader:  
 train\_loader\_iterations += 1

Número de interações por época:

Epoch [1/5], Loss: 0.3918, Elapsed Time: 6.79 sec, Loader Iterations: 196  
Epoch [2/5], Loss: 0.3028, Elapsed Time: 0.56 sec, Loader Iterations: 196  
Epoch [3/5], Loss: 0.1997, Elapsed Time: 0.57 sec, Loader Iterations: 196  
Epoch [4/5], Loss: 0.1883, Elapsed Time: 0.57 sec, Loader Iterations: 196  
Epoch [5/5], Loss: 0.3806, Elapsed Time: 0.56 sec, Loader Iterations: 196

##### III.2.b) Imprima o número de amostras do último batch do train\_loader e justifique o valor encontrado? Ele pode ser menor que o batch\_size?

Number of samples in last batch: 40

O valor encontrado é menor que o tamanho do batch size (nesse caso, 128) pois esta é a quantidade de amostras restantes nas base. Como temos 196 iterações, o total de amostras nos primeiros 195 ciclos totaliza 24.960. Portanto, o último batch tem um total de 25.000 (tamanho da base) - 24.960 = 40.

##### III.2.c) Calcule R, a relação do número de amostras positivas sobre o número de amostras no batch e no final encontre o valor médio de R, para ver se o data loader está entregando batches balanceados. Desta vez, em vez de fazer um laço explícito, utilize list comprehension para criar uma lista contendo a relação R de cada amostra no batch. No final, calcule a média dos elementos da lista para fornecer a resposta final.

Médias de R por época:

R avg: 0.4999  
R avg: 0.5004  
R avg: 0.5003  
R avg: 0.4999  
R avg: 0.5000

##### III.2.d) Mostre a estrutura de um dos batches. Cada batch foi criado no método **getitem** do Dataset, linha 20. É formado por uma tupla com o primeiro elemento sendo a codificação one-hot do texto e o segundo elemento o label esperado, indicando positivo ou negativo. Mostre o shape (linhas e colunas) e o tipo de dado (float ou integer), tanto da entrada da rede como do label esperado. Desta vez selecione um elemento do batch do train\_loader utilizando as funções next e iter: batch = next(iter(train\_loader)).

my\_loader = DataLoader(train\_data, batch\_size=1)  
batch = next(iter(my\_loader))  
  
# Dado de entrada  
entrada = batch[0].tolist()  
dado\_entrada = (batch[0])[0]  
  
print("Dado de entrada:")  
print(dado\_entrada.size())  
print(dado\_entrada.dtype)  
print()  
  
# Label  
print("Label:")  
lbl = batch[1]  
print(lbl.size())  
print(lbl.dtype)

Dado de entrada:  
torch.Size([20001])  
torch.float32  
  
Label:  
torch.Size([1])  
torch.int64

##### III.3.a) Verifique a influência do batch size na acurácia final do modelo. Experimente usar um batch size de 1 amostra apenas e outro com mais de 128 e comente sobre os resultados.

Notei que o cálculo de perda da linha criterion() gera um erro com batch size = 1, então usei batch size = 2 para este exercício.

# Acurácia com batch = 2  
train\_loader = DataLoader(train\_data, batch\_size=2, shuffle=train\_shuffle)  
model = OneHotMLP(vocab\_size) # to reset weights  
train\_mdl(model, best\_LR)  
eval\_mdl(model)  
print()  
  
# Acurácia com batch = 256  
train\_loader = DataLoader(train\_data, batch\_size=256, shuffle=train\_shuffle)  
model = OneHotMLP(vocab\_size) # to reset weights  
train\_mdl(model, best\_LR)  
eval\_mdl(model)  
print()

Epoch [1/5], Loss: 0.0217, Elapsed Time: 17.09 sec, Loader Iterations: 12500, Spls lst batch: 2 , R avg: 0.5000   
Epoch [2/5], Loss: 0.5510, Elapsed Time: 16.61 sec, Loader Iterations: 12500, Spls lst batch: 2 , R avg: 0.5000   
Epoch [3/5], Loss: 0.3738, Elapsed Time: 16.82 sec, Loader Iterations: 12500, Spls lst batch: 2 , R avg: 0.5000   
Epoch [4/5], Loss: 0.0002, Elapsed Time: 17.54 sec, Loader Iterations: 12500, Spls lst batch: 2 , R avg: 0.5000   
Epoch [5/5], Loss: 0.0039, Elapsed Time: 17.01 sec, Loader Iterations: 12500, Spls lst batch: 2 , R avg: 0.5000   
Test Accuracy: 86.568%  
  
Epoch [1/5], Loss: 0.4962, Elapsed Time: 0.35 sec, Loader Iterations: 98, Spls lst batch: 168 , R avg: 0.4999   
Epoch [2/5], Loss: 0.4107, Elapsed Time: 0.35 sec, Loader Iterations: 98, Spls lst batch: 168 , R avg: 0.5003   
Epoch [3/5], Loss: 0.3138, Elapsed Time: 0.38 sec, Loader Iterations: 98, Spls lst batch: 168 , R avg: 0.5000   
Epoch [4/5], Loss: 0.3772, Elapsed Time: 0.36 sec, Loader Iterations: 98, Spls lst batch: 168 , R avg: 0.4999   
Epoch [5/5], Loss: 0.2645, Elapsed Time: 0.38 sec, Loader Iterations: 98, Spls lst batch: 168 , R avg: 0.5002   
Test Accuracy: 87.728%

Pudemos verificar que o batch size muito reduzido aumenta em muito a acurácia, mas em contrapartida aumenta muito a complexidade computacional. O ganho da acurácia pode ser explicado pela melhoria na generalização. Nesse caso os pesos do modelo são atualizados depois da análise de cada amostra de forma independente. O aumento do batch size de 128 para 256 não trouxe ganhos na acurácia. Portanto, para datasets pequenos como o caso deste exercício, uma redução do tamanho do batch pode ser benéfico desde que o custo computacional não seja excessivo.

#### Seção IV

##### IV.1.a) Faça a predição do modelo utilizando um batch do train\_loader: extraia um batch do train\_loader, chame de (input, target), onde input é a entrada da rede e target é o label esperado. Como a rede está com seus parâmetros (weights) aleatórios, o logito de saída da rede será um valor aleatório, porém a chamada irá executar sem erros:

logit = model( input)

aplique a função sigmoidal ao logito para convertê-lo numa probabilidade de valor entre 0 e 1.

import numpy as np  
new\_loader = DataLoader(train\_data, batch\_size=128, shuffle=train\_shuffle)  
model = OneHotMLP(vocab\_size).to(device)  
  
input, target = next(iter(new\_loader))  
logit = model(input)  
  
# Define the sigmoid function  
def sigmoid(x):  
 return 1 / (1 + torch.exp(-x))  
  
probability = sigmoid(logit[0])\*100  
  
# Cálculo da probabilidade para a primeira amostra  
print(f'Probabilidade: {probability.item():.2f} %')

Probabilidade: 47.73 %

##### IV.1.b) Agora, treine a rede executando o notebook todo e verifique se a acurácia está alta. Agora repita o exercício anterior, porém agora, compare o valor da probabilidade encontrada com o target esperado e verifique se ele acertou. Você pode considerar que se a probabilidade for maior que 0.5, pode-se dar o label 1 e se for menor que 0.5, o label 0. Observe isso que é feito na linha 11 da seção VI - Avaliação.

import numpy as np  
new\_loader = DataLoader(train\_data, batch\_size=128, shuffle=train\_shuffle)  
model = OneHotMLP(vocab\_size).to(device)  
  
input, target = next(iter(new\_loader))  
logit = model(input).cpu()  
  
predicted = torch.round(torch.sigmoid(logit.squeeze()))  
  
print("Predição = ", predicted[0].item())  
print("Target Esperado = ", target[0].item())

Predição = 0.0  
Target Esperado = 1

Se você der um print no modelo: print(model), você obterá:

OneHotMLP(  
 (fc1): Linear(in\_features=20001, out\_features=200, bias=True)   
 (fc2): Linear(in\_features=200, out\_features=1, bias=True)   
 (relu): ReLU()  
)

Os pesos da primeira camada podem ser visualizados com model.fc1.weight e o elemento constante (bias) pode ser visualizado com model.fc1.bias  
Calcule o número de parâmetros do modelo, preenchendo a seguinte tabela (utilize shape para verificar a estrutura de cada parâmetro do modelo).

from tabulate import tabulate  
  
model = OneHotMLP(vocab\_size)  
  
w\_fc1 = model.fc1.weight.size()  
b\_fc1 = model.fc1.bias.size()  
w\_fc2 = model.fc2.weight.size()  
b\_fc2 = model.fc2.bias.size()  
  
print("Model Parameters:")  
print("FC1 weights dimensions: ", list(w\_fc1))  
print("FC2 weights dimensions: ", list (w\_fc2))  
print("FC1 bias dimensions: ", list(b\_fc1))  
print("FC2 bias dimensions: ", list (b\_fc2))  
print()  
  
# Table data  
data = [  
 ['', 'weight', 'bias', 'weight', 'bias', ''],  
 ['size', w\_fc1[0]\*w\_fc1[1], b\_fc1[0], w\_fc2[0]\*w\_fc2[1], b\_fc2[0], '']  
]  
  
# Headers  
headers = ['layer', 'fc1', '', 'fc2', '', 'total']  
  
# Print the table  
print(tabulate(data, headers=headers))

Model Parameters:  
FC1 weights dimensions: [200, 20001]  
FC2 weights dimensions: [1, 200]  
FC1 bias dimensions: [200]  
FC2 bias dimensions: [1]  
  
layer fc1 fc2 total  
------- ------- ---- ------ ---- -------  
 weight bias weight bias  
size 4000200 200 200 1

#### Seção V

##### V.1.a) Qual é o valor teórico da Loss quando o modelo não está treinado, mas apenas inicializado? Isto é, a probabilidade predita tanto para a classe 0 como para a classe 1, é sempre 0,5 ? Justifique. Atenção: na equação da Entropia Cruzada utilize o logaritmo natural.

Utilizando a equação da entropia cruzada, podemos obter o valor teórico da perda:

Loss = -1/n SumN (yi \* ln(y^i) + (1-yi)ln(1-y^i))

com y^i = 0.5, temos: ln(y^i) = ln(1-y^i) = ln(0.5) = -0.69314

Loss = -1/n SumN(yi \* (-0.69314) + (1-yi) \* (-0.69314)) Loss = -1/n SumN(-0.69314yi - 0.69314 + 0.69314yi)

cancelando ambos termos em yi ->

Loss = -1/n SumN(-0.69314) e portanto Loss = 0.69314 para o modelo inicializado independente do número de amostras N.

No entanto, para um modelo não inicializado, o valor da perda depende do valor dos pesos da rede neural não inicializada, que pode variar e não ser o mesmo que o valor teórico.

##### V.1.b) Utilize as amostras do primeiro batch: (input,target) = next(iter(train\_loader)) e calcule o valor da Loss utilizando a equação fornecida anteriormente utilizando o pytorch. Verifique se este valor confere com o valor teórico do exercício anterior.

new\_loader = DataLoader(train\_data, batch\_size=128, shuffle=train\_shuffle)  
model = OneHotMLP(vocab\_size).to(device)  
  
input, target = next(iter(new\_loader))  
logit = model(input)  
prob = torch.sigmoid(logit)  
  
# Calculo numérico da perda  
loss = - torch.sum(torch.mul(target, torch.log(prob).t()) + torch.mul(1-target, torch.log(1-prob).t())) / prob.shape[0]  
print(loss)

tensor(0.6917, device='cuda:0', grad\_fn=<DivBackward0>)

Notamos que para um batch acima o valor da perda calculada não é a mesma da perda teórica, mas é muito próxima devido aos valores dos pesos não inicializados da rede neural.

##### V.1.c) O pytorch possui várias funções que facilitam o cálculo da Loss pela Entropia Cruzada. Utilize a classe nn.BCELoss (Binary Cross Entropy Loss). Você primeiro deve instanciar uma função da classe nn.BCELoss. Esta função instanciada recebe dois parâmetros (probs , targets) e retorna a Loss. Use a busca do Google para ver a documentação do BCELoss do pytorch. Calcule então a função de Loss da entropia cruzada, porém usando agora a função instanciada pelo BCELoss e confira se o resultado é exatamente o mesmo obtido no exercício anterior.

loss\_fn = nn.BCELoss()  
  
loss = loss\_fn(prob.squeeze(), target.float())  
print(loss)

tensor(0.6917, device='cuda:0', grad\_fn=<BinaryCrossEntropyBackward0>)

Notamos que o valor foi o mesmo que o obtido acima.

##### V.1.d) Repita o mesmo exercício, porém agora usando a classe nn.BCEWithLogitsLoss, que é a opção utilizada no notebook. O resultado da Loss deve igualar aos resultados anteriores.

loss\_fn = nn.BCEWithLogitsLoss()  
  
loss = loss\_fn(logit.squeeze(), target.float())  
print(loss)

tensor(0.6917, device='cuda:0',  
 grad\_fn=<BinaryCrossEntropyWithLogitsBackward0>)

Novamente chegamos ao mesmo valor calculado.

##### V.2.a) Modifique a célula do laço de treinamento de modo que a primeira Loss a ser impressa seja a Loss com o modelo inicializado (isto é, sem nenhum treinamento), fornecendo a Loss esperada conforme os exercícios feitos anteriormente. Observe que desta forma, fica fácil verificar se o seu modelo está correto e a Loss está sendo calculada corretamente. Atenção: Mantenha esse código da impressão do valor da Loss inicial, antes do treinamento, nesta célula, pois ela é sempre útil para verificar se não tem nada errado, antes de começar o treinamento.

# Medição da perda  
def train\_first\_loss(model, lr):  
  
 model = model.to(device)  
 # Define loss and optimizer  
 criterion = nn.BCEWithLogitsLoss()  
  
 optimizer = optim.SGD(model.parameters(), lr)  
  
 # Training loop  
 num\_epochs = 5  
 # First loss calculation  
 is\_first\_loss = True  
  
 for epoch in range(num\_epochs):  
 start\_time = time.time()  
 model.train()  
  
 for inputs, labels in train\_loader:  
  
 if not preload\_to\_gpu:  
 inputs = inputs.to(device)  
 labels = labels.to(device)  
  
 # Forward pass  
 outputs = model(inputs)  
 loss = criterion(outputs.squeeze(), labels.float())  
  
 if is\_first\_loss:  
 print(f'Loss before training: {loss.item():.4f}')  
 is\_first\_loss = False  
 print()  
  
 # Backward and optimize  
 backward\_start = time.time()  
 optimizer.zero\_grad()  
 loss.backward()  
 optimizer.step()  
  
 print(f'Epoch [{epoch+1}/{num\_epochs}], \  
 Loss: {loss.item():.4f}')  
 print()  
  
model = OneHotMLP(vocab\_size)  
train\_first\_loss(model, best\_LR)

Loss before training: 0.6928  
  
Epoch [1/5], Loss: 0.4690  
Epoch [2/5], Loss: 0.3171  
Epoch [3/5], Loss: 0.3201  
Epoch [4/5], Loss: 0.3079  
Epoch [5/5], Loss: 0.3412

Notamos que o primeiro valor calculado da perda se manteve o mesmo.

##### V.2.b) Execute a célula de treinamento por uma segunda vez e observe que a Loss continua diminuindo e o modelo está continuando a ser treinado. O que é necessário fazer para que o treinamento comece novamente do modelo aleatório? Qual(is) célula(s) é(são) preciso executar antes de executar o laço de treinamento novamente?

Para que o treinamento inicie novamente, os pesos devem ser resetados a seus valores iniciais. Uma maneira de fazer isso é criando uma função que resete os parãmetros de cada camada, por exemplo:

def reset\_weights(model):  
 for module in model.modules():  
 if isinstance(module, nn.Linear):  
 module.reset\_parameters()

E adicionar ao loop de treinamento acima.

# Medição da perda  
def train\_first\_loss\_reset(model, lr):  
  
 model = model.to(device)  
 reset\_weights(model)  
 # Define loss and optimizer  
 criterion = nn.BCEWithLogitsLoss()  
  
 optimizer = optim.SGD(model.parameters(), lr)  
  
 # Training loop  
 num\_epochs = 5  
 # First loss calculation  
 is\_first\_loss = True  
  
 for epoch in range(num\_epochs):  
 start\_time = time.time()  
 model.train()  
  
 for inputs, labels in train\_loader:  
  
 if not preload\_to\_gpu:  
 inputs = inputs.to(device)  
 labels = labels.to(device)  
  
 # Forward pass  
 outputs = model(inputs)  
 loss = criterion(outputs.squeeze(), labels.float())  
  
 if is\_first\_loss:  
 print(f'Loss before training: {loss.item():.4f}')  
 is\_first\_loss = False  
 print()  
  
 # Backward and optimize  
 backward\_start = time.time()  
 optimizer.zero\_grad()  
 loss.backward()  
 optimizer.step()  
  
 print(f'Epoch [{epoch+1}/{num\_epochs}], \  
 Loss: {loss.item():.4f}')  
 print()

Sem o reset de parâmetros:

model = OneHotMLP(vocab\_size)  
train\_first\_loss(model, best\_LR)  
train\_first\_loss(model, best\_LR)

Loss before training: 0.6927  
  
Epoch [1/5], Loss: 0.4922  
Epoch [2/5], Loss: 0.3288  
Epoch [3/5], Loss: 0.3500  
Epoch [4/5], Loss: 0.2923  
Epoch [5/5], Loss: 0.3409  
  
Loss before training: 0.2243  
  
Epoch [1/5], Loss: 0.3253  
Epoch [2/5], Loss: 0.2242  
Epoch [3/5], Loss: 0.2936  
Epoch [4/5], Loss: 0.2388  
Epoch [5/5], Loss: 0.1457

Com reset de parâmetros:

model = OneHotMLP(vocab\_size)  
train\_first\_loss\_reset(model, best\_LR)  
train\_first\_loss\_reset(model, best\_LR)

Loss before training: 0.6957  
  
Epoch [1/5], Loss: 0.4929  
Epoch [2/5], Loss: 0.3588  
Epoch [3/5], Loss: 0.2829  
Epoch [4/5], Loss: 0.2523  
Epoch [5/5], Loss: 0.2483  
  
Loss before training: 0.6972  
  
Epoch [1/5], Loss: 0.4644  
Epoch [2/5], Loss: 0.3141  
Epoch [3/5], Loss: 0.3423  
Epoch [4/5], Loss: 0.3514  
Epoch [5/5], Loss: 0.3239

##### V.3.a) Repita o exercício V.1.a) porém agora utilizando a equação acima.

M - número de amostras  
N - número de classes

loss = - 1/(M*N) sumM (sumN (yij*log(y^ij)))

Podemos supor 2 classes (positiva e negativa, e portanto y^ij = 50%)

daí log(y^ij) = log(0.5) = -0.69314.

Com duas classes:

sumN(yij*log(y^ij)) = 2*(yij*(-0.69314)) = -1.38628*yij

loss = -1/(M*N)* sumM(-1.38628\*yij)

Se temos duas classes, podemos assumir que metade são da classe 0 e metade da classe 1, e portanto

sumM(yij) = M/2

daí a perda seria dada por

loss = -1/(M*2)*(M/2)\*-1.38628 = 1.38628/2 = 0.69314.

##### V.3.b) Modifique a camada de saída da rede para 2 logitos e utilize a função Softmax para converter os logitos em probabilidades. Repita o exercício V.1.b)

class OneHotMLP\_2logits(nn.Module):  
 def \_\_init\_\_(self, vocab\_size):  
 super(OneHotMLP\_2logits, self).\_\_init\_\_()  
 self.fc1 = nn.Linear(vocab\_size+1, 200)  
 self.fc2 = nn.Linear(200, 2)  
 self.relu = nn.ReLU()  
 self.softmax = nn.Softmax(dim=1)  
  
 def forward(self, x):  
 o = self.fc1(x.float())  
 o = self.relu(o)  
 o = self.fc2(o)  
 return self.softmax(o)

import torch.nn.functional as F  
  
new\_loader = DataLoader(train\_data, batch\_size=128, shuffle=train\_shuffle)  
model = OneHotMLP\_2logits(vocab\_size).to(device)  
  
input, target = next(iter(new\_loader))  
probs\_2logits = model(input)  
  
# Calculo numérico da perda  
log\_probs = F.log\_softmax(probs\_2logits, dim=1)  
log\_probs\_correct\_class = torch.gather(log\_probs, 1, target.unsqueeze(1)).squeeze(1)  
loss = -log\_probs\_correct\_class.mean()  
print(loss)

tensor(0.6941, device='cuda:0', grad\_fn=<NegBackward0>)

##### V.3.c) Utilize agora a função nn.CrossEntropyLoss para calcular a Loss e verifique se os resultados são os mesmos que anteriormente.

loss\_fn = nn.CrossEntropyLoss()  
  
loss = loss\_fn(probs\_2logits.squeeze(), target)  
print(loss)

tensor(0.6941, device='cuda:0', grad\_fn=<NllLossBackward0>)

Notamos que o valor foi o mesmo que o obtido acima.

##### V.3.d) Modifique as seções V e VI para que o notebook funcione com a saída da rede com 2 logitos. Há necessidade de alterar o laço de treinamento e o laço de cálculo da acurácia.

# Treinamento e inferência multi-classe  
  
def train\_two\_logits(model, lr):  
  
 model = model.to(device)  
 reset\_weights(model)  
 # Define loss and optimizer  
 criterion = nn.CrossEntropyLoss()  
  
 optimizer = optim.SGD(model.parameters(), lr)  
  
 # Training loop  
 num\_epochs = 5  
  
 for epoch in range(num\_epochs):  
  
 model.train()  
  
 for inputs, labels in train\_loader:  
  
 if not preload\_to\_gpu:  
 inputs = inputs.to(device)  
 labels = labels.to(device)  
  
 # Forward pass  
 outputs = model(inputs)  
 loss = criterion(outputs.squeeze(), labels)  
  
 # Backward and optimize  
 optimizer.zero\_grad()  
 loss.backward()  
 optimizer.step()  
  
 print(f'Epoch [{epoch+1}/{num\_epochs}], \  
 Loss: {loss.item():.4f}')  
 print()  
  
def eval\_two\_logits(model):  
 model.eval()  
  
 with torch.no\_grad():  
 correct = 0  
 total = 0  
 for inputs, labels in test\_loader:  
 inputs = inputs.to(device)  
 labels = labels.to(device)  
 outputs = model(inputs)  
 \_, predicted = torch.max(outputs, 1)  
 total += labels.size(0)  
 correct += (predicted == labels).sum().item()  
  
 acc = 100\* correct/total  
 print(f'Test Accuracy: {acc}%')  
 return acc  
  
  
model = OneHotMLP\_2logits(vocab\_size)  
train\_two\_logits(model, best\_LR)  
eval\_two\_logits(model)

Epoch [1/5], Loss: 0.5940  
Epoch [2/5], Loss: 0.5067  
Epoch [3/5], Loss: 0.4641  
Epoch [4/5], Loss: 0.4820  
Epoch [5/5], Loss: 0.4484  
  
Test Accuracy: 86.74%

86.74

#### Seção VI

##### VI.1.a) Calcule o número de amostras que está sendo considerado na seção de avaliação.

print(len(test\_data))

25000

##### VI.1.b) Explique o que faz os comandos model.eval()e with torch.no\_grad().

O comando model.eval() informa para o Pytorch que estamos em modo de inferência, o que faz com que algumas camadas dos modelos (como camadas de dropout) sejam desabilitadas.

O loop **with torch.no\_grad()** informa o Pytorch para não calcular gradientes relacionados a um tensor. Assim, loops onde o gradiente precisa ser preservado utilizam essa configuração.

##### VI.1.c) Existe uma forma mais simples de calcular a classe predita na linha 11, sem a necessidade de usar a função torch.sigmoid?

Torch.sigmoid() é uma função de ativação, para transformar uma entrada numérica em um número entre zero e um. Uma maneira muito simples de fazer a mesma coisa é dividir a entrada pelo valor máximo da entrada observada, além de, claro, utilizar outras funções de ativação alternativas (ReLU, etc).

##### VI.2.a) Utilizando a resposta do exercício V.1.a, que é a Loss teórica de um modelo aleatório de 2 classes, qual é o valor da perplexidade?

torch.exp(torch.tensor(-0.69314))

tensor(0.5000)

A perplexidade neste caso nos retorna a probabilidade de distribuição das classes de 50%.

##### VI.2.b) E se o modelo agora fosse para classificar a amostra em N classes, qual seria o valor da perplexidade para o caso aleatório?

Para N classes, a perplexidade seria dada por 1/N.

##### VI.2.c) Qual é o valor da perplexidade quando o modelo acerta todas as classes com 100% de probabilidade?

Quando um modelo acerta 100% das previsões, a perplexidade é 1.

##### VI.3.a) Modifique o código da seção VI - Avaliação, para que além de calcular a acurácia, calcule também a perplexidade. lembrar que PPL = torch.exp(CE). Assim, será necessário calcular a entropia cruzada, como feito no laço de treinamento.

def eval\_with\_perplexity(model):  
 model.eval()  
  
 criterion = nn.CrossEntropyLoss()  
 #total\_loss = 0  
 #total\_labels = 0  
 perplexity = 0  
  
 with torch.no\_grad():  
 correct = 0  
 total = 0  
 for inputs, labels in test\_loader:  
 inputs = inputs.to(device)  
 labels = labels.to(device)  
 outputs = model(inputs)  
  
 loss = criterion(outputs, labels)  
 perplexity = torch.exp(loss)  
  
 \_, predicted = torch.max(outputs, 1)  
 total += labels.size(0)  
 correct += (predicted == labels).sum().item()  
  
 acc = 100\* correct/total  
 print(f'Test Accuracy: {acc}% \  
 Test Perplexity: {perplexity}')  
 return acc  
  
eval\_with\_perplexity(model)

Test Accuracy: 86.74% Test Perplexity: 1.5994813442230225

86.74

##### VI.4.a) Modifique o laço de treinamento para incorporar também o cálculo da avaliação ao final de cada época. Aproveite para reportar também a perplexidade, tanto do treinamento como da avaliação (observe que será mais fácil de interpretar). Essa é a forma usual de se fazer o treinamento, monitorando se o modelo não entra em overfitting.

def train\_and\_eval(model, lr, epochs):  
  
 model = model.to(device)  
 reset\_weights(model)  
 # Define loss and optimizer  
 criterion = nn.CrossEntropyLoss()  
  
 optimizer = optim.SGD(model.parameters(), lr)  
  
 perplexity = 0  
 for epoch in range(epochs):  
  
 model.train()  
  
 for inputs, labels in train\_loader:  
  
 if not preload\_to\_gpu:  
 inputs = inputs.to(device)  
 labels = labels.to(device)  
  
 # Forward pass  
 outputs = model(inputs)  
 loss = criterion(outputs.squeeze(), labels)  
 perplexity = torch.exp(loss)  
  
 # Backward and optimize  
 optimizer.zero\_grad()  
 loss.backward()  
 optimizer.step()  
  
 eval\_with\_perplexity(model)  
 model.train()  
  
 print(f'Epoch [{epoch+1}/{epochs}], \  
 Loss: {loss.item():.4f}, \  
 Train Perplexity: {perplexity}')  
 print()  
  
model = OneHotMLP\_2logits(vocab\_size)  
train\_and\_eval(model, best\_LR, 5)

Test Accuracy: 79.372% Test Perplexity: 1.8890479803085327  
Epoch [1/5], Loss: 0.6000, Train Perplexity: 1.822061538696289  
Test Accuracy: 83.416% Test Perplexity: 1.7140659093856812  
Epoch [2/5], Loss: 0.5073, Train Perplexity: 1.6608058214187622  
Test Accuracy: 85.052% Test Perplexity: 1.6764986515045166  
Epoch [3/5], Loss: 0.4443, Train Perplexity: 1.559381365776062  
Test Accuracy: 86.024% Test Perplexity: 1.583828091621399  
Epoch [4/5], Loss: 0.4751, Train Perplexity: 1.6081184148788452  
Test Accuracy: 86.616% Test Perplexity: 1.6186288595199585  
Epoch [5/5], Loss: 0.4688, Train Perplexity: 1.5980067253112793

##### Por fim, como o dataset tem muitas amostras, ele é demorado de entrar em overfitting. Para ficar mais evidente, diminua novamente o número de amostras do dataset de treino de 25 mil para 1 mil amostras e aumente o número de épocas para ilustrar o caso do overfitting, em que a perplexidade de treinamento continua caindo, porém a perplexidade no conjunto de teste começa a aumentar.

batch\_size = 128  
  
train\_data\_short = IMDBDataset('train', vocab, samples = 1000)  
test\_data\_short = IMDBDataset('test', vocab, samples = 1000)  
  
train\_loader = DataLoader(train\_data\_short, batch\_size=batch\_size, shuffle=train\_shuffle)  
test\_loader = DataLoader(test\_data\_short, batch\_size=batch\_size, shuffle=False)  
  
model = OneHotMLP\_2logits(vocab\_size)  
train\_and\_eval(model, best\_LR, 100)

Test Accuracy: 100.0% Test Perplexity: 1.4652374982833862  
Epoch [1/100], Loss: 0.3981, Train Perplexity: 1.4890598058700562  
Test Accuracy: 100.0% Test Perplexity: 1.402090311050415  
Epoch [2/100], Loss: 0.3379, Train Perplexity: 1.4019744396209717  
Test Accuracy: 100.0% Test Perplexity: 1.3873093128204346  
Epoch [3/100], Loss: 0.3279, Train Perplexity: 1.3881028890609741  
Test Accuracy: 100.0% Test Perplexity: 1.3811591863632202  
Epoch [4/100], Loss: 0.3222, Train Perplexity: 1.3801277875900269  
Test Accuracy: 100.0% Test Perplexity: 1.3778650760650635  
Epoch [5/100], Loss: 0.3193, Train Perplexity: 1.376153588294983  
Test Accuracy: 100.0% Test Perplexity: 1.3758258819580078  
Epoch [6/100], Loss: 0.3194, Train Perplexity: 1.376289963722229  
Test Accuracy: 100.0% Test Perplexity: 1.3744561672210693  
Epoch [7/100], Loss: 0.3184, Train Perplexity: 1.3749326467514038  
Test Accuracy: 100.0% Test Perplexity: 1.373476505279541  
Epoch [8/100], Loss: 0.3167, Train Perplexity: 1.3725703954696655  
Test Accuracy: 100.0% Test Perplexity: 1.372735619544983  
Epoch [9/100], Loss: 0.3178, Train Perplexity: 1.374079704284668  
Test Accuracy: 100.0% Test Perplexity: 1.3721671104431152  
Epoch [10/100], Loss: 0.3151, Train Perplexity: 1.370390772819519  
Test Accuracy: 100.0% Test Perplexity: 1.3717094659805298  
Epoch [11/100], Loss: 0.3163, Train Perplexity: 1.3721094131469727  
Test Accuracy: 100.0% Test Perplexity: 1.371337890625  
Epoch [12/100], Loss: 0.3155, Train Perplexity: 1.37095308303833  
Test Accuracy: 100.0% Test Perplexity: 1.3710299730300903  
Epoch [13/100], Loss: 0.3152, Train Perplexity: 1.3705111742019653  
Test Accuracy: 100.0% Test Perplexity: 1.3707679510116577  
Epoch [14/100], Loss: 0.3165, Train Perplexity: 1.372287631034851  
Test Accuracy: 100.0% Test Perplexity: 1.3705463409423828  
Epoch [15/100], Loss: 0.3147, Train Perplexity: 1.3699136972427368  
Test Accuracy: 100.0% Test Perplexity: 1.3703547716140747  
Epoch [16/100], Loss: 0.3148, Train Perplexity: 1.3700511455535889  
Test Accuracy: 100.0% Test Perplexity: 1.37018620967865  
Epoch [17/100], Loss: 0.3158, Train Perplexity: 1.371317744255066  
Test Accuracy: 100.0% Test Perplexity: 1.3700395822525024  
Epoch [18/100], Loss: 0.3147, Train Perplexity: 1.3699020147323608  
Test Accuracy: 100.0% Test Perplexity: 1.3699101209640503  
Epoch [19/100], Loss: 0.3144, Train Perplexity: 1.3694963455200195  
Test Accuracy: 100.0% Test Perplexity: 1.3697941303253174  
Epoch [20/100], Loss: 0.3145, Train Perplexity: 1.3696213960647583  
Test Accuracy: 100.0% Test Perplexity: 1.3696900606155396  
Epoch [21/100], Loss: 0.3147, Train Perplexity: 1.3698198795318604  
Test Accuracy: 100.0% Test Perplexity: 1.3695961236953735  
Epoch [22/100], Loss: 0.3146, Train Perplexity: 1.3697214126586914  
Test Accuracy: 100.0% Test Perplexity: 1.3695107698440552  
Epoch [23/100], Loss: 0.3147, Train Perplexity: 1.3699082136154175  
Test Accuracy: 100.0% Test Perplexity: 1.3694332838058472  
Epoch [24/100], Loss: 0.3147, Train Perplexity: 1.3698039054870605  
Test Accuracy: 100.0% Test Perplexity: 1.3693631887435913  
Epoch [25/100], Loss: 0.3140, Train Perplexity: 1.368842363357544  
Test Accuracy: 100.0% Test Perplexity: 1.3692984580993652  
Epoch [26/100], Loss: 0.3144, Train Perplexity: 1.3694206476211548  
Test Accuracy: 100.0% Test Perplexity: 1.3692387342453003  
Epoch [27/100], Loss: 0.3146, Train Perplexity: 1.3697483539581299  
Test Accuracy: 100.0% Test Perplexity: 1.3691834211349487  
Epoch [28/100], Loss: 0.3147, Train Perplexity: 1.3698594570159912  
Test Accuracy: 100.0% Test Perplexity: 1.3691322803497314  
Epoch [29/100], Loss: 0.3146, Train Perplexity: 1.369748592376709  
Test Accuracy: 100.0% Test Perplexity: 1.3690849542617798  
Epoch [30/100], Loss: 0.3146, Train Perplexity: 1.369724988937378  
Test Accuracy: 100.0% Test Perplexity: 1.3690409660339355  
Epoch [31/100], Loss: 0.3139, Train Perplexity: 1.3687975406646729  
Test Accuracy: 100.0% Test Perplexity: 1.368999719619751  
Epoch [32/100], Loss: 0.3142, Train Perplexity: 1.3691222667694092  
Test Accuracy: 100.0% Test Perplexity: 1.368961215019226  
Epoch [33/100], Loss: 0.3141, Train Perplexity: 1.3689675331115723  
Test Accuracy: 100.0% Test Perplexity: 1.3689253330230713  
Epoch [34/100], Loss: 0.3143, Train Perplexity: 1.3693097829818726  
Test Accuracy: 100.0% Test Perplexity: 1.368891716003418  
Epoch [35/100], Loss: 0.3137, Train Perplexity: 1.368466854095459  
Test Accuracy: 100.0% Test Perplexity: 1.368859887123108  
Epoch [36/100], Loss: 0.3139, Train Perplexity: 1.368798017501831  
Test Accuracy: 100.0% Test Perplexity: 1.3688302040100098  
Epoch [37/100], Loss: 0.3138, Train Perplexity: 1.3685476779937744  
Test Accuracy: 100.0% Test Perplexity: 1.3688018321990967  
Epoch [38/100], Loss: 0.3140, Train Perplexity: 1.3688825368881226  
Test Accuracy: 100.0% Test Perplexity: 1.3687752485275269  
Epoch [39/100], Loss: 0.3136, Train Perplexity: 1.3684091567993164  
Test Accuracy: 100.0% Test Perplexity: 1.3687498569488525  
Epoch [40/100], Loss: 0.3138, Train Perplexity: 1.3686186075210571  
Test Accuracy: 100.0% Test Perplexity: 1.3687258958816528  
Epoch [41/100], Loss: 0.3139, Train Perplexity: 1.3687593936920166  
Test Accuracy: 100.0% Test Perplexity: 1.368703007698059  
Epoch [42/100], Loss: 0.3142, Train Perplexity: 1.3691896200180054  
Test Accuracy: 100.0% Test Perplexity: 1.3686814308166504  
Epoch [43/100], Loss: 0.3137, Train Perplexity: 1.3684498071670532  
Test Accuracy: 100.0% Test Perplexity: 1.3686609268188477  
Epoch [44/100], Loss: 0.3137, Train Perplexity: 1.3684937953948975  
Test Accuracy: 100.0% Test Perplexity: 1.3686413764953613  
Epoch [45/100], Loss: 0.3136, Train Perplexity: 1.368397831916809  
Test Accuracy: 100.0% Test Perplexity: 1.3686225414276123  
Epoch [46/100], Loss: 0.3138, Train Perplexity: 1.3685566186904907  
Test Accuracy: 100.0% Test Perplexity: 1.3686045408248901  
Epoch [47/100], Loss: 0.3143, Train Perplexity: 1.3692359924316406  
Test Accuracy: 100.0% Test Perplexity: 1.3685874938964844  
Epoch [48/100], Loss: 0.3137, Train Perplexity: 1.368489146232605  
Test Accuracy: 100.0% Test Perplexity: 1.368571162223816  
Epoch [49/100], Loss: 0.3139, Train Perplexity: 1.3687890768051147  
Test Accuracy: 100.0% Test Perplexity: 1.3685554265975952  
Epoch [50/100], Loss: 0.3138, Train Perplexity: 1.3686046600341797  
Test Accuracy: 100.0% Test Perplexity: 1.3685404062271118  
Epoch [51/100], Loss: 0.3139, Train Perplexity: 1.368720293045044  
Test Accuracy: 100.0% Test Perplexity: 1.3685258626937866  
Epoch [52/100], Loss: 0.3142, Train Perplexity: 1.369149923324585  
Test Accuracy: 100.0% Test Perplexity: 1.3685119152069092  
Epoch [53/100], Loss: 0.3136, Train Perplexity: 1.368369698524475  
Test Accuracy: 100.0% Test Perplexity: 1.368498682975769  
Epoch [54/100], Loss: 0.3136, Train Perplexity: 1.3682812452316284  
Test Accuracy: 100.0% Test Perplexity: 1.3684860467910767  
Epoch [55/100], Loss: 0.3137, Train Perplexity: 1.368517518043518  
Test Accuracy: 100.0% Test Perplexity: 1.3684738874435425  
Epoch [56/100], Loss: 0.3135, Train Perplexity: 1.3682323694229126  
Test Accuracy: 100.0% Test Perplexity: 1.3684619665145874  
Epoch [57/100], Loss: 0.3137, Train Perplexity: 1.3685381412506104  
Test Accuracy: 100.0% Test Perplexity: 1.368450403213501  
Epoch [58/100], Loss: 0.3142, Train Perplexity: 1.369145154953003  
Test Accuracy: 100.0% Test Perplexity: 1.3684395551681519  
Epoch [59/100], Loss: 0.3134, Train Perplexity: 1.3680731058120728  
Test Accuracy: 100.0% Test Perplexity: 1.3684290647506714  
Epoch [60/100], Loss: 0.3136, Train Perplexity: 1.368316411972046  
Test Accuracy: 100.0% Test Perplexity: 1.3684189319610596  
Epoch [61/100], Loss: 0.3136, Train Perplexity: 1.368345022201538  
Test Accuracy: 100.0% Test Perplexity: 1.3684089183807373  
Epoch [62/100], Loss: 0.3139, Train Perplexity: 1.368705153465271  
Test Accuracy: 100.0% Test Perplexity: 1.3683993816375732  
Epoch [63/100], Loss: 0.3140, Train Perplexity: 1.3688586950302124  
Test Accuracy: 100.0% Test Perplexity: 1.3683900833129883  
Epoch [64/100], Loss: 0.3143, Train Perplexity: 1.3693068027496338  
Test Accuracy: 100.0% Test Perplexity: 1.368381142616272  
Epoch [65/100], Loss: 0.3135, Train Perplexity: 1.368220567703247  
Test Accuracy: 100.0% Test Perplexity: 1.3683725595474243  
Epoch [66/100], Loss: 0.3136, Train Perplexity: 1.3683326244354248  
Test Accuracy: 100.0% Test Perplexity: 1.3683642148971558  
Epoch [67/100], Loss: 0.3135, Train Perplexity: 1.3681540489196777  
Test Accuracy: 100.0% Test Perplexity: 1.3683559894561768  
Epoch [68/100], Loss: 0.3139, Train Perplexity: 1.3686999082565308  
Test Accuracy: 100.0% Test Perplexity: 1.368348240852356  
Epoch [69/100], Loss: 0.3135, Train Perplexity: 1.368147611618042  
Test Accuracy: 100.0% Test Perplexity: 1.3683404922485352  
Epoch [70/100], Loss: 0.3139, Train Perplexity: 1.3687909841537476  
Test Accuracy: 100.0% Test Perplexity: 1.3683332204818726  
Epoch [71/100], Loss: 0.3139, Train Perplexity: 1.3687148094177246  
Test Accuracy: 100.0% Test Perplexity: 1.36832594871521  
Epoch [72/100], Loss: 0.3136, Train Perplexity: 1.3683280944824219  
Test Accuracy: 100.0% Test Perplexity: 1.3683189153671265  
Epoch [73/100], Loss: 0.3139, Train Perplexity: 1.3687971830368042  
Test Accuracy: 100.0% Test Perplexity: 1.368312120437622  
Epoch [74/100], Loss: 0.3139, Train Perplexity: 1.368705153465271  
Test Accuracy: 100.0% Test Perplexity: 1.3683055639266968  
Epoch [75/100], Loss: 0.3135, Train Perplexity: 1.3682001829147339  
Test Accuracy: 100.0% Test Perplexity: 1.3682992458343506  
Epoch [76/100], Loss: 0.3136, Train Perplexity: 1.3683652877807617  
Test Accuracy: 100.0% Test Perplexity: 1.3682929277420044  
Epoch [77/100], Loss: 0.3135, Train Perplexity: 1.3682271242141724  
Test Accuracy: 100.0% Test Perplexity: 1.3682869672775269  
Epoch [78/100], Loss: 0.3135, Train Perplexity: 1.3681442737579346  
Test Accuracy: 100.0% Test Perplexity: 1.3682812452316284  
Epoch [79/100], Loss: 0.3135, Train Perplexity: 1.3682386875152588  
Test Accuracy: 100.0% Test Perplexity: 1.3682754039764404  
Epoch [80/100], Loss: 0.3134, Train Perplexity: 1.3681339025497437  
Test Accuracy: 100.0% Test Perplexity: 1.368269920349121  
Epoch [81/100], Loss: 0.3138, Train Perplexity: 1.3685861825942993  
Test Accuracy: 100.0% Test Perplexity: 1.3682644367218018  
Epoch [82/100], Loss: 0.3135, Train Perplexity: 1.3681398630142212  
Test Accuracy: 100.0% Test Perplexity: 1.3682591915130615  
Epoch [83/100], Loss: 0.3135, Train Perplexity: 1.368175745010376  
Test Accuracy: 100.0% Test Perplexity: 1.3682540655136108  
Epoch [84/100], Loss: 0.3135, Train Perplexity: 1.368200421333313  
Test Accuracy: 100.0% Test Perplexity: 1.3682489395141602  
Epoch [85/100], Loss: 0.3137, Train Perplexity: 1.3684122562408447  
Test Accuracy: 100.0% Test Perplexity: 1.3682440519332886  
Epoch [86/100], Loss: 0.3140, Train Perplexity: 1.3688278198242188  
Test Accuracy: 100.0% Test Perplexity: 1.3682392835617065  
Epoch [87/100], Loss: 0.3135, Train Perplexity: 1.3682599067687988  
Test Accuracy: 100.0% Test Perplexity: 1.3682345151901245  
Epoch [88/100], Loss: 0.3135, Train Perplexity: 1.368152141571045  
Test Accuracy: 100.0% Test Perplexity: 1.3682301044464111  
Epoch [89/100], Loss: 0.3134, Train Perplexity: 1.3681014776229858  
Test Accuracy: 100.0% Test Perplexity: 1.3682255744934082  
Epoch [90/100], Loss: 0.3136, Train Perplexity: 1.3683029413223267  
Test Accuracy: 100.0% Test Perplexity: 1.3682212829589844  
Epoch [91/100], Loss: 0.3134, Train Perplexity: 1.3680342435836792  
Test Accuracy: 100.0% Test Perplexity: 1.36821711063385  
Epoch [92/100], Loss: 0.3135, Train Perplexity: 1.3682141304016113  
Test Accuracy: 100.0% Test Perplexity: 1.3682128190994263  
Epoch [93/100], Loss: 0.3138, Train Perplexity: 1.368651270866394  
Test Accuracy: 100.0% Test Perplexity: 1.368208885192871  
Epoch [94/100], Loss: 0.3135, Train Perplexity: 1.3681857585906982  
Test Accuracy: 100.0% Test Perplexity: 1.3682047128677368  
Epoch [95/100], Loss: 0.3139, Train Perplexity: 1.3687580823898315  
Test Accuracy: 100.0% Test Perplexity: 1.3682007789611816  
Epoch [96/100], Loss: 0.3135, Train Perplexity: 1.368249535560608  
Test Accuracy: 100.0% Test Perplexity: 1.368196964263916  
Epoch [97/100], Loss: 0.3138, Train Perplexity: 1.3686814308166504  
Test Accuracy: 100.0% Test Perplexity: 1.36819326877594  
Epoch [98/100], Loss: 0.3138, Train Perplexity: 1.3686493635177612  
Test Accuracy: 100.0% Test Perplexity: 1.3681894540786743  
Epoch [99/100], Loss: 0.3136, Train Perplexity: 1.3683782815933228  
Test Accuracy: 100.0% Test Perplexity: 1.3681858777999878  
Epoch [100/100], Loss: 0.3142, Train Perplexity: 1.3692086935043335