Exercícios Pré-Curso IA-024 2S2024 FEEC-UNICAMP

Aluno: Lucas Couto Lima RA: 220696

I - Vocabulário e tokenização

Exercícios 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.

Respostas:
I.1.a) a modificação do trecho de código

```
# limit the vocabulary size to 20000 most frequent tokens
vocab_size = 20000

counter = Counter()
counter_label = Counter({0: 0, 1: 0})
total_text_length = 0
for sample in train_dataset:
    counter.update(sample["text"].split())
    label = sample["label"]
    counter_label[label] += 1

    total_text_length += len(sample["text"].split())

# Calcula o comprimento médio do texto
total_size = counter_label[1]+counter_label[0]
average_text_length = total_text_length / total_size

# Adição: Exibição dos resultados
print(f"Número de amostras positivas: {counter_label[1]}")
print(f"Número de amostras negativas: {counter_label[0]}")
print(f"Número total de amostras: {total_size}")
```

```
print(f"Comprimento médio do texto (em número de palavras):
{average_text_length}")
```

I.1.b) número de amostras positivas, amostras negativas e amostras totais

Número de amostras positivas: 12500
Número de amostras negativas: 12500
Número de amostras totais: 25000

I.1.c) comprimento médio dos textos das amostras (em número de palavras)

Comprimento médio dos textos das amostras: 233.7872

Exercícios I.2

As linhas 9 e 10 da célula do vocabulário são linhas típicas de programação python em listas com dicionários com laços na forma compreensão de listas ou *list comprehension* em inglês. Procure analisar e estudar profundamente o uso de lista e dicionário do python. Estude também a função *encode_sentence.*

Mostre as cinco palavras mais frequentes do vocabulário e as cinco palavras menos frequentes. Qual é o código do token que está sendo utilizado quando a palavra não está no vocabulário? Calcule quantos tokens das frases do conjunto de treinamento que não estão no vocabulário.

Respostas:
I.2.a) Cinco palavras mais frequentes, e as cinco menos frequentes. Mostre o código utilizado, usando fatiamento de listas (*list slicing*).

Cinco palavras mais frequentes: ['the', 'a', 'and', 'of', 'to']
Cinco palavras menos frequentes: ['age-old', 'place!', 'Bros', 'tossing', 'nation,']

```
print(f"Cinco palavras mais frequentes: {most_frequent_words[:5]}")
print(f"Cinco palavras menos frequentes: {most_frequent_words[-5:]}")
```

I.2.b) Explique onde está a codificação que atribui o código de "unknown token" e qual é esse código.

A codificação está na linha:

```
return [vocab.get(word, 0) for word in sentence.split()] # 0 for OOV
```

A função busca a palavra “word” no dicionário “vocab”. Se a palavra não estiver no dicionário, a função retorna 0, que é o código para "unknown token". Isso é indicado pelo segundo argumento da função “get”.

I.2.c) Calcule o número de *unknown tokens* no conjunto de treinamento e mostre o código de como ele foi calculado.

Número de tokens desconhecidos no conjunto de treinamento: 566141

```
# Calcular quantos tokens das frases do conjunto de treinamento não estão no vocabulário
num_unknown_tokens = 0
for sample in train_dataset:
    encoded_sentence = encode_sentence(sample["text"], vocab)
    num_unknown_tokens += encoded_sentence.count(0)

print(f"Número de tokens desconhecidos no conjunto de treinamento: {num_unknown_tokens}")
```

Reduzindo o número de amostras para 200

Para se depurar um programa de machine learning, é usual utilizar bem poucas amostras até que o programa esteja funcionando, pois assim, o ciclo completo de treinamento e verificação do resultado fica muito mais rápido.
Os próximos exercícios deverão ser feitos apenas com 200 amostras do dataset para que o programa seja mais fácil de entender e experimentar.

Uma forma simples de reduzir o número de amostras é utilizar o fatiamento de listas para selecionar apenas as primeiras 200 amostras utilizando [:200] na lista do dataset IMDB:

```
self.data = list(load_dataset("stanfordnlp/imdb", split="train"))[:200]
```

Faça a redução para 200 amostras no split de treino e no de teste.

Faça isto, tanto na linha 5 da célula de calcular o vocabulário como na linha 5 da célula do "**II - Dataset**".
Com estas duas modificações, execute o notebook por completo novamente. Você verá que o tempo de processamento cairá drasticamente, para aproximadamente 1 a 2 segundos por época. Porém você vai notar que a Acurácia calculada na célula **VI - Avaliação** sobe para 100% ou próximo disso.
Consegue justificar a razão deste resultado inesperado, entendendo que no treinamento, as perdas em cada época continuam próximas de valores com todo o dataset?
Para ver a resposta, verifique agora no dataset com 200 amostras, quantas são as amostras positivas e quantas são as amostras negativas no dataset de teste.

Exercícios I.3:

I.3.a) Calcule novamente o número de amostras positivas e negativas das 200 amostras de treinamento e de teste?

Treinamento:
Número de amostras positivas: 0
Número de amostras negativas: 200

Teste:

Número de amostras positivas: 0
Número de amostras negativas: 200

I.3.b) Procure balancear os dois datasets de forma que as 200 amostras do dataset fiquem balanceadas, isto é, aproximadamente 100 amostras positivas e 100 amostras negativas. Existem várias maneiras de se fazer este exercício. Procure fazer uma forma simples.

```
def balance_dataset(dataset, num_samples_per_class=100):
    # Inicialize contadores e listas para armazenar amostras
balanceadas
    counter_label = Counter({0: 0, 1: 0})
    balanced_dataset = []

    # Itere sobre o dataset e selecione as amostras balanceadas
    for sample in dataset:
        label = sample["label"]
        if counter_label[label] < num_samples_per_class:
            balanced_dataset.append(sample)
            counter_label[label] += 1
        if counter_label[0] >= num_samples_per_class and
counter_label[1] >= num_samples_per_class:
            break

    return balanced_dataset, counter_label

# Balanceando o conjunto de treinamento
balanced_train_dataset, train_counter_label =
balance_dataset(list(train_dataset))
# Balanceando o conjunto de teste
balanced_test_dataset, test_counter_label =
balance_dataset(list(test_dataset))
```

II - Dataset

Precisamos entender como funciona a classe `IMDBDataset`. Ela é a classe responsável para acessar cada amostra do dataset.

Em primeiro lugar precisamos entender qual será a entrada da rede neural para decidir se o texto é uma crítica positiva ou negativa. Uma das formas mais simples de construir um modelo preditivo é com base nas palavras utilizadas no texto. A distribuição das palavras de um texto tem alta correlação com o fato do texto estar falando bem ou falando mal de um

filme. Certamente é estimativa que possui seus erros, mas é a forma mais simples e eficiente de se fazer uma análise de sentimento ou de maneira geral uma classificação de um texto. Esse método é denominado "Bag of Words". A entrada da rede neural, para cada amostra, será um vetor de comprimento do vocabulário, com valores todos zero, com exceção dos tokens que aparecem no texto da amostra. Esse método de codificação é também denominado "One-Hot". Estude o código da classe `IMDBDataset` fazendo experimentos e perguntas ao chatGPT para entender com profundidade esta classe.

Exercício II.1:

II.1.a) Investigue o dataset criado na linha 25. Faça um código que aplique um laço sobre o dataset `train_data` e calcule novamente quantas amostras positivas e negativas do dataset. É esperado que seja um valor balanceado próximo de 100 amostras positivas e 100 amostras negativas.

II.1.b) Calcule também o número médio de palavras codificadas em cada vetor one-hot. Compare este valor com o comprimento médio de cada texto (contado em palavras), conforme calculado no exercício I.1.c. e explique a diferença.

A rede neural será alimentada pelo vetor one-hot (quais suas dimensões) e fará uma predição da probabilidade do texto associado ao one-hot ser uma mensagem positiva.

III - DataLoader

Vamos estudar agora o **Data Loader** da seção III do notebook. Depois que você otimizar o treinamento vai conseguir uma acurácia no dataset de teste de mais de 80% utilizando os 25 mil dados de treino e avaliando nos 25 mil dados de teste. Entretanto, se você colocar o parâmetro `shuffle` na construção do objeto `train_loader` para False, a acurácia não passa de 50% (aleatório). Observação: você não consegue reproduzir estes resultados agora, pois ficará muito lento, mas depois que tiver feito todos os exercícios, inclusive a otimização, você pode fazê-lo e você vai comprovar a importância de embaralhar os dados na divisão em batches, no começo de cada nova época.

Shuffle	Acurácia
True	83%
False	50%

Tabela 1

Estude o método de minimização da Loss pelo gradiente descendente utilizado em redes neurais, utilizando processamento por *batches*.

Esse é um conceito muito importante. Veja no chatGPT qual é a relação da função Loss a ser minimizada no treinamento em função do *batch size*.

Exercícios III.1:

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

Uma das principais vantagens é a eficiência computacional, com o processamento simultâneo de múltiplos dados, o treinamento se torna mais rápido e a utilização de memória é utilizada. Outra vantagem é a melhor qualidade dos gradientes, batches maiores produzem estimativas de gradientes mais estáveis e menos ruidosas, o que leva a atualizações de pesos mais consistentes e uma convergência mais suave do modelo.

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

Embaralhar as amostras dos batches em cada época de treinamento é importante para que o modelo não aprenda padrões específicos da ordem dos dados e garantir que ele generalize melhor. Isso evita correlações indesejadas entre exemplos consecutivos e promove uma maior variabilidade nos gradientes, resultando em atualizações de pesos mais diversificadas e uma convergência mais estável.

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

Porque a ordem dos dados não afeta a avaliação do modelo, como a acurácia mede a razão entre as predições corretas e o dataset completo, a ordem em que os dados são passados para o modelo não importa.

Exercícios III.2

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.

```
num_iterations = 0
for batch in train_loader:
    num_iterations += 1

print(f"Número de iterações no train_loader: {num_iterations}")
```

Número de iterações no train_loader: 196 (dataset completo), 2 (apenas as 200 primeiras amostras).

O número de iterações que o train_loader tem é o número de batches necessários para processar todo o conjunto de treinamento. Este valor pode ser calculado matematicamente dividindo o tamanho do dataset de treinamento pelo tamanho do batch.

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`?

Número de amostras do último batch do train_loader: 40 (dataset completo), 72 (apenas as 200 primeiras amostras).

Sim, o número de amostras do último batch pode ser menor que o `batch_size` se o tamanho total do conjunto de dados não for exatamente divisível pelo `batch_size`.

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.

O valor médio de R encontrado foi 0.5, logo o data loader está entregando batches balanceados.

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 *target* 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 *target* esperado. Desta vez selecione um elemento do batch do train_loader utilizando as funções `next` e `iter`: `batch = next(iter(train_loader))`.

Shape da entrada: [128, 20001]
Tipo de dado da entrada: float32
Shape do target esperado: [128]
Tipo de dado do target esperado: int64

IV - Modelo MLP

A célula da seção **IV - Modelo** é provavelmente uma das mais difíceis de entender, juntamente com a seção **V - Treinamento**, pois são onde aparecem as principais funções do PyTorch.

Iremos utilizar uma rede neural de duas camadas ditas MLP (Multi-Layer Perceptron). São duas camadas lineares, `fc1` e `fc2`. Essas camadas também são denominadas *fully connected* para diferenciar de camadas convolucionais. As camadas são onde estão os parâmetros (*weights*) da rede neural. É importante estudar como estas camadas lineares funcionam, elas são compostas de neurônios que fazem uma média ponderada pelos parâmetros W_i mais uma constante B_i. Esses parâmetros são treinados para minimizar a função de *Loss*. Uma função não linear é colocada entre as camadas lineares. No caso, usamos a função ReLU (*Rectified Linear Unit*).

Para entender o código da célula do Modelo MLP é fundamental conhecer os conceitos de orientação a objetos do Python. O modelo é definido pela classe `OneHotMLP` e é instanciado no objeto `model` na linha 16 que implementa o modelo da rede neural, recebendo uma entrada no formato one-hot e retornando o logito para ser posteriormente convertido em probabilidade do frase ser positiva ou negativa. O método `forward` será chamado automaticamente quando o objeto `model` for usado como função. Esses modelos são projetados para processar um batch de entrada de cada vez no formato devolvido pelo Data Loader visto na seção III (Exercício III.2.d)

Exercícios para experimentar o modelo
(continue fazendo os exercícios com 200 amostras no dataset)

Exercícios IV.1:

IV.1.a) Faça a predição do modelo utilizando um batch do train_loader: extraia um batch do `train_loader`, chame de `(inputs, targets)`, onde `input` é a entrada da rede e `target` é a classe (positiva ou negativa) esperada. 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:

```
logits = model( inputs)
```

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

```python
# Selecionar um batch
batch = next(iter(train_loader))
inputs, targets = batch

# Fazer a predição usando o modelo
logits = model(inputs)

# Aplicar a função sigmoidal para converter os logits em probabilidades
probabilities = torch.sigmoid(logits)

print(f"Logits: {logits.squeeze()}")
print(f"Probabilidades: {probabilities.squeeze()}")
```

```
Logits: tensor([0.0408, 0.0111, 0.0247, 0.0346, 0.0463, 0.0574, 0.0206, 0.0235, 0.0363,
        0.0143, 0.0240, 0.0365, 0.0289, 0.0195, 0.0276, 0.0185, 0.0486, 0.0355,
        0.0425, 0.0440, 0.0214, 0.0228, 0.0425, 0.0346, 0.0567, 0.0423, 0.0159,
        0.0531, 0.0049, 0.0425, 0.0252, 0.0316, 0.0213, 0.0359, 0.0174, 0.0198,
        0.0203, 0.0512, 0.0688, 0.0442, 0.0371, 0.0031, 0.0012, 0.0758, 0.0263,
        0.0325, 0.0262, 0.0628, 0.0163, 0.0440, 0.0433, 0.0393, 0.0165, 0.0131,
        0.0118, 0.0399, 0.0593, 0.0401, 0.0182, 0.0393, 0.0298, 0.0148, 0.0662,
        0.0313, 0.0269, 0.0632, 0.0362, 0.0376, 0.0750, 0.0414, 0.0152, 0.0162,
        0.0199, 0.0442, 0.0272, 0.0197, 0.0360, 0.0605, 0.0384, 0.0420, 0.0229,
        0.0404, 0.0517, 0.0338, 0.0577, 0.0612, 0.0370, 0.0380, 0.0643, 0.0211,
        0.0210, 0.0153, 0.0160, 0.0527, 0.0073, 0.0092, 0.0228, 0.0519, 0.0278,
        0.0336, 0.0606, 0.0867, 0.1260, 0.0350, 0.0258, 0.0214, 0.0255, 0.0242,
        0.0305, 0.0572, 0.0465, 0.0125, 0.0344, 0.0406, 0.0225, 0.0118, 0.0155,
        0.0613, 0.0209, 0.0218, 0.0588, 0.0187, 0.0392, 0.0578, 0.0325, 0.0394,
        0.0415, 0.0460], grad_fn=<SqueezeBackward0>)
Probabilidades: tensor([0.5102, 0.5028, 0.5062, 0.5087, 0.5116, 0.5143, 0.5052, 0.5059, 0.5091,
        0.5036, 0.5060, 0.5091, 0.5072, 0.5049, 0.5069, 0.5046, 0.5122, 0.5089,
        0.5106, 0.5110, 0.5054, 0.5057, 0.5106, 0.5086, 0.5142, 0.5106, 0.5040,
        0.5133, 0.5012, 0.5106, 0.5063, 0.5079, 0.5053, 0.5090, 0.5043, 0.5050,
        0.5051, 0.5128, 0.5172, 0.5110, 0.5093, 0.5008, 0.5003, 0.5189, 0.5066,
        0.5081, 0.5066, 0.5157, 0.5041, 0.5110, 0.5108, 0.5098, 0.5041, 0.5033,
        0.5029, 0.5100, 0.5148, 0.5100, 0.5046, 0.5098, 0.5075, 0.5037, 0.5166,
        0.5078, 0.5067, 0.5158, 0.5091, 0.5094, 0.5187, 0.5103, 0.5038, 0.5041,
        0.5050, 0.5111, 0.5068, 0.5049, 0.5090, 0.5151, 0.5096, 0.5105, 0.5057,
        0.5101, 0.5129, 0.5085, 0.5144, 0.5153, 0.5092, 0.5095, 0.5161, 0.5053,
        0.5053, 0.5038, 0.5040, 0.5132, 0.5018, 0.5023, 0.5057, 0.5130, 0.5070,
        0.5084, 0.5152, 0.5217, 0.5315, 0.5087, 0.5064, 0.5053, 0.5064, 0.5060,
        0.5076, 0.5143, 0.5116, 0.5031, 0.5086, 0.5102, 0.5056, 0.5029, 0.5039,
        0.5153, 0.5052, 0.5054, 0.5147, 0.5047, 0.5098, 0.5144, 0.5081, 0.5099,
        0.5104, 0.5115], grad_fn=<SqueezeBackward0>)
```

IV.1.b) Agora, treine a rede executando o notebook todo e verifique se a acurácia. 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 target 1 e se for menor que 0.5, o target 0. Observe isso que é feito na linha 11 da seção VI - Avaliação.
Calcule então a acurácia do modelo treinado no primeiro batch dos dados de teste e mostre o código para calcular essa acurácia do primeiro batch e a acurácia obtida:

Acurácia do primeiro batch dos dados de teste: 1

```python
batch = next(iter(test_loader))
inputs, targets = batch
```

```
inputs, targets = inputs.to(device), targets.to(device)
logits = model(inputs)
probabilities = torch.sigmoid(logits)
predicted = torch.round(probabilities)
accuracy = (predicted == targets).float().mean()

print(f"Acurácia do primeiro batch dos dados de teste:
{accuracy.item()}")
```

IV.1.c) Número de parâmetros do modelo

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):

layer	`fc1`		`fc2`		TOTAL
	weight	bias	weight	bias	
size	4000200	200	200	1	4000601

Tabela 2

V - Treinamento

Agora vamos entrar na principal seção do notebook que minimiza a Loss para ajustar os pesos do modelo.

Cálculo da Loss

A *Loss* é uma comparação entre a saída do modelo e o label (target). A *Loss* mais utilizada para problemas de classificação é a Entropia Cruzada. A equação da entropia cruzada para o caso binário (2 classes: 0 ou 1; True ou False) é dada por:

$$L(y, \hat{y}) = -\frac{1}{N} \sum_{i=1}^{N} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

Onde:

- L é a função de perda de entropia cruzada binária.
- N é o número total de amostras.
- y_i é o rótulo real da i-ésima amostra, com valor 0 ou 1.
- $\hat{y}_i$ é a probabilidade predita pelo modelo de que a i-ésima amostra pertença à classe 1.

Muitas vezes chamamos y_i de target e $\widehat{y_i}$ de prob.

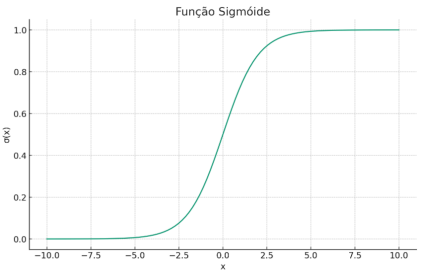
Quando a Loss é zero, significa que o modelo está predizendo tanto as amostras positivas como as amostras negativas com probabilidade de 100%. O objetivo é otimizar o modelo para conseguir minimizar a *Loss* ao máximo.

A rede neural é o nosso modelo que recebe a entrada com um batch de amostras e retorna um batch de logitos.

```
logits = model( inputs)
```

para converter o logito em probabilidade, utiliza-se a função sigmóide que é dada pela equação:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



Assim, o código pytorch para estimarmos a probabilidade de um texto codificado no formato one-hot na variável input pode ser:

```
probs = torch.sigmoid(logits)
```

Atenção: observe que esses comandos estão processando todas as amostras no batch, que nesse notebook tem 128 amostras no batch size.

Exercícios V.1:

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.

Considerando que o modelo apenas inicializado tende a prever uma probabilidade de 0,5 para ambas as classes, temos:

Para y = 1:

$$L = -1[1 \cdot log(0.5) + (1 - 1) \cdot log(1 - 0.5)] = -log(0.5) = log(2) = 0.6931$$

Para y = 0:

$$L = -1[0 \cdot log(0.5) + (1 - 0) \cdot log(1 - 0.5)] = -log(0.5) = log(2) = 0.6931$$

Portanto, o valor teórico da Loss quando o modelo não está treinado é 0,6931.

V.1.b) Utilize as amostras do primeiro batch: `(input,target) = next(iter(train_loader))` e calcule o valor da Loss utilizando apenas as operações da equação fornecida anteriormente utilizando o pytorch (operações de soma, subtração, multiplicação, divisão e log). Verifique se este valor é próximo do valor teórico do exercício anterior.

O valor obtido foi 0.7037867307662964, muito próximo do valor teórico.

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.

O resultado obtido foi de 0.7037867307662964, exatamente o mesmo obtido no exercício anterior.

V.1.d) Repita o mesmo exercício, porém agora usando a classe `nn.BCEWithLogitsLoss`, que é a opção utilizada no notebook. Observe que com essa classe o cálculo da Loss é feito com o logito sem precisar calcular a probabilidade. O resultado da *Loss* deve igualar aos resultados anteriores.

O resultado obtido foi de 0.7037867307662964, exatamente o mesmo obtido no exercício anterior.

Minimização da Loss pelo gradiente descendente

Estude o método do gradiente descendente para minimizar uma função. Como curiosidade, pergunte ao chatGPT quando este método de minimização foi usado pela primeira vez.
Aproveite e peça para ele explicar o método de uma maneira bem simples e ilustrativa.
Peça para ele explicar qual é a forma moderna de se calcular computacionalmente o gradiente de uma função.
Finalmente peça para ele explicar as linhas 3, 6, e (20, 21 e 22) do laço de treinamento.

Exercícios V.2:

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.
Atenção 1: A Loss antes da época 0 deve ser calculada em todas as amostras. (Neste caso ainda as 200 amostras)
Atenção 2: Como temos apenas 200 amostras é bem possível que a Loss não caia durante o treinamento, por falta de dados. Posteriormente, iremos re-executar este treino com todas as amostras de treino e daí a Loss irá cair consistentemente.

Loss antes da época 0	0.6644
Loss após época 0	0.6641
Loss após época 5	0.6575

V.2.b) Execute a célula de treinamento por uma segunda vez e observe que a Loss continua a partir do valor que estava antes, após a época 5. 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 começar o treinamento novamente do modelo aleatório é necessário inicializar o modelo executando a célula da seção IV - Modelo, em específico a linha `model = OneHotMLP(vocab_size)`.

V.2.c) Coloque um print logo após o cálculo da Loss (linha 18) e verifique que o valor da Loss impressa não é a Loss média da época, mas apenas a Loss do último batch. Corrija este código para que a Loss impressa no final de cada época seja de fato a Loss da época, isto é, considerando todos os dados da época, de todos os batches. Verifique que a equação da Loss é uma média das entropias cruzadas de cada amostra - equação Loss no início desta seção.

Mostre o código corrigido:

```
23 # Training loop
24 num_epochs = 5
25 for epoch in range(num_epochs):
26     start_time = time.time()  # Start time of the epoch
27     model.train()
28     total_epoch_loss = 0.0 # Adicionado
29     for inputs, targets in train_loader:
30         inputs = inputs.to(device)
31         targets = targets.to(device)
32         # Forward pass
33         logits = model(inputs)
34         loss = criterion(logits.squeeze(), targets.float())
35         # Backward and optimize
36         optimizer.zero_grad()
37         loss.backward()
38         optimizer.step()
39 
40         # Acumula a Loss do batch
41         total_epoch_loss += loss.item() * inputs.size(0) # Adicionado
42 
43     # Calcula a Loss média da época
44     average_epoch_loss = total_epoch_loss / len(train_loader.dataset) # Adicionado
45     end_time = time.time()  # End time of the epoch
46     epoch_duration = end_time - start_time  # Duration of epoch
47 
48     print(f'Epoch [{epoch+1}/{num_epochs}], \
49             Loss: {average_epoch_loss:.4f}, \
50             Elapsed Time: {epoch_duration:.2f} sec') # Modificado
```

VI - Avaliação

Observe que o módulo de avaliação utiliza o `test_loader` que foi carregado do dataset IMDB especialmente preparado para fazer a avaliação.

Exercícios VI.1:

VI.1.a) Calcule o número de amostras que está sendo considerado na seção de avaliação. Observe que ainda estamos usando 200 amostras para o dataset de treino e 200 amostras do dataset de teste.

O número de amostras que está sendo considerado na seção de avaliação é 200.

```
print(f'Total number of test samples: {total}')
```

VI.1.b) Explique o que faz os comandos `model.eval()` e `with torch.no_grad()`.

O comando `model.eval()` configura o modelo para o modo de avaliação, desativando comportamentos específicos de treinamento, como dropout e normalização por batch, para garantir previsões consistentes. Já o bloco `with torch.no_grad()` desativa o cálculo de gradientes, economizando memória e processamento durante a inferência, já que não são necessários para a atualização dos pesos.

VI.1.c) Qual é uma forma mais simples de calcular a classe predita na linha 11, sem a necessidade de usar a função torch.sigmoid ou qualquer outra função do pytorch? (Dica: analise todos os comandos da linha 11 e veja qual é a simplificação possível e implemente-a e verifique se o valor da acurácia continua o mesmo)

```
predicted_simples = (logits.squeeze() > 0).float()
```

Isso cria um tensor booleano onde os valores são positivos (1.0) se o logit for maior que 0 e negativos (0.0) caso contrário.

Com isso, a acurácia continua a mesma.

VII - Aumentando a eficiência do treinamento

Agora que você já conhece como o notebook está organizado e conhece suas seções, iremos fazer várias otimizações para executá-lo mais rápido e conseguir que a convergência do treino seja mais rápida e que consiga uma acurácia maior que 80% sem a necessidade de mudar o número de épocas nem a forma fazer o gradiente descendente usando o SGD.

Agora, volte a ter o dataset de treino usando as 25 mil amostras.

O código do notebook está preparado para executar tanto com ambiente usando CPU como com GPU, entretanto o ganho de velocidade está sendo reduzido de 45 segundos (sem GPU) para 29 segundos (com GPU) que é um ganho muito aquém do esperado, que seria um speedup entre 7 e 11 vezes dependendo da aplicação. Vamos entender a razão desta baixa eficiência e corrigir o problema.

A GPU é utilizada durante o treinamento do modelo, onde é utilizada a técnica de minimização da *Loss* utilizando o gradiente descendente. Isso ocorre na segunda célula do "**V - Laço de Treinamento**". Iremos analisar os detalhes mais à frente, para por enquanto basta entender onde a GPU é utilizada. A linha 17 é onde o modelo está fazendo a predição (passo *forward*), dado a entrada, calcula a saída da rede (muitas vezes chamado de logito) e o cálculo da loss está sendo feito na linha seguinte e o cálculo do gradiente ocorre na linha 21 e a linha 22 é onde ocorre o ajuste dos parâmetros (*weights*) da rede neural fazendo ela minimizar a Loss. Esse é o processo que é mais demorado e onde a GPU tem muitos ganhos, pois envolve praticamente apenas multiplicação de matrizes. Existem apenas 3 linhas que controlam o uso da GPU que servem para colocar o modelo, a entrada e a saída esperada (targets) na GPU: linhas 3, 14 e 15, respectivamente.

Exercício VII.1:

Com o notebook configurado para GPU T4, meça o tempo de dois laços dentro do `for` da linha 13 (coloque um break após dois laços) e determine quanto demora para o passo de *forward* (linhas 14 a 18), para o backward (linhas 20, 21 e 22) e o tempo total de um laço. Faça as contas e identifique o trecho que é mais demorado.

VII.1.a)

```
13 for inputs, targets in train_loader:
14        inputs = inputs.to(device)
15        targets = targets.to(device)
16        # Forward pass
17        logits = model(inputs)
18        loss = criterion(logits.squeeze(), targets.float())
19        # Backward and optimize
20        optimizer.zero_grad()
21        loss.backward()
22        optimizer.step()
```

Tempo de execução	
linha 13	167.943001 ms
linha 14	2.244949 ms
linha 15	0.051379 ms

linha 17	0.375628 ms
linha 18	0.254273 ms
linha 20	0.288367 ms
linha 21	0.712991 ms
linha 22	0.302196 ms

Conclusão: A demora principal está na execução da linha: 13

VII.1.b) Trecho que precisa ser otimizado. (Esse é um problema bem mais difícil) A dica aqui é que precisa muito de conceitos de iterador e programação orientada a objetos, variável de instância, variável de classe, variável local.

O trecho que precisa ser otimizado é:

```
    def __getitem__(self, idx):
        sample = self.data[idx]
        target = sample["label"]
        line = sample["text"]
        target = 1 if target == 1 else 0

        # one-hot encoding
        X = torch.zeros(len(self.vocab) + 1)
        for word in encode_sentence(line, self.vocab):
            X[word] = 1

        return X, torch.tensor(target)
```

Pois realizar o one-hot encoding e a leitura dos dados a cada iteração do DataLoader é muito custoso.

VII.1.c) Otimize o código e explique aqui.

```
class IMDBDataset(Dataset):
    def __init__(self, split, vocab):
        self.data = load_dataset("stanfordnlp/imdb", split=split)
        self.vocab = vocab
        # Pré-processar os dados para one-hot encoding
        self.encoded_data = []
        for sample in self.data:
            target = sample["label"]
            line = sample["text"]
            target = 1 if target == 1 else 0
```

```
            # one-hot encoding
            X = torch.zeros(len(self.vocab) + 1)
            for word in encode_sentence(line, self.vocab):
                X[word] = 1
            self.encoded_data.append((X, torch.tensor(target)))

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

    def __getitem__(self, idx):
        return self.encoded_data[idx]
```

Para otimizar o código, a abordagem principal envolve o pré-processamento dos dados durante a inicialização da classe IMDBDataset. Em vez de realizar o one-hot encoding e a leitura dos dados a cada iteração do DataLoader com a função `__getitem__`, os dados são pré-processados uma vez e armazenados em memória. Usamos iteradores para acessar os dados de forma eficiente durante o treinamento, eliminando a necessidade de transformações repetitivas.

Após esta otimização, **é esperado que o tempo de processamento de cada época caia tanto para execução em CPU (da ordem de 15 segundos por época) como para GPU T4 (da ordem de 1 a 2 segundos por época).** Isso utilizando as 25 mil amostras do dataset inteiro do IMDB de treino.

Atenção: Se não conseguir atingir esse objetivo, procure árduamente entender com maiores detalhes o código. Esse exercício é fundamental para poder acompanhar o curso durante o semestre.

Agora que a execução está bem mais otimizada em tempos de execução, modifique o notebook para que o número de amostras de treino seja o dataset inteiro (25 mil amostras de treino). Todos os exercícios finais a seguir devem usar a totalidade do dataset do IMDB.

Vamos agora analisar um outro fator importante que é a escolha do LR (*Learning Rate*)

Escolhendo um bom valor de LR

Exercício VII.2:

O Learning Rate (LR) é um dos parâmetros mais importantes para ser escolhido na otimização do gradiente descendente.

Peça ao chatGPT para explicar como funciona a equação do gradiente descendenteQual é o problema de escolher um LR muito baixo? E de escolher um LR muito alto?

VII.2.a) Mostre a equação utilizada no gradiente descendente e qual é o papel do LR no ajuste dos parâmetros (*weights*) do modelo da rede neural.

A equação utilizada no gradiente descendente é dada por:

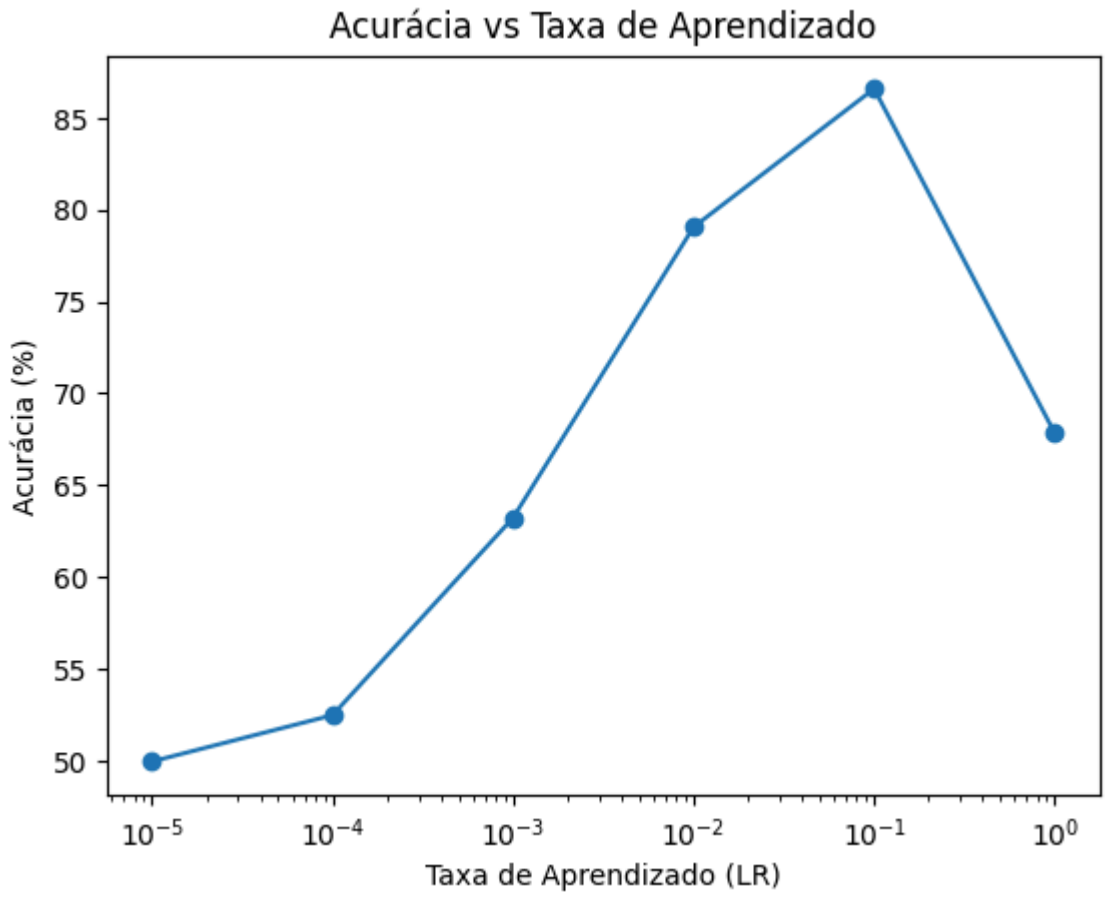
$$\theta_{i+1} = \theta_i - LR \cdot \nabla_\theta J(\theta)$$

onde θ são os parâmetros do modelo, LR o Learning Rate e $\nabla_\theta J(\theta)$ é o gradiente da função de perda em relação aos parâmetros.

O papel do LR é controlar a magnitude das atualizações dos parâmetros, um valor alto pode levar a atualizações grandes, resultando em uma convergência rápida, porém instável, com o risco de o modelo saltar sobre o mínimo da função de perda. Por outro lado, um valor baixo faz com que as atualizações sejam pequenas, promovendo uma convergência mais estável, mas muito lenta, e em alguns casos pode levar à estagnação. A taxa de aprendizado afeta diretamente a estabilidade e a velocidade do treinamento.

Faça uma melhor escolha do LR, analisando o valor da acurácia no conjunto de teste, medindo 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. Atenção, mantenha o número de épocas como 5 e varie o LR numa faixa que o LR seja bem baixo e bem alto, reproduzindo os efeitos da resposta de usar LR muito baixo ou muito alto.

VII.2.b) Gráfico Acurácia vs LR



VII.2.c) Valor ótimo do LR (para isso é desejável que o LR ótimo que forneça maior acurácia no conjunto de testes seja maior que usar um LR menor e um LR maior que o LR ótimo.)

O valor ótimo do LR encontrado foi de 0.1 fornecendo acurácia de 86.57%.

Otimizando o tokenizador

Agora que a convergência da Loss está melhor, vamos experimentar ajustar os parâmetros do tokenizador, isto é, como as palavras estão codificadas em tokens.

Observe novamente o `vocab` criado na parte **I - Vocabulário e Tokenização**. Perceba como as pontuações estão influenciando nos tokens criados e como o uso de letras maiúsculas e minúsculas também podem atrapalhar a consistência dos tokenizador em representar o significado semântico das palavras. Experimente rodar o `encode_sentence` com frases que tenham pontuações e letras maiúsculas e minúsculas. Baseado nessas informações, procure melhorar a forma de tokenizar o dataset.

Exercício VII.3:

Melhore 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.

VII.3.a) Mostre os trechos modificados para este novo tokenizador, tanto na seção I - Vocabulário, como na seção II - Dataset.

Código de pré-processamento:
I - Vocabulário:

```
import string
def preprocess_text(text):
    # Converter para minúsculas
    text = text.lower()
    # Remover pontuações
    text = text.translate(str.maketrans('', '', string.punctuation))
    return text

train_dataset = load_dataset("stanfordnlp/imdb", split="train")

def encode_sentence(sentence, vocab):
    sentence = preprocess_text(sentence) # Modificação
    return [vocab.get(word, 0) for word in sentence.split()] # 0 for
OOV

encode_sentence("I like Pizza.", vocab)

vocab_size = 20000

counter = Counter()
for sample in train_dataset: #
    preprocessed_text = preprocess_text(sample["text"])  #
Pré-processar o texto
    counter.update(preprocessed_text.split())  # Atualizar o contador
com tokens

# create a vocabulary of the 20000 most frequent tokens
most_frequent_words = sorted(counter, key=counter.get,
reverse=True)[:vocab_size]
vocab = {word: i for i, word in enumerate(most_frequent_words, 1)}
vocab_size = len(vocab)
```

II - Dataset:

Alteração no Dataset: Não houve.

VII.3.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.

tokenizador original	566141
tokenizador novo	214473

O resultado obtido foi um número menor de tokens desconhecidos, pois o novo tokenizador faz a remoção de pontuações, gerando uma representação mais consistente das palavras, levando a uma melhor correspondência com o vocabulário.

VII.3.c) Execute agora no notebook inteiro com o novo tokenizador e anote o novo valor da acurácia obtido com a melhoria do tokenizador, treinando o modelo por 5 épocas.

Acurácia nova: 87.40%
Acurácia antiga: 86.57%