

PS IA-024 1S2024 FEEC-UNICAMP

Rian Radeck - 187793

18 de fevereiro de 2024

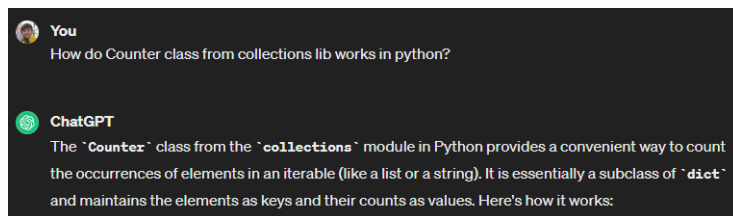
1 Vocabulário e Tokenização

Vamos iniciar identificando como o dado está sendo apresentado. Para isso imprimimos os dados do conjunto e observamos que eles estão organizados em tuplas com a label (1 para negativo e 2 para positivo) e a review, respectivamente.

```
print([x[0] for x in list(IMDB(split='train'))[-5:]])  
print([x[0] for x in list(IMDB(split='train'))[:5:]])  
  
slabel = set()  
counter = Counter()  
for (label, line) in list(IMDB(split='train')):  
    counter.update(line.split())  
    slabel.add(label)  
print(slabel)
```

Com essa análise foi possível entender melhor como nosso dataset deve ser manipulado. Percebemos também que os dados estão ordenados pela label, com todas as reviews negativas antes das positivas.

O último passo para a análise inicial desta célula foi entender sobre a classe Counter. Para isso, perguntamos ao chatGPT com o seguinte prompt



Em suma, a classe Counter armazenará a frequência de cada palavra que aparece no dataset. Um exemplo simples para demonstrar o funcionamento da classe pode ser observado com a seguinte linha de código

```
print(counter.get("movie"))
```

que retorna a quantidade de vezes que a palavra “movie” aparece no dataset, no caso 30506. Agora vamos aos exercícios dessa seção.

1.1 Exercício I.1

Nosso objetivo é utilizar um segundo contador para calcular o número de amostras positivas e amostras negativas, além de calcular o comprimento médio do texto em número de palavras dos textos das amostras. Para tanto, as seguintes modificações foram incorporadas ao código

```
counter = Counter()
label_counter = Counter()
total_words = 0
for (label, line) in list(IMDB(split='train')):
    words_in_line = line.split()
    total_words += len(words_in_line)
    counter.update(words_in_line)
    label_counter.update([label])

number_of_reviews = len(list(IMDB(split='train')))
avg_words_in_review = total_words / number_of_reviews
```

Os resultados obtidos foram:

```
Negative labels: 12500
Positive labels: 12500
Total labels: 25000
Average number of words in a review 233.7872
```

Em seguida, o código foi comentado pelo chatGPT. Essa prática será comum para melhor documentarmos o código durante todo o notebook, razão pela qual omitiremos esse passo nas seções seguintes do relatório. Omitiremos também os comentários nos prints de código desse relatório para fins de legibilidade.

1.2 Exercício I.2

O objetivo deste exercício é encontrar as cinco palavras mais frequentes do vocabulário e as cinco palavras menos frequentes. Também devemos encontrar qual é o código do token que está sendo utilizado quando a palavra não está no vocabulário e calcularmos quantos tokens das frases do conjunto de treinamento que não estão no vocabulário.

Primeiro vamos entender como os dados foram manipulados pelas funções (sorted e enumerate). Sabemos que a função sorted ordena o contador de forma decrescente e o slicing considera apenas as 20000 palavras mais frequentes. Por sua vez, a função enumerate ranqueia as palavras com um número entre 1 e

20000 inclusive, onde quanto menor o ranking, mais frequente é aquela palavra no dataset.

Agora podemos concluir que `most_frequent_words` é uma lista com as 20000 palavras mais frequentes, ordenadas da mais frequente a menos frequente e o dicionário `vocab` codifica (tokeniza) cada uma dessas palavras. Observe a lista e o dicionário a seguir

```
['the', 'a', 'and', 'of', 'to', 'is', 'in', 'I', 'that', 'this',  
{ 'the': 1, 'a': 2, 'and': 3, 'of': 4, 'to': 5, 'is': 6, 'in': 7,
```

Para descobrir o código do token que está sendo utilizado quando a palavra não está no vocabulário vamos analisar a função `encode_sentence`. Percebe-se que ela passa por cada palavra de uma frase (que recebeu como parâmetro) e retorna uma lista com o valor correspondente de cada uma no dicionário `vocab`, ou seja, os tokens de cada palavra. Caso a palavra não esteja entre as 20000 mais frequentes (não está no dicionário), o valor que representará aquela palavra será 0 (unknown token), pois é o valor padrão do `get` do vocabulário.

Com as observações acima, vamos utilizar o seguinte código para responder as perguntas

```
# Code to calculate 5 most frequent words, 5 least frequent words and number of words in dataset without encoding.  
  
print("5 most frequent words:", most_frequent_words[:5])  
print("5 least frequent words:", most_frequent_words[-5:])  
number_of_tokened_words = np.sum([(freq if encode_sentence(word, vocab) != [0] else 0) for (word, freq) in counter.items()])  
print("Number of words with a token:", number_of_tokened_words)  
print("Number of words without a token:", total_words - number_of_tokened_words)  
  
5 most frequent words: ['the', 'a', 'and', 'of', 'to']  
5 least frequent words: ['age-old', 'place!', 'Bros', 'tossing', 'nation,']  
Number of words with a token: 5278539  
Number of words without a token: 566141
```

Portanto,

- As 5 palavras mais frquentes: the, a, and, of e to.
- As 5 palavras menos frquentes: age-old, place!, Bros, tossing e nation,.
- Número de palavras com token 0 (unknow token): 566,141

1.3 Exercício 1.3

Vamos iniciar o terceiro e último ponto desta seção de vocabulário e tokenização. O objetivo deste exercício é descobrir por qual razão o modelo preditivo consegue acertar 100% das amostras de teste do dataset com apenas as primeiras 200 amostras. Também deveremos modificar a forma de selecionar 200 amostras do dataset, para garantir que ele continue balanceado

Após selecionar apenas as primeiras 200 amostras do dataset, o modelo treinou e testou apenas com amostras negativas, pois nossos dados estão ordenados pela label, como pudemos ver na análise inicial da seção, o que justifica a acurácia de 100%.

```
for (label, line) in list(IMDB(split='train'))[:200]:
    Negative labels: 200
    Positive labels: None
```

Para resolver esse problema deve-se aleatorizar a escolha de amostras. Dessa forma, o seguinte código foi adicionado no início da célula I,

```
# Reduce dataset if needed
reduce_dataset = True
reduced_dataset_size = 200

dataset_train = list(IMDB(split='train')).copy()
if reduce_dataset:
    np.random.shuffle(dataset_train)
    dataset_train = dataset_train[:reduced_dataset_size]
```

impondo modificações necessárias ao longo do código, como por exemplo, no construtor do data set

```
class IMDBDataset(Dataset):
    def __init__(self, split, vocab):

        self.data = list(IMDB(split=split))
        if reduce_dataset:
            np.random.shuffle(self.data)
            self.data = self.data[:reduced_dataset_size]
        self.vocab = vocab
```

2 Dataset

Iniciaremos esta seção explorando a classe `IMDBDataset`, com o auxílio do chatGPT onde for necessário.

Analisando o construtor da classe, percebe-se que o dataset que será utilizado pelo modelo é construído a partir do dataset IMDB.

Outra observação importante é sobre a função `__getitem__`, que é responsável por adaptar os dados originais, trocando labels positivos de 2 para 0 e construindo o vetor correspondente àquela determinada review com o One-hot Encoding (função `encode_sentence()`).

Essa técnica criará um vetor de 20001 espaços (a quantidade de tokens disponíveis para as palavras mais frequentes, juntamente com o único token para as demais palavras), que serão os neurônios da camada de entrada de nossa rede neural.

Os neurônios estão inicialmente todos desligados (têm valor 0) e serão ligados aqueles que tenham mesmo valor da codificação de alguma palavra daquela review. Por exemplo, observe a figura abaixo e perceba que a frase “I like Pizza” ligaria os neurônios 8, 35 e 0 respectivamente (considerando o dataset completo).

```
encode_sentence("I like Pizza", vocab)
[8, 35, 0]
```

2.1 Exercício II.1

No primeiro exercício dessa seção, o objetivo é calcular quantas amostras positivas e negativas existem no dataset reduzido, além de encontrar o número médio de palavras codificadas em cada vetor one-hot.

Iniciando com a classe Counter, os resultados para a quantidade de labels positivas e negativas podem ser obtidos com o código a seguir

```
# Counting the number of positive and negative labels
label_counter = Counter()
for label, line in train_data.data:
    label_counter.update([label])
print(f"Negative labels: {label_counter.get(1)}\nPositive labels: {label_counter.get(2)}")
```

Com o objetivo de contar a quantidade de 1's em um `np.array`, o chat-GPT sugeriu utilizar a função `np.count_nonzero`, que contribuiu na tarefa de encontrar o número médio de palavras codificadas em cada vetor one-hot.

```
# Counting the number of positive and negative labels
# Calculating number of ON neurons
label_counter = Counter()
idx = 0
total_on_words = 0
for label, line in train_data.data:
    label_counter.update([label])

    total_on_words += np.count_nonzero(train_data[idx][0] == 1)
    idx += 1
average_on_words_by_review = total_on_words / len(train_data)
print(f"Negative labels: {label_counter.get(1)}")
print(f"Positive labels: {label_counter.get(2)}")
print(f"Average ON neurons: {average_on_words_by_review}")

Negative labels: 12500
Positive labels: 12500
Average ON neurons: 133.09548
```

Como observado, a diferença é bem grande, de 233 palavras em média por review para 133 neurônios ligados em média por vetor, o que é esperado já que uma palavra pode aparecer diversas vezes em uma review.

Por exemplo, a seguinte review “I like you and you like me” possui 7 palavras porém apenas 5 neurônios seriam ligados (assumindo que todas as palavras tem uma codificação no vocabulário). Logo, a quantidade de neurônios ligados em um vetor que representa uma review, será sempre menor ou igual ao número de palavras naquela review.

2.2 Exercício II.2

O objetivo desse exercício é calcular os tempos de Forward e Backward do laço de treinamento e otimizar o código.

Para calcular os tempos de Forward e Backward dos laços, foram feitas as seguintes modificações na seção de treinamento do modelo

```
loops = 0
for inputs, labels in train_loader:
    loops += 1
    if loops == 3:
        break
    loop_start_time = time.time()
    inputs = inputs.to(device)
    labels = labels.to(device)
    # Forward pass
    outputs = model(inputs)
    loss = criterion(outputs.squeeze(), labels.float())
    end_forward = time.time()
    # Backward and optimize
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    end_backward = time.time()
    print(f"Total time: {(end_backward - loop_start_time) * 1000:.2f}ms")
    print(f"Forward time: {(end_forward - loop_start_time) * 1000:.2f}ms")
    print(f"Backward time: {(end_backward - end_forward) * 1000:.2f}ms")
```

```
Total time: 3.93ms
Forward time: 2.82ms
Backward time: 1.11ms
Total time: 4.00ms
Forward time: 2.88ms
Backward time: 1.12ms
```

Observe que o passo de Forward é quase 3 vezes mais demorado que o passo de Backward. Note também que o contador da seção de Forward inicia antes do envio de dados para o device (GPU).

Fazendo experimentos com o contador iniciando após o envio de dados para o device, observa-se uma queda drástica na contagem (passa a ser algo em torno de 0.5ms).

Portanto se todos os batches de dados estiverem no device, o passo de treinamento será acelerado consideravelmente.

```
# Taking data to device is taking 28s
# This will reduce each epoch training to 0.6s

# Taking data each epoch is taking 31s per epoch

on_device_inputs = []
on_device_labels = []
for inputs, labels in train_loader:
    on_device_inputs.append(inputs.to(device))
    on_device_labels.append(labels.to(device))

on_device_train_loader = [(on_device_inputs[i], on_device_labels[i]) for i in range(len(train_loader))]
```

O código acima garante que todos os dados necessários para o treinamento estarão no device para a execução das épocas.

```
# Training loop
num_epochs = 5
for epoch in range(num_epochs):
    start_time = time.time() # Start time of the epoch
    model.train()
    for inputs, labels in on_device_train_loader:
        # Forward pass
        outputs = model(inputs)
        loss = criterion(outputs.squeeze(), labels.float())
        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Epoch [1/5],	Loss: 0.6735,	Elapsed Time: 0.22 sec
Epoch [2/5],	Loss: 0.6714,	Elapsed Time: 0.21 sec
Epoch [3/5],	Loss: 0.6692,	Elapsed Time: 0.22 sec
Epoch [4/5],	Loss: 0.6669,	Elapsed Time: 0.22 sec
Epoch [5/5],	Loss: 0.6645,	Elapsed Time: 0.30 sec

Com as modificações necessárias foi possível reduzir o tempo de treino de cada época em aproximadamente 100 vezes.

Existe uma otimização para o tempo de execução no passo de Backward, que seria o cálculo do `optimizer` apenas no final de cada batch. Essa otimização piora um pouco o aprendizado do modelo, já que torna o gradiente descendente menos preciso.

Após testes com a otimização de Backward, conclui-se que o ganho em tempo de execução é quase que irrelevante, portanto essa modificação não entrará em vigor.

2.3 Exercício II.3

Os objetivos desse exercício são: considerando o conjunto de teste, fazer um gráfico de acurácia vs learning rate e escolher o learning rate que forneça a maior acurácia possível, além de mostrar a equação utilizada no gradiente descendente e qual é o papel do learning rate no ajuste dos parâmetros (weights) do modelo da rede neural.

Para fazer um gráfico LR x Acurácia os trechos do código que fazem o treinamento e a avaliação foram colocados em funções, onde a função de treino instancia um modelo novo, o treina com um LR fornecido como parâmetro e o retorna, enquanto a função de eval recebe um modelo como parâmetro e retorna a acurácia dele.

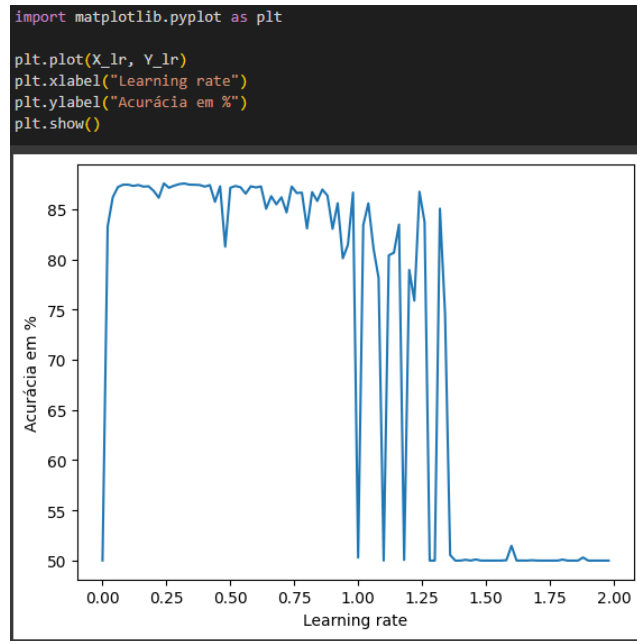
Com as modificações em prática, o seguinte código testa vários learning rates e guarda a acurácia do modelo para cada um deles

```

X_lr = []
Y_lr = []
for lr in np.arange(0, 2, 0.02):
    X_lr.append(lr)
    model = train(lr)
    Y_lr.append(eval(model))

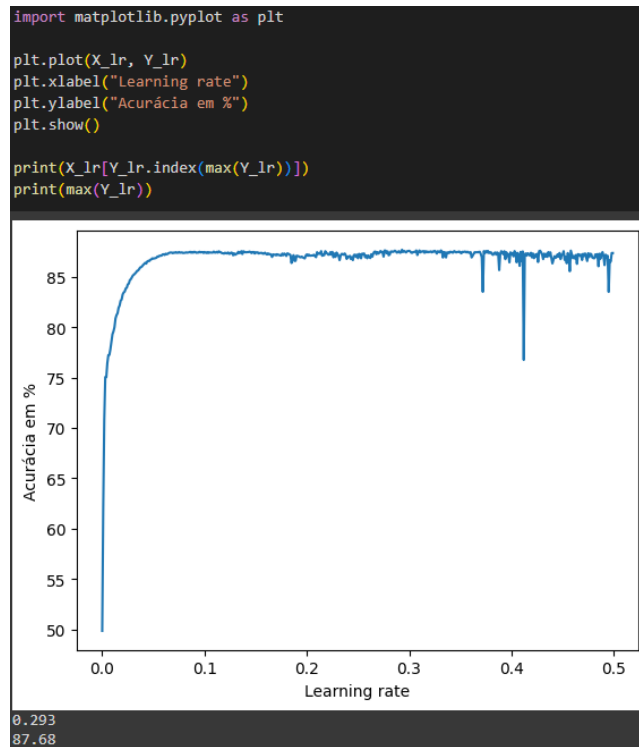
```

Após, os dados são plotados com o seguinte código para obter o gráfico



Analisando o gráfico é possível ver que valores acima de 0.5 causam muita perturbação no aprendizado do modelo e a máxima acurácia parece estar entre 0 e 0.5.

Com limites do `np.arange` entre 0 e 0.5 e saltos de 0.001 conseguimos encontrar a learning rate que nos dá a maior acurácia de 87.68 % em 0.293



A equação utilizada no gradiente descendente para atualizar os parâmetros (pesos) de um modelo em uma rede neural é:

$$W_{i+1} = W_i - \eta \cdot \nabla J(W_i)$$

Onde

- W_i são os parâmetros (pesos) na iteração atual
- η é o learning rate
- $\nabla J(W_i)$ é o gradiente da função de perda (loss)

Sendo assim, o learning rate determina o quão grande será o ajuste dos pesos na direção oposta ao gradiente. Se for um ajuste muito pequeno, o modelo pode demorar demais para convergir ou até mesmo convergir em um mínimo local da função de perda; se for muito grande, pode nunca convergir (overshooting) e como consequência o modelo não aprende nunca.

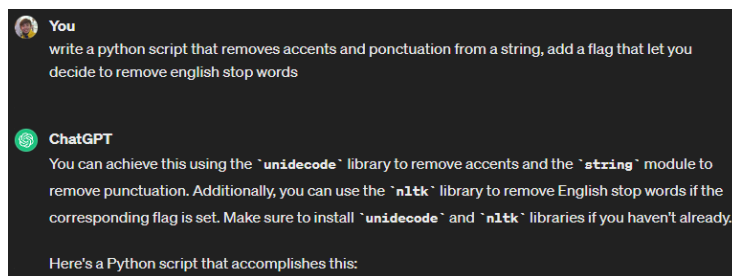
2.4 Otimizando o tokenizador

Nessa seção vamos discutir a coleta de tokens e temos como objetivo melhorar a forma de tokenizar, isto é, pré-processar o dataset de modo que a cod-

ificação seja indiferente quando tratar de palavras escritas com maiúsculas ou minúsculas, além de sofrer pouca influência pelas pontuações.

Podemos observar que os tokens estão sendo coletados de maneira muito simples, apenas com um split em uma frase, o que deixa várias palavras com pontuação e letras maiúsculas e minúsculas sendo tratadas como palavras diferentes. Não é o caso da língua inglesa, mas as palavras poderiam ter acentos. Existem também as *stop words*, que geralmente não agregam muito ao sentido da frase.

Para isso a ajuda do chatGPT foi obtida com o seguinte prompt e resposta



O script gerado foi colocado em uma célula separada no início do notebook com algumas modificações para as palavras estarem sempre em lower case.

O seguinte código foi adicionado a célula de tokenização

```
for i in range(len(dataset_train)):
    dataset_train[i] = dataset_train[i][0], preprocess_text(dataset_train[i][1], True)
```

e foram obtidos os seguintes resultados

Processamento	Média de palavras	Palavras sem token	Acurácia
Nenhum	233	566141	87.68%
Com stop words	233	213754	86.11%
Sem stop words	124	212309	85.83%

Pode-se concluir que a acurácia é quase a mesma nas três situações. Essa pequena diferença pode ter relação com a quantidade de palavras sem token, já que mais palavras que provavelmente não tenham tanto sentido para a análise sentimental estão sendo consideradas. Isso poderia ser verificado alterando a quantidade de neurônios na camada de entrada, diminuindo o tamanho do vocabulário, ou até mesmo normalizando os valores dos neurônios para corresponderem a frequência das palavras.

A opção do processamento com stop words será a adotada para a continuação do notebook.

3 DataLoader

3.1 Exercício III.1

Objetivos desse exercício

- Explicar as duas principais vantagens do uso de batch no treinamento de redes neurais.
- Explicar por que é importante fazer o embaralhamento das amostras do batch em cada nova época.
- Explicar por que quando alterado o `shuffle=False` no instanciamento do objeto `test_loader`, o cálculo da acurácia não se altera.

Conforme resposta do chatGPT, as vantagens de utilizar batches no treinamento do modelo são as seguintes

- Eficiência de memória: Processar grandes conjuntos de dados pode exigir muita memória. Usar batches permite que o modelo processe apenas um subconjunto dos dados por vez, reduzindo significativamente os requisitos de memória.
- Tempo de execução: Processar várias amostras simultaneamente aproveita a paralelização oferecida por hardware moderno, como GPUs. Isso resulta em treinamentos mais rápidos, pois as operações matriciais são eficientes e podem ser executadas em paralelo.

A importância de embaralhar os dados para que os batches não tenham várias amostras com o mesmo label será discutida a seguir.

A acurácia com o embaralhamento do conjunto de treino já foi apresentada.

Em teste feito sem o embaralhamento do conjunto de treino, o modelo apresentou uma acurácia de 50%, razão pela qual se mostra indispensável o prévio embaralhamento do conjunto de treino.

Levando em consideração que o treinamento foi feito apenas com batches onde as amostras têm o mesmo label, é esperado que o modelo não aprenda, já que o gradiente encontra um mínimo local durante o aprendizado muito rapidamente.

Em relação ao cálculo da acurácia com o conjunto de testes, o modelo não está aprendendo nem dando passos com o gradiente descendente, mas na verdade apenas verificando se a resposta predita para um determinado input é a mesma que a resposta verdadeira. Sendo assim, a ordem que processamos as amostras do conjunto de teste não importa.

3.2 Exercício III.2

Objetivo do exercício

- Medir quantas iterações possui o objeto `train_loader` e explicar o valor encontrado.
- Imprimir o número de amostras do último batch do `train_loader` e justificar o valor encontrado.
- Utilizando técnicas de *list comprehension*, calcular R , a relação do número de amostras positivas sobre o número de amostras no batch e no final encontrar o valor médio de R , para ver se o data loader está entregando batches balanceados.
- Mostrar a estrutura de um dos batches.

Para verificar quantos batches de 128 amostras possui o DataLoader e quantas amostras vão sobrar no último batch, o seguinte código foi implementado

```
batch_size = 128
# define dataloaders
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_data, batch_size=batch_size, shuffle=False)

batches_cnt = 0
last_batch = None
for batch in train_loader:
    batches_cnt += 1
    last_batch = batch
```

A variável `batches_cnt` tem valor 196, significando que existem 196 batches para treinamento e `len(last_batch[1])` tem valor 40 indicando que existem 40 amostras no último batch. Nesse caso o último batch fica com o restante dos elementos que não couberam nos batches anteriores, ou seja, o tamanho do último batch vai ser $25000 \bmod 128 = 40$.

Para calcular o valor médio de R dos batches, a seguinte linha de código foi utilizada,

```
np.average([np.count_nonzero(label == 1) for _, label in train_loader]) / batch_size
```

retornando o valor de 0.4928, significando que o dataset é balanceado como um todo, porém não garante o balanceamento de cada batch. Para garantir o balanceamento individual de cada batche, deve-se verificar o desvio padrão dos valores de R

Iniciando a análise do formato dos batches, pode-se utilizar o seguinte código

```
batch = next(iter(train_loader))
print(batch[0].shape, batch[1].shape)
```

Os valores obtidos são `torch.Size([128, 20001])` e `torch.Size([128])` respectivamente. Como existem 128 entradas, cada uma composta por um vetor

com 20001 valores (que são os neurônios da camada de entrada) e uma label (um único inteiro), os valores obtidos estão condizentes.

Para analisar uma única entrada desse batch, foi utilizada a seguinte linha de código,

```
print(batch[0][0], batch[1][0])
```

que nos retorna `tensor([1., 1., 1., ..., 0., 0., 0.]) tensor(0)`, indicando um vetor de 20001 floats e um inteiro que seria o label daquele vetor.

Nesse vetor em específico podemos ver que essa review tem pelo menos uma palavra sem token, além das palavras “the” (token 1) e “and” (token 2) em algum momento da review (posições 0, 1 e 2 do vetor estão ligadas).

3.3 Exercício III.3

O objetivo desse exercício é verificar a influência do batch size na acurácia final do modelo.

Testes serão feitos com o batch size em 1 e em 256 para análise da acurácia. O learning rate será mantido em 0.1 pois é um valor bem estável, como visto anteriormente.

Quando o batch size é igual a 1 a cópia dos dados de treinamento para a memória da GPU leva 26.9s, comparado a 21.6s com batches de 128 e 22s com batches com 256.

Quanto ao treinamento, com o batch size de 1 em relação ao tempo de execução de cada época, fomos para uma média de 30s, comparado com 0.25s com batches de 128 e 4s com batches de 256.

A acurácia também foi alterada pelo tamanho dos batches. Obtivemos 81.01% com batch size de 1, 85.4% com batches de 128 e 85.28% com batches de 256.

Pode-se concluir que batches grandes demais ultrapassam o ponto de vantagem do paralelismo das GPUs, criando um overheading de muitas threads para poucos núcleos físicos. Quanto a batches muito pequenos, conclui-se que são ineficientes, pois não é utilizada a capacidade de computação paralela das GPUs.

4 Modelo MLP

Para responder aos exercícios dessa seção deve-se entender os conceitos de redes neurais fully connected.

4.1 Exercício IV.1

Como objetivo desse exercício devemos experimentar e entender o funcionamento da rede neural, além de calcular a sua quantidade de parâmetros.

Observe o funcionamento do modelo

```

model = OneHotMLP(vocab_size)
input, label = batch
logit = model(input)
logit = torch.sigmoid(logit)
logit

```

O modelo é instanciado e iniciado com valores aleatórios nos seus neurônios. Sabendo disso, o logit (valor de saída da rede neural) terá um valor não significativo para prever o valor da label.

A função sigmoide apenas normaliza os valores para algo entre 0 e 1, para que possam ser interpretados como uma probabilidade.

Como saída obtém-se um tensor de 128 (tamanho do nosso batch) valores entre 0 e 1, que significam o que a rede neural previu para cada uma das 128 reviews que entraram como input.

Para o cálculo da acurácia do modelo, deve-se considerar saídas de até 0.5 como previsões 0 e o restante como previsões 1. Dessa forma, a acurácia é algo próximo aos 50%, pois o modelo não aprendeu nada e está apenas prevendo valores aleatórios, conforme a figura a seguir.

```

corrects = 0
predicted = torch.round(logit)
for i in range(batch_size):
    corrects += int(predicted[i] == int(label[i]))
print(f"Acurácia: {100 * corrects / batch_size}")
print(f"Input shape {input.shape}, Label shape {label.shape}")

```

Acurácia: 47.65625
Input shape torch.Size([128, 20001])
Label shape torch.Size([128])

Agora será feito o treinamento do modelo com LR de 0.1, para que sua acurácia seja verificada no mesmo batch. Nesse caso, o treinamento está “vazando” no teste e o valor da acurácia encontrado não é representativo para um conjunto de dados desconhecido. Observe a figura abaixo onde o cálculo da acurácia no modelo treinado é realizado.

```

model = train(0.1)

# Após o treinamento devolvemos o modelo para a CPU
model = model.to('cpu')
input, label = batch
logit = model(input)
logit = torch.sigmoid(logit)

corrects = 0
predicted = torch.round(logit)
for i in range(batch_size):
    corrects += int(predicted[i] == int(label[i]))
print(f"Acurácia: {100 * corrects / batch_size}")
print(f"Input shape {input.shape}\nLabel shape {label.shape}")

```

```
Epoch [1/5],      Loss: 0.5702
Epoch [2/5],      Loss: 0.4324
Epoch [3/5],      Loss: 0.3695
Epoch [4/5],      Loss: 0.3301
Epoch [5/5],      Loss: 0.2998
Acurácia: 86.71875
Input shape torch.Size([128, 20001])
Label shape torch.Size([128])
```

Observe que após 5 épocas de treinamento sobre o conjunto de 12500 amostras de treinamento, o modelo consegue aprender sobre as reviews.

Com a ajuda das funções `.weight` e `.bias` das FCs, é possível encontrar o número de parâmetros (pesos ajustados no treinamento) da rede neural.

	FC1		FC2		Total
	Weight	Bias	Weight	Bias	
Shape	200, 20001	200	1, 200	1	
Parameters	4000400		201		4000601

5 Treinamento

Um bom entendimento sobre a função sigmoide e a função de entropia cruzada além de noções de inicialização de nossa rede neural é necessário para os exercícios dessa seção.

5.1 Exercício V.1

O objetivo desse exercício é estimar o valor teórico da função de perda para uma rede neural com pesos aleatórios além de verificar diferentes implementações da função de perda.

Quando o modelo é inicializado, os seus pesos virão com valores aleatórios, ou seja, a probabilidade de predição de uma determinada classe é aleatória e nem sempre 50%. O que fica próximo a 50% é a acurácia, pois o modelo ainda não aprendeu e está apenas chutando valores independente da entrada que recebe.

Os valores chutados pela nossa rede neural se comportam como uma distribuição uniforme, já que ela não favorece nenhum valor específico, que quando passados na função sigmoide se acumulam nos extremos. Logo, teremos uma grande quantidade de valores muito próximos a 0 ou 1.

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

Portanto, o valor esperado tanto para y_i quanto para \hat{y}_i será 0.5, pois nosso modelo não favorece nenhuma das duas classes. Assumindo esses valores, o somatório desaparece e ficamos apenas com

$$L(y, \hat{y}) = -2 \cdot 0.5 \cdot \log(0.5) = \log(2)$$

Com o código abaixo, é possível verificar que o valor experimental é realmente próximo a $\log(2)$, que é o valor teórico encontrado

```
def exp_loss(target, prob):
    assert len(target) == len(prob), "Target and Prob have different sizes."
    target = np.array(target).squeeze()
    prob = np.array(prob.detach().numpy()).squeeze()
    N = len(target)
    return - np.sum(target * np.log(prob) + (1 - target) * np.log(1 - prob)) / N
```

Fortalecendo as confirmações anteriores, funções já existentes do pytorch também apresentam um resultado similar

```
input, label = first_batch
model = OneHotMLP(vocab_size)
outputs = torch.sigmoid(model(input))

bce = nn.BCELoss()
bce_logits = nn.BCEWithLogitsLoss()

print(exp_loss(label, outputs))
print(float(bce(outputs.squeeze(), label.float()))))
print(float(bce_logits(outputs.squeeze(), label.float())))
```

```
0.6888516480103135
0.6888515949249268
0.6854892373085022
```

Como observado, a função de entropia cruzada binária é exatamente a mesma que a `BCELoss()` do pytorch, e a pequena diferença de valor se dá pela imprecisão do float.

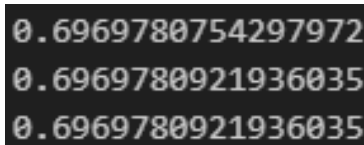
Quando comparadas com a função `BCEWithLogitsLoss()` é notada uma maior diferença e isso se dá pela dupla aplicação da função sigmoide, já que a função será aplicada novamente pela chamada do criterion `bce_logits()`, causando instabilidade numérica.

Com pouca adaptação ao código, obtemos resultados mais consistentes.

```
input, label = first_batch
model = OneHotMLP(vocab_size)
outputs = model(input)
probs = torch.sigmoid(outputs)

bce = nn.BCELoss()
bce_logits = nn.BCEWithLogitsLoss()

print(exp_loss(label, probs))
print(float(bce(probs.squeeze(), label.float()))))
print(float(bce_logits(outputs.squeeze(), label.float())))
```

5.2 Exercício V.2

No enunciado desses exercício algumas sugestões foram feitas para debug do modelo.

As recomendações do exercício foram incorporadas ao código.

O objetivo desse exercício é entender o que é necessário para que o treinamento comece novamente do modelo aleatório.

Para o modelo ser reinicializado e perder todo o progresso aprendido, devemos aleatorizar os valores de seus pesos, manualmente ou apenas o instanciando novamente.

5.3 Exercício V.3

O objetivo desse exercício é o mesmo do Exercício V.1, porém agora vamos lidar com o modelo com 2 logitos e uma softmax na saída da rede neural.

De maneira similar à entropia cruzada binária, a função de perda será um valor muito próximo a $\log(2)$, pois desenvolvendo o somatório de classes da fórmula geral, chegamos na mesma fórmula da entropia cruzada binária (quando $C = 2$). Algo análogo também acontece na função softmax, que se torna a função sigmoide e seu complemento.

$$H(y, \hat{y}) = -\frac{1}{N} \sum_{i=1}^N \sum_{c=1}^C y_{i,c} \log \hat{y}_{i,c}$$

Assumindo um dataset balanceado, a probabilidade de y_i ser da classe C será de $\frac{1}{C}$, portanto $y_{i,c} = \frac{1}{C}$ e o mesmo argumento vale para o predito, pois se o modelo não favorece nenhuma classe, a probabilidade de previsão para uma determinada classe tem que ser a mesma para outras classes. Portanto $\hat{y}_{i,c} = \frac{1}{C}$.

Sendo assim

$$H(y, \hat{y}) = -\frac{1}{N} \sum_{i=1}^N \sum_{c=1}^C \frac{1}{C} \log \frac{1}{C}$$

$$H(y, \hat{y}) = -\frac{1}{N} \sum_{i=1}^N \log \frac{1}{C}$$

$$H(y, \hat{y}) = -\log \frac{1}{C} = \log C$$

Iniciando a adaptação do código para ter 2 logitos na saída da rede neural, devemos incorporar o uso da função softmax. Assim, a soma dos valores de saída será 1 e ambos os valores serão probabilidades entre 0 e 1.

Para tanto, primeiro deve-se modificar a FC2 do modelo

```
self.fc2 = nn.Linear(200, 2)
```

Logo após, deve-se ajustar a função sigmoide e a função de perda para que seja averiguado o seu valor no primeiro batch do conjunto de treino.

```
input, label = first_batch
model = OneHotMLP(vocab_size)
outputs = model(input)
probs = torch.log_softmax(outputs, dim=1)
probs_ = torch.softmax(outputs, dim=1)

logits_label = torch.tensor(
    [(int(not label), int(label)) for label in label]
)

CE = nn.NLLLoss()
SMCE = nn.CrossEntropyLoss()

print(NL_loss(logits_label, probs_))
print(float(CE(probs, label)))
print(float(SMCE(outputs.squeeze(), label)))
```

```
0.6935951230116189
0.6935951113700867
0.6935951113700867
```

Note que o output da rede está na variável `outputs`, e as probabilidades das duas funções softmax em `probs` e `_probs`.

Isso é necessário pois a implementação explícita da função de entropia cruzada recebe os valores da softmax em si, enquanto a `NLLLoss` recebe os valores da `log_softmax`, para não ter que aplicar o logaritmo durante seus cálculos.

Por último existe a função `CrossEntropyLoss`, que combina as funções `log_softmax` e a `NLLoss`. Para referência, aqui está a implementação explícita da função de entropia cruzada para o caso geral

```
def NL_loss(target, prob):
    assert len(target) == len(prob), "Target and Prob
    target = np.array(target).squeeze()
    prob = np.array(prob.detach().numpy()).squeeze()
    N = len(target)
    C = len(target[0])
    sum = 0
    for i in range(N):
        for c in range(C):
            sum += target[i][c] * np.log(prob[i][c])
    return - sum / N
```

A partir daqui, foram feitas as modificações necessárias no código do treinamento e da avaliação para o modelo com 2 logitos.

6 Avaliação

6.1 Exercício VI.1

Objetivos desse exercício

- Calcular o número de amostras que está sendo considerado na seção de avaliação.
- Explicar o que fazem os comandos `model.eval()` e `with torch.no_grad()`
- Encontrar alternativas para a função sigmoide.

Para encontrar a quantidade de amostras de teste basta utilizar o comando `len(test_loader.dataset)`, que nos retorna 25000.

O comando `model.eval()` coloca o modelo em modo de teste, para ser avaliado. Esse comando desliga algumas partes específicas do modelo que se comportam de maneira diferente durante o treinamento e a avaliação, como por exemplo Dropouts Layers e BatchNorm Layers. É necessário desligar essas partes durante a avaliação do modelo e `.eval()` faz justamente isso.

Quanto ao comando `with torch.no_grad()`, especifica que naquele escopo não serão feitas computações de gradiente.

Em relação a uma melhor maneira de computar sem a função sigmoide (no caso com 1 logito apenas), pode ser observado o ponto médio dos valores preditos e separá-los em 2 conjuntos.

6.2 Exercício VI.2

O objetivo desse exercício é entender sobre a perplexidade como métrica de avaliação do modelo e discutir suas vantagens em relação a acurácia, além de prever seus valores esperados.

Como já concluído anteriormente, o valor esperado para entropia de um modelo com pesos aleatórios em um dataset balanceado com C classes é dado por

$$H(y, \hat{y}) = \log C$$

portanto o valor estimado para a perplexidade será

$$PPL = e^{\log C} = C$$

A perplexidade também possui um valor muito significativo quando a Loss é 0, ou seja, quando nosso modelo acerta 100% das amostras

$$PPL = e^{H(y, \hat{y})} = 1 \iff H(y, \hat{y}) = 0$$

6.3 Exercício VI.3

Objetivo do exercício: Modificar o código da seção VI - Avaliação, para que além de calcular a acurácia, calcule também a perplexidade.

O cálculo da perplexidade foi adicionado ao código, conforme segue

```
def eval(model):
    model.eval()
    criterion = nn.CrossEntropyLoss()
    total_loss = 0.0

    with torch.no_grad():
        correct = 0
        total = 0
        for inputs, labels in on_device_test_loader:
            outputs = model(inputs)
            predicted = convert_output(outputs)

            loss = criterion(outputs, labels)
            total_loss += loss.item()

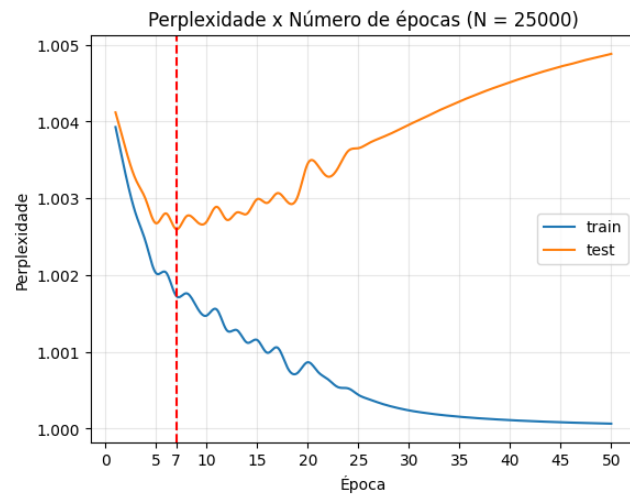
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    avg_loss = total_loss / len(test_loader.dataset)
    print(f'Test Loss: {avg_loss:.4f}')
    print(f'Test PPL: {np.exp(avg_loss):.4f}')
    print(f'Test Accuracy: {100 * correct / total}%')
    return 100 * correct / total
```

6.4 Exercício VI.4

O objetivo desses exercício é observar o overfitting do modelo quando o número de épocas aumenta.

Veja o caso de overfitting nos dados de treinamento conforme aumentamos o número de épocas de treinamento do modelo.



Como observado, a perplexidade do conjunto de treino continua diminuindo após a época 7, o contrário da perplexidade do conjunto de teste, que tem seu valor cada vez maior (pior) após essa época, indicando que o modelo está overfitando.

