# Processo Seletivo IA024 1S2024 – Relatório

Leandro Carísio Fernandes

## I Vocabulário e tokenização

### I.1 Total de amostras e comprimento médio dos textos

| |
|---|
| 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.<br><br>Resposta esperada:<br><br>- a modificação do trecho de código<br>- número de amostras positivas, amostras negativas e amostras totais<br>- comprimento médio dos textos das amostras (em número de palavras) |

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

O dataset torchtext.datasets.IMDB usa os valores 1 e 2 para a classificação negativa e positiva. Para evitar erros ao usar esses valores, criei variáveis para elas:

Código

```
neg = 1
pos = 2
```

Para calcular o total de amostras positivas/negativas, criei um Counter (counter_sentiment) e, no loop, atualizei essa variável com o label.

A partir dos dois objetos Counter é possível calcular a média de palavras por amostra: a variável counter possui o total de palavras do dataset e a nova variável counter_sentiment possui o total de amostras:

Código

```
counter_sentiment = Counter()
  for (label, line) in list(IMDB(split='train')):
      ...
      counter_sentiment.update([label])
avg_words_per_sample = sum(counter.values())/sum(counter_sentiment.values())
```

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

O número de amostras pode ser extraído diretamente do novo contador implementado:

```
Código
print('Número de amostras positivas: ', counter_sentiment [pos])
print('Número de amostras negativas: ', counter_sentiment[neg])
print('Total de amostras: ', sum(counter_sentiment.values()))
Saída
Número de amostras positivas:  12500
Número de amostras negativas:  12500
Total de amostras:  25000
```

I.1.c) Comprimento médio dos textos das amostras:

O comprimento médio do texto pode ser extraído com o resultado dos dois contadores usados:

```
Código
print('Média de palavra por amostra: ', avg_words_per_sample)
Saída
Média de palavra por amostra:  233.7872
```

## I.2 Frequência de palavras e representação de cada palavra

Enunciado do exercício: 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.

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

A frequência das palavras está ordenada na lista representada pela variável most_frequent_words. Assim, basta extrair as 5 primeiras e as 5 últimas da lista:

```
Código
print('Cinco palavras mais frequentes:', most_frequent_words[0:5])
print('Cinco palavras menos frequentes:', most_frequent_words[-5:])
Saída
Cinco palavras mais frequentes: ['the', 'a', 'and', 'of', 'to']
Cinco palavras menos frequentes: ['age-old', 'place!', 'Bros', 'tossing',
'nation,']
```

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

O código para unknown token (UNK) é 0.

A primeira linha do código abaixo ordena os dados do contador de palavras do maior para o menor e, em seguida, seleciona as 20.000 (vocab_size) primeiras palavras.

A segunda linha cria um mapa onde a chave é uma palavra e o valor é um inteiro. Da forma como foi gerado, o inteiro representa a posição da palavra na lista

most_frequent_words (começando a indexação em 1). Ou seja, a palavra mais frequente (the) é a primeira posição no vetor e, por isso, sua representação numérica é 1.

Dessa forma, a representação das palavras no vocabulário varia do índice 1 ao índice 20.000. O índice 0 é usado para representar UNK.

Código

```
most_frequent_words = sorted(counter, key=counter.get,
                                                reverse=True)[:vocab_size]
vocab = {word: i for i, word in enumerate(most_frequent_words, 1)}
```

A função encode_sentence recebe uma frase, separa cada palavra da frase e, para cada palavra, pesquisa na variável vocab pela sua representação numérica. A pesquisa é feita pela função get e, caso a palavra não esteja no vocabulário, a função get retorna 0, que é o valor default para *unknown tokens*:

Código

```
def encode_sentence(sentence, vocab):
    return [vocab.get(word, 0) for word in sentence.split()]
```

Para testar essa lógica, podemos chamar encode_sentence com uma palavra que certamente não existe no vocabulário:

Código

```
print(encode_sentence("alskfjlaksjflasfjlaskfjlas", vocab))
```

Saída

```
[0]
```

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

Há várias formas de calcular quantas palavras do conjunto de treinamento foram classificadas como unknown tokens. A mais simples é verificando quantas chaves o contador possui e subtrair de 20000 (tamanho do vocabulário).

Outra forma é extrair os UNK da mesma forma que foi calculada a variável most_frequenct_words, mas em vez de extrair os 20000 primeiros do counter, extrair a partir do elemento 20001.

Com essa segunda forma é possível calcular, em seguida, o total de palavras classificadas como UNK considerando repetições.

O código abaixo mostra que 260.617 palavras foram classificadas como UNK e elas aparecem 566.141 no conjunto de treinamento:

Código

```
print('Total de palavras distintas classificadas como UNK', len(counter.keys())
                                                              - 20000)

unk_tokens = sorted(counter, key=counter.get, reverse=True)[vocab_size:]
print('Total de palavras distintas classificadas como UNK', len(unk_tokens))

total_unk = sum([counter[unk] for unk in unk_tokens])
print('Total de palavras classificadas como UNK', total_unk)
```

Saída

```
Total de palavras distintas classificadas como UNK 260617
Total de palavras distintas classificadas como UNK 260617
Total de palavras classificadas como UNK 566141
```

## I.3 Reduzindo o número de amostras para 200

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?

O comando da questão solicitou para trocar a lista tanto da linha 5 da célula de calcular o vocabulário e da célula do "II – Dataset" por:

list(IMDB(split='train'))[:200]

Ao fazer isso, estamos usando, tanto para o dataset de treino quanto para o dataset de teste, as mesmas 200 primeiras amostras de treinamento.

Há algumas questões aqui:

- Ao usar apenas 200 amostras, é possível que o modelo esteja decorando os resultados (*overfitting*). Considerando que a base de teste é a mesma que a base de treinamento, o modelo está apenas retornando os resultados decorados.
- Nesse caso, mesmo que os datasets fossem diferentes e tivéssemos usados as 200 primeiras amostras de treinamento e as 200 primeiras amostras de teste, o resultado seria parecido. Isso ocorre pois as 200 primeiras amostras de ambos os dataset possuem o label = 1. Assim, na etapa de treinamento bastaria "aprender" que o resultado seria sempre 1 e sempre retornar esse valor. Como a base de testes também sempre tem o label = 1, o resultado seria 100%.

Observação:

Ao fazer o teste eu tive o seguinte erro: `RuntimeError: mat1 and mat2 shapes cannot be multiplied (128x10119 and 20001x200).`

Isso ocorreu porque, ao construir o vocabulário usando apenas as primeiras 200 amostras, o tamanho do vocabulário foi de 10.119. Para conseguir simular com essa redução de amostras do conjunto de treinamento, reduzi o tamanho do vocabulário para 10.000.

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.

Uma forma de deixar perfeitamente balanceado é selecionar as 100 primeiras (negativas) e as 100 últimas amostras (positivas):

```
Código
idx_amostras = list(range(100)) + list(range(24900, 25000))
...
for (label, line) in [imdb_train[i] for i in idx_amostras]:
  ...
```

É possível fazer desse jeito pela forma como o dataset foi construído. Em casos mais genéricos poderíamos simplesmente amostrar aleatoriamente 200 índices do dataset original.

# II Dataset

## II.1 Funcionamento da classe IMDBDataset

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.

A classe IMDBDataset representa o conjunto de dados do IMDB. Ele retorna uma codificação para a frase e um label associado.

Cada frase é codificada como um tensor de tamanho igual a (vocab_size + 1). Os elementos do tensor são valores 1 ou 0. O valor 1 representa a presença (independentemente de quantas vezes a palavra está na frase) da palavra na frase. O valor 0 indica a ausência da palavra na frase.

Em relação ao label, é importante ressaltar que houve uma tradução dos valores em relação à base original. A base original trata os labels negativo = 1 e positivo = 2. O código da classe IMDBDataset usa labels 0 e 1. Para isso, ele mantém o label negativo = 1 e transforma o label positivo em 0.

O código para o cálculo do número de amostras positivas/negativas é:

```
Código
# Se antes o sentimento era calculado como 1 (negativo) e 2 (positivo), agora
ele passou a calcular como 1 (negativo) e 0 (positivo)
# Assim, vamos ajustar as variavéis pos e neg:
neg = 1
pos = 0

# Faça um código que aplique um laço sobre o dataset train_data e calcule
novamente quantas amostras positivas e negativas do dataset.
counter_sentiment = Counter()
sum_one_hot_tensor = torch.zeros(vocab_size + 1)
for one_hot_tensor, label in train_data:
  counter_sentiment.update([label.item()])
  sum_one_hot_tensor += one_hot_tensor

avg_words_per_sample = sum(sum_one_hot_tensor)/sum(counter_sentiment.values())

print('Número de amostras positivas: ', counter_sentiment[pos])
print('Número de amostras negativas: ', counter_sentiment[neg])
```

```
print('Total de amostras: ', sum(counter_sentiment.values()))
```

Saída

```
Número de amostras positivas:  12500
Número de amostras negativas:  12500
Total de amostras:  25000
```

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.

O código é o da listagem acima. A impressão da variável avg_words_per_sample retorna apenas 133.0955 “palavras” por frase, bem menor do que 233.7872 palavras por frase calculadas anteriormente.

Essa diferença ocorre porque a codificação usada na classe IMDBDataset para a frase apenas considera se a palavra foi usada ou não. Assim, mesmo que uma palavra seja usada 10 vezes na frase, ela contará como apenas uma vez nessa codificação. Isso é importante para reduzir o peso de palavras muito frequentes (the, a, an, etc) no texto.

Código

```
print('Média de palavra por amostra: ', avg_words_per_sample)
```

Saída

```
Média de palavra por amostra:  tensor(133.0955)
```

## II.2 Aumentando a eficiência do treinamento com o uso da GPU T4

Com a 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 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.

II.2.a) Tempo do laço = ; Tempo do forward = ;Tempo do backward = ; Conclusão

Alterei o código para deixar apenas os loops e medir o tempo para percorrer o DataLoader. Em 5 épocas o tempo médio foi de ~27 s:

Código

```
for epoch in range(num_epochs):
    start_time = time.time()  # Start time of the epoch
    for inputs, labels in train_loader:
        teste = 1

    end_time = time.time()  # End time of the epoch
    epoch_duration = end_time - start_time  # Duration of epoch

    print(f'Epoch [{epoch+1}/{num_epochs}], \
            Elapsed Time: {epoch_duration:.2f} sec')
```

Saída

```
Epoch [1/5],              Elapsed Time: 29.65 sec
Epoch [2/5],              Elapsed Time: 26.44 sec
Epoch [3/5],              Elapsed Time: 26.38 sec
Epoch [4/5],              Elapsed Time: 27.46 sec
Epoch [5/5],              Elapsed Time: 26.26 sec
```

Usando o código passado como parâmetro, medi o tempo do loop forward. Para isso, coloquei uma instrução continue depois do forward pass. O tempo médio foi de ~29 s.

Código

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

    end_time = time.time()  # End time of the epoch
    epoch_duration = end_time - start_time  # Duration of epoch

    print(f'Epoch [{epoch+1}/{num_epochs}], \
            Loss: {loss.item():.4f}, \
            Elapsed Time: {epoch_duration:.2f} sec')
```

Saída

```
Epoch [1/5],              Loss: 0.6909,             Elapsed Time: 29.08 sec
Epoch [2/5],              Loss: 0.6948,             Elapsed Time: 30.05 sec
Epoch [3/5],              Loss: 0.6936,             Elapsed Time: 29.10 sec
Epoch [4/5],              Loss: 0.6924,             Elapsed Time: 29.95 sec
Epoch [5/5],              Loss: 0.6909,             Elapsed Time: 29.22 sec
```

Já o tempo médio para executar todo o loop com o código original foi de ~33s:

Código

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

    end_time = time.time()  # End time of the epoch
    epoch_duration = end_time - start_time  # Duration of epoch

    print(f'Epoch [{epoch+1}/{num_epochs}], \
            Loss: {loss.item():.4f}, \
            Elapsed Time: {epoch_duration:.2f} sec')
```

Saída

```
Epoch [1/5],              Loss: 0.6933,             Elapsed Time: 32.79 sec
Epoch [2/5],              Loss: 0.6920,             Elapsed Time: 38.42 sec
Epoch [3/5],              Loss: 0.6894,             Elapsed Time: 32.33 sec
Epoch [4/5],              Loss: 0.6891,             Elapsed Time: 33.05 sec
Epoch [5/5],              Loss: 0.6845,             Elapsed Time: 31.61 sec
```

Nessa última rodada (código completo), houve uma oscilação na segunda época no notebook. Se desconsiderarmos esse valor o tempo médio reduz para ~32s.

A conclusão é que o *tempo para percorrer o DataLoader é muito alto*, é o principal causador do problema.

II.2.b) Trecho que precisa ser otimizado

Como o tempo para extrair os dados do DataLoader é alto, esse deve ser o primeiro trecho de código que devemos procurar otimizar. A extração é feita no método __getitem__. Dentro deste método há um loop que pode ser otimizado:

Código

```
class IMDBDataset(Dataset):
    ...
    def __getitem__(self, idx):
        label, line = self.data[idx]
        label = 1 if label == 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(label)
```

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

Temos pelo menos duas possibilidades de otimização aqui:

Primeira possibilidade: Toda vez que o __getitem__ é executado é gerado um tensor. Assim, todo esse cálculo é refeito em diferentes épocas. Uma alternativa para resolver isso é criar um cache com todos os tensores em memória. *Essa alternativa é viável para essa base de dados, que é relativamente pequena, mas não é uma solução escalável e, por isso, será desconsiderada.*

A segunda alternativa é a remoção do loop (for word in encode_sentence...). O que esse loop faz é apenas atribuir 1 para qualquer elemento retornado pela função encode_sentence. Em outras palavras, o retorno de encode_sentece é um array de índices, e queremos atribuir a constante 1 a esses índices em X. O Python permite fazer essa operação diretamente:

Código

```
class IMDBDataset(Dataset):
    ...
    def __getitem__(self, idx):
        label, line = self.data[idx]
        label = 1 if label == 1 else 0

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

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

Para garantir que a refatoração está correta, podemos comparar as duas codificações com uma frase genérica qualquer:

Código

```
# Trecho de código para assegurar que o refactoring está correto
sentence = "Some sentence to check if the refactoring is correct. Let's just
use some random words now: the is a blue color"
X = torch.zeros(len(vocab) + 1)
for word in encode_sentence(line, vocab):
    X[word] = 1
Y = torch.zeros(len(vocab) + 1)
Y[encode_sentence(line, vocab)] = 1
sum(abs(X-Y))
```

Saída

```
tensor(0.)
```

Após essa modificação, o tempo para execução do código reduziu para ~4s:

Código

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

    end_time = time.time()  # End time of the epoch
    epoch_duration = end_time - start_time  # Duration of epoch

    print(f'Epoch [{epoch+1}/{num_epochs}], \
            Loss: {loss.item():.4f}, \
            Elapsed Time: {epoch_duration:.2f} sec')
```

Saída

```
Epoch [1/5],            Loss: 0.6861,            Elapsed Time: 4.30 sec
Epoch [2/5],            Loss: 0.6826,            Elapsed Time: 5.66 sec
Epoch [3/5],            Loss: 0.6847,            Elapsed Time: 4.18 sec
Epoch [4/5],            Loss: 0.6767,            Elapsed Time: 4.76 sec
Epoch [5/5],            Loss: 0.6858,            Elapsed Time: 4.64 sec
```

## II.3 Escolhendo um bom valor de LR

| |
|---|
| 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. |

Para resolver essa questão, alterei o loop de treino e de teste para executarem dentro de funções. Em seguida, para um conjunto de LR possíveis, iniciei o modelo, treinei por 15 épocas e calculei a loss e a acurácia.

II.3.a) Gráfico acurácia vs LR:

Código usado para testar 4 LR potência de 10, para checar a ordem de grandeza:

```
Código
lrs_para_avaliar = [0.1, 0.01, 0.001, 0.0001]

resultado_por_lr = {}
num_epoch = 15

for lr in lrs_para_avaliar:
    # Cria um modelo
    model = OneHotMLP(vocab_size)
    print(f'Treinando modelo com lr={lr}')
    loss_por_epoca = train_model(model, lr, num_epoch)
    acc = eval_model(model)
    resultado_por_lr[lr] = {'loss_por_epoca': loss_por_epoca, 'acc': acc}
```

Gráfico da LR versus Loss:

```
Código
import matplotlib.pyplot as plt

plt.figure()

# Plota loss
for lr in lrs_para_avaliar:
    plt.plot(range(1, num_epoch+1),  resultado_por_lr[lr]['loss_por_epoca'],
label=f'lr={lr}')
plt.ylabel('Loss')
plt.xlabel('Época')
plt.legend()
```

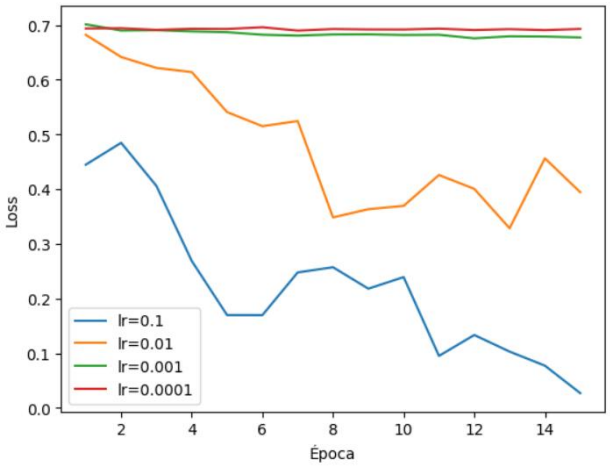
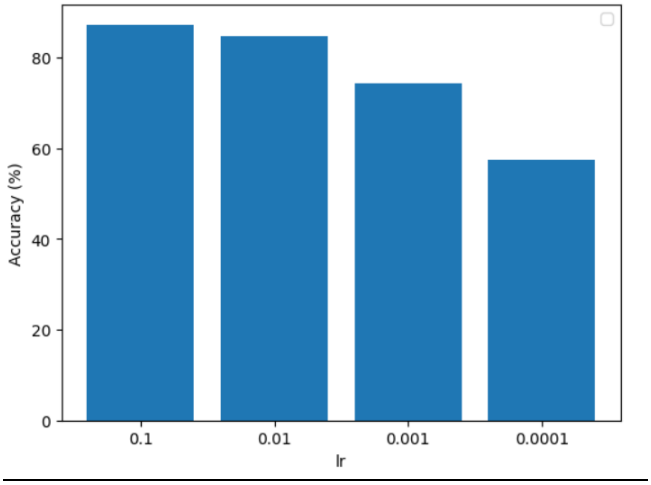



Gráfico da LR versus Accuracy:

```
Código
import matplotlib.pyplot as plt

plt.figure()

plt.bar([str(lr) for lr in lrs_para_avaliar], [resultado_por_lr[lr]['acc'] for
lr in lrs_para_avaliar])
plt.xlabel('lr')
plt.ylabel('Accuracy (%)')
plt.legend()
```




II.3.b) Valor ótimo da LR:

A partir do gráfico da Loss versus LR, pode-se observar que LR de 0.001 e 0.0001 são muito baixas e fazem com que a convergência seja muito lenta. LR de 0.01 e 0.1 melhoraram bastante a convergência da Loss com alguma oscilação aceitável.

Para o conjunto de valores testados e 15 épocas, **a LR ótima foi de 0.1**, com acurácia de ~87%.

Após isso testei também com valores ao redor de 0.1 (0.5, 0.4, 0.3, 0.2, 0.1, 0.09, 0.08, 0.07, 0.06, 0.05). Não houve melhoras significativas.

II.3.c) Equação do gradiente descendente e o papel da LR:

A equação de ajustes dos pesos é essa:

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

O papel da learning rate ($\alpha$) no ajuste dos parâmetros do modelo da rede neural é controlar a magnitude das atualizações dos pesos a cada iteração do algoritmo de otimização. A taxa de aprendizado determina o tamanho do passo que os parâmetros do modelo atualizam durante o processo de otimização.

Uma taxa de aprendizado maior fará com que os parâmetros do modelo sejam atualizados em passos maiores, o que pode levar a convergência mais rápida, mas também pode resultar em oscilações indesejadas ou divergência do processo de otimização. Por outro lado, uma taxa de aprendizado menor resultará em atualizações menores dos parâmetros do modelo, o que pode levar a uma convergência muito mais lenta.

## II.4 Otimizando o tokenizador

| |
|---|
| 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 |

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

Com a ajuda do ChatGPT gerei a seguinte função para fazer uma tokenização simples:

Código

```
## Função gerada com ajuda do ChatGPT
import re

def tokenizar(frase):
    # Converter a frase para minúsculo
    frase = frase.lower()
    # Remover caracteres especiais e pontuação usando expressões regulares
    frase = re.sub(r'[^\w\s]', '', frase)
    # Dividir a frase em palavras
    palavras = frase.split()
    return palavras

# Exemplo de uso
frase = "Olá! Como você está? Eu estou bem, obrigado."
palavras = tokenizar(frase)
print(palavras)
```

Saída

```
['olá', 'como', 'você', 'está', 'eu', 'estou', 'bem', 'obrigado']
```

Na seção “I – Vocabulário”, o counter foi atualizado com o resultado de tokenizar:

Código

```
for (label, line) in [imdb_train[i] for i in idx_amostras]:
    # Com tokenizador:
    counter.update(tokenizar(line))
    # Código removido:
    # counter.update(line.split())
    ...
```

O código acima irá alterar a variável vocab para considerar o resultado dessa nova forma de tokenizar a frase.

Na seção II – Dataset, o dataset retorna os dados encodados pela função encode_sentence, que por sua vez consulta o código da palavra na variável vocab. Assim, é necessário que a função seja alterada para também tokenizar usando a função tokenizar:

Código

```
def encode_sentence(sentence, vocab):
    return [vocab.get(word, 0) for word in tokenizar(sentence)] # 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.

| | Tamanho médio da frase em palavras | Número de tokens unknown |
|---|---|---|
| Tokenizador comum (split) | 233.78 | Distintas: 260617<br>Total: 566141 |
| Novo tokenizador (converte para minúsculo e remove pontuação) | 232.81 | Distintas: 101045<br>Total: 214060 |

O tamanho médio da frase reduziu muito pouco. O novo tokenizador desconsidera pontuação. Entretanto, como em regra a pontuação é colocada ao lado de uma palavra, sem espaço entre eles, esse resultado já era esperado.

Por outro lado, o número de palavras classificadas como UNK reduziu substancialmente. Isso ocorre pois, com a nova tokenização, não importa mais se uma palavra é escrita em letras maiúsculas ou minúsculas. Na versão original as palavras “Not”, “not” e “NOT” geravam três novos tokens. Nessa versão, geram apenas um token.

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.

Segue resultados, considerando 5 épocas e LR de 0.1:

| | Tokenizar original (split) | Novo tokenizador |
|---|---|---|
| Acurácia | ~87% | ~87% |

Não houve alteração relevante dos resultados.

# III DataLoader

## III.1 Shuffle

| |
|---|
| 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: |

| Shuffle | Acurácia |
|---|---|
| True | ~87% |
| False | 50% |

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

De acordo com o ChatGPT há duas principais vantagens:

- *Eficiência computacional*: Ao treinar uma rede neural em lotes de dados em vez de em um único exemplo de cada vez, é possível aproveitar a eficiência de processamento das unidades de processamento gráfico (GPUs) e de cálculo vetorial para processar várias amostras simultaneamente. Isso resulta em um treinamento mais rápido da rede neural, pois os cálculos são paralelizados e otimizados para aproveitar a capacidade de processamento massivamente paralela das GPUs.
- *Estabilidade do treinamento*: Treinar uma rede neural em lotes de dados em vez de em exemplos individuais pode ajudar a estabilizar o processo de otimização e melhorar a convergência do modelo. Ao calcular o gradiente da função de perda em relação aos parâmetros do modelo em um lote de dados, os gradientes resultantes são uma estimativa mais precisa do gradiente médio sobre todo o conjunto de dados. Isso pode ajudar a suavizar a superfície da função de perda e evitar oscilações indesejadas ou desvios significativos durante o treinamento, resultando em uma convergência mais estável e consistente do modelo.

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

Embaralhar as amostras fazem com que os dados apresentados para o modelo em cada batch sejam mais representativos da realidade. Em um conjunto de treinamento os dados podem, por exemplo, possuir alguma ordenação (como no caso da base do IMDB, que é ordenada pelo label). Além disso (ChatGPT):

- *Redução do viés do modelo*: Se as amostras não forem embaralhadas, o modelo pode aprender a depender da ordem específica das amostras no conjunto de dados. Isso pode introduzir um viés no modelo, onde ele se torna sensível à ordem das amostras de entrada. Embaralhar as amostras em cada época ajuda a reduzir esse viés, garantindo que o modelo seja exposto a diferentes ordens de amostras durante o treinamento.
- *Generalização melhorada*: Embaralhar as amostras em cada época ajuda a promover uma melhor generalização do modelo. Se as amostras não forem embaralhadas, o modelo pode memorizar a relação entre as amostras adjacentes e, consequentemente, não ser capaz de generalizar bem para dados novos e não vistos. Embaralhar as amostras garante que o modelo seja

exposto a uma variedade de padrões de entrada, o que pode melhorar sua capacidade de generalização para dados novos.

- *Evita ciclos de treinamento*: Embaralhar as amostras em cada época ajuda a evitar ciclos de treinamento, onde o modelo pode ficar preso em padrões específicos de amostras que não mudam ao longo do treinamento. Isso é especialmente importante quando o conjunto de dados não é grande o suficiente para garantir que todas as amostras sejam usadas várias vezes durante o treinamento.

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

Na base de testes os pesos já estão definidos e não são alterados. Dessa forma, considerando que a métrica não depende da ordem com que os dados são apresentados, não faz diferença a ordem com que os dados são avaliados.

## III.2 Estruturas de um batch

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.

Como há 128 amostras em cada batch e o conjunto de treinamento tem 25.000 registros, são necessárias $\lceil 25000/128 \rceil = \lceil 195.31 \rceil = 196$ iterações.

Podemos verificar usando um enumerate em train_loader iniciando em 1:

Código

```
for iteracao, batch in enumerate(train_loader, 1):
    pass
print(iteracao)
```

Saída

```
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?

O batch possui a entrada (batch[0]) e o label (batch[1]). Como o label é unidimensional, podemos verificar o tamanho do batch calculando len(batch[1]):

Código

```
total_amostras_batch = 0
for iteracao, batch in enumerate(train_loader, 1):
    total_amostras_batch = len(batch[1])
    pass
print(total_amostras_batch)
```

Saída

```
40
```

Esse resultado é esperado. Há 195 batchs completos (195 x 128 = 24960) e 40 registros no último batch. Sempre que o total de dados do dataset não for múltiplo do batch_size, o último batch conterá um total de amostras diferente de 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 resultado foi de 0.4996 e, conforme esperado, próximo de 0.5:

Código

```
R = [sum(batch[1])/len(batch[1]) for batch in train_loader]
print(sum(R)/len(R))
```

Saída

```
tensor(0.4996)
```

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)).

Código

```
batch = next(iter(train_loader))
print(f"O batch possui {len(batch)} elementos")
print(f"O primeiro elemento são os dados de entrada. Possui shape
        {batch[0].shape}")
print(f"Ou seja, são {batch[0].shape[0]} linhas (tamanho do batch) e
       {batch[0].shape[1]} colunas (tamanho do vocabulário com UNK)")
print(f"O tipo de dado do input é {batch[0].dtype}")
print(f"O segundo elemento são os labels relacionados. Possui shape
       {batch[1].shape} (tamanho do batch)")
print(f"O tipo de dado do label é {batch[1].dtype}")
```

Saída

```
O batch possui 2 elementos
O primeiro elemento são os dados de entrada. Possui shape torch.Size([128,
20001])
Ou seja, são 128 linhas (tamanho do batch) e 20001 colunas (tamanho do
vocabulário com UNK)
O tipo de dado do input é torch.float32
O segundo elemento são os labels relacionados. Possui shape torch.Size([128])
(tamanho do batch)
O tipo de dado do label é torch.int64
```

## III.3 Tamanho de um batch

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.

Da forma como o notebook está codificado, há um problema com o shape do batch ao tentar usar batch_size = 1. Para simplificar, fiz os testes com batch_size começando em 2 e mantendo o número de épocas (5) e learning rate (0.1). Como há uma amostragem envolvida (shuffle = True), o valor da acurácia amostrada é a média de 2 execuções do algoritmo:

| Tamanho do batch | Acurácia |
|---|---|
| 2 | 83% |
| 128 | 87% |
| 512 | 85% |
| 1024 | 83% |

Como o cálculo dentro do batch é paralelizado, quanto menor o tamanho do batch, mais lenta é a execução do algoritmo.

Como a learning rate estava ajustada para um tamanho de batch igual a 128, aumentar o tamanho do batch sem reajustar a learning rate fez com que a acurácia começasse a cair. Com tamanhos de batchs maiores, há menos atualizações dos pesos e, por isso, uma convergência mais lenta. Isso provavelmente pode ser compensado ajustando a learning rate (aumentando-a).

# IV Modelo MLP

## IV.1 Experimentando o modelo

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.

O código abaixo extrai os logits do modelo e calcula a probabilidade usando torch.sigmoid(logit). Isso é feito para um batch. Em seguida, para calcular o target estimado pelo modelo para comparar, a probabilidade foi arredondada para o inteiro mais próximo, o que vai jogar para 0 ou 1. Com isso, foi possível calcular o acerto médio dentro do batch (nesse caso de 50.7%):

Código

```
# Testes para experimentar o modelo
# Pega um batch qualquer
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
input, target = next(iter(train_loader))
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = OneHotMLP(vocab_size)
model.to(device)

# Calcula os logitos
logit = model(input.to(device))
## Converte para probabilidade
prob = torch.sigmoid(logit)
print(f"A média do cálculo da probabilidade da saída é {sum(prob)/len(prob)}")

target_estimado = torch.round(prob).squeeze()
num_acertos = ((target_estimado - target.to(device)) == 0).sum().item()

print(f"Número de acertos: {num_acertos}")
print(f"Tamanho do batch: {len(target)}")
print(f"Acerto médio: {num_acertos/len(target)}")

# Após treinar o modelo
train_model(model, lr=0.1, num_epochs=5)
logit = model(input.to(device))
prob = torch.sigmoid(logit)
print(f"A média do cálculo da probabilidade da saída é {sum(prob)/len(prob)}")

target_estimado = torch.round(prob).squeeze()
num_acertos = ((target_estimado - target.to(device)) == 0).sum().item()

print(f"Número de acertos: {num_acertos}")
print(f"Tamanho do batch: {len(target)}")
print(f"Acerto médio: {num_acertos/len(target)}")
```

Saída

```
A média do cálculo da probabilidade da saída é tensor([0.5173],
device='cuda:0', grad_fn=<DivBackward0>)
Número de acertos: 65
Tamanho do batch: 128
Acerto médio: 0.5078125
...
```

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.

O código do item acima (a) também faz o treinamento e exibe os resultados em seguida. Como pode ser observado no quadro abaixo, após o treinamento o número de acertos subiu consideravelmente, atingindo 89% de acerto para o batch testado:

Saída

```
...
A média do cálculo da probabilidade da saída é tensor([0.5083],
device='cuda:0', grad_fn=<DivBackward0>)
Número de acertos: 114
Tamanho do batch: 128
Acerto médio: 0.890625
```

Essa rede possui o seguinte número de parâmetros:

```
Código
print(model.fc1.weight.shape,
model.fc1.weight.shape[0]*model.fc1.weight.shape[1])
print(model.fc1.bias.shape)
print(model.fc2.weight.shape,
model.fc2.weight.shape[0]*model.fc2.weight.shape[1])
print(model.fc2.bias.shape)
Saída
torch.Size([200, 20001]) 4000200
torch.Size([200])
torch.Size([1, 200]) 200
torch.Size([1])
```

| layer | fc1 | | fc2 | | TOTAL |
|---|---|---|---|---|---|
| | weight | bias | weight | bias | |
| size | 4.000.200 | 200 | 200 | 1 | 4.000.601 |

# V Treinamento

## V.1 Cálculo da loss

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 a loss é calculada como:

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

Se a probabilidade predita é 0.5, podemos substituir $\hat{y}$ por 05 na equação:

$$L = -\frac{1}{N}\sum_{i}^{N}[y_i \log(0.5) + (1 - y_i) log(0.5)]$$

$$L = -\frac{\log(0.5)}{N}\sum_{i}^{N}[y_i + (1 - y_i)]$$

Como $y_i$ pode ser apenas 0 ou 1, o argumento do somatório sempre será log(0.5). Assim, a equação se reduz para:

$$L = -\frac{1}{N} N log(0.5) = -\log(0.5) = \log(2)$$

Ou seja, o valor teórico 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 a equação fornecida anteriormente utilizando o pytorch. Verifique se este valor confere com o valor teórico do exercício anterior.

O valor confere com o calculado teoricamente.

Código

```
# Inicializa um dataloader
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
# Pega o primeiro batch
inputs, y = next(iter(train_loader))
# Inicializa um modelo
model = OneHotMLP(vocab_size)
model.to(device)

# Calcula a saída do model
logit = model(inputs.to(device))
prob = torch.sigmoid(logit)
y_hat = prob.squeeze()

# Cálculo da loss:
N = len(prob)
loss_por_amostra = [y[i]*torch.log(y_hat[i]) + (1-y[i])*torch.log(1-y_hat[i])
for i in range(N)]
loss = -1/N * sum(loss_por_amostra)
print(loss)
```

Saída

```
tensor(0.6972, device='cuda:0', grad_fn=<MulBackward0>)
```

V.1.c) 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 valor é exatamente o mesmo:

Código

```
# Cálculo da loss:
loss_fn = nn.BCELoss()
loss = loss_fn(y_hat, y.to(device).float())
print(loss)
```

Saída

```
tensor(0.6972, device='cuda:0', grad_fn=<BinaryCrossEntropyBackward0>)
```

c) 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.

O valor é exatamente o mesmo:

Código

```
# Cálculo da loss:
loss_fn = nn.BCEWithLogitsLoss()
loss = loss_fn(logit.squeeze(), y.to(device).float())
print(loss)
```

Saída

```
tensor(0.6972, device='cuda:0',
       grad_fn=<BinaryCrossEntropyWithLogitsBackward0>)
```

## V.2 Minimização da loss pelo gradiente descendente

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.

O ChatGPT sugeriu uma forma de calcular a loss antes de iniciar o treinamento. Para isso, ele apresentou um código que encapsulei numa função e chamei antes do treinamento:

Código para calcular a loss

```
def calcula_loss(model, criterion):
  with torch.no_grad():  # Garante que nenhum gradiente seja calculado
    model.eval()  # Coloca o modelo no modo de avaliação (não treinamento)
    total_loss = 0.0
    total_samples = 0
    for inputs, labels in train_loader:
        inputs = inputs.to(device)
        labels = labels.to(device)
        # Forward pass
        outputs = model(inputs)
        # Calcula a perda
        loss = criterion(outputs.squeeze().to(device),
                         labels.float()).to(device)
        # Acumula a perda e o número total de amostras
        total_loss += loss.item() * inputs.size(0)
        total_samples += inputs.size(0)

  return total_loss / total_samples
```

Chamada a função antes do início do treinamento:

```
print(f'Loss antes de iniciar o treinamento: {calcula_loss(model, criterion)}')
# Training loop
for epoch in range(num_epochs):
    start_time = time.time()  # Start time of the epoch
    model.train()
    for inputs, labels in train_loader:
    ...
```

Saída

```
Loss antes de iniciar o treinamento: 0.6935799642562867
Epoch [1/5],            Loss: 0.3570,            Elapsed Time: 5.60 sec
Epoch [2/5],            Loss: 0.1987,            Elapsed Time: 4.78 sec
Epoch [3/5],            Loss: 0.2924,            Elapsed Time: 5.27 sec
Epoch [4/5],            Loss: 0.3177,            Elapsed Time: 5.20 sec
Epoch [5/5],            Loss: 0.1298,            Elapsed Time: 4.73 sec
```

V.2.b) Execute a célula de treinamento novamente e observe o comportamento da Loss. 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?

Executar a célula não vai começar de um modelo aleatório, e sim de um modelo que já passou pelo treinamento. Assim, a loss antes de iniciar o treinamento será igual a última loss do treinamento anterior. Isso pode ser visto na saída abaixo:

```
Saída
Loss antes de iniciar o treinamento: 0.19223326541900634
Epoch [1/5],            Loss: 0.1341,            Elapsed Time: 4.99 sec
Epoch [2/5],            Loss: 0.2315,            Elapsed Time: 4.75 sec
Epoch [3/5],            Loss: 0.1054,            Elapsed Time: 5.63 sec
Epoch [4/5],            Loss: 0.1476,            Elapsed Time: 4.76 sec
Epoch [5/5],            Loss: 0.1296,            Elapsed Time: 5.54 sec
```

Para iniciar com um modelo aleatório é necessário executar novamente a célula que inicializa o modelo, a célula que possui a instrução abaixo:

model = OneHotMLP(vocab_size)

## V.3 Modificando a rede para gerar dois logitos no lugar de 1

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

A equação de CrossEntropy fornecida é:

$$\text{CrossEntropyLoss}(y, \hat{y}) = -\frac{1}{MN} \sum_{i=1}^{M} \sum_{j=1}^{N} y_{ij} \log(\hat{y}_{ij}),$$

Para um caso genérico com M classes e usando a função Softwmax, $\hat{y}_i$ será um vetor de tamanho M de forma que a soma de seus elementos é 1. Cada elemento indica uma classe. Considerando que, em um modelo iniciado mas não treinado todas as classes tem a mesma probabilidade de ser selecionada, temos que:

$$\hat{y}_i = \frac{1}{M} \quad \dots \quad \frac{1}{M}$$

Assim:

$$CrossEntropyLoss = -\frac{1}{MN}\sum_{i=1}^{M}\sum_{j=1}^{N} y_{ij}\log\left(\hat{y}_{ij}\right)$$

$$CrossEntropyLoss = -\frac{1}{MN}\sum_{i=1}^{M}\sum_{j=1}^{N} y_{ij}\log\left(\frac{1}{M}\right)$$

$$CrossEntropyLoss = -\frac{\log\left(\frac{1}{M}\right)}{MN}\sum_{i=1}^{M}\sum_{j=1}^{N} y_{ij} = \frac{\log\left(M\right)}{MN}\sum_{i=1}^{M}\sum_{j=1}^{N} y_{ij}$$

Como $y_i$ é o valor do label, essa variável será um vetor de tamanho M com (M-1) elementos iguais a 0 e um elemento igual a 1. A posição do elemento igual a 1 indicará a classe a que pertence a amostra. Dessa forma, o somatório duplo se reduz a N.

$$CrossEntropyLoss = \frac{\log\left(M\right)}{MN}\sum_{i=1}^{M}\sum_{j=1}^{N} y_{ij} = \frac{\log\left(M\right)}{MN}N$$

$$CrossEntropyLoss = \frac{\log\left(M\right)}{M} = \sim 0{,}3465 \text{ (para M = 2)}$$

De acordo com essa equação, o resultado teórico é 0,3465. Ou seja, é metade do resultado com o cálculo usando 1 logito.

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)

Para facilitar o trabalho, alterei o modelo para receber um novo parâmetro no construtor (n_logito), que indica a quantidade de logitos na saída:

Código

```
class OneHotMLP(nn.Module):
    def __init__(self, vocab_size, n_logitos = 1):
        super(OneHotMLP, self).__init__()

        self.fc1 = nn.Linear(vocab_size+1, 200)
        self.fc2 = nn.Linear(200, n_logitos)

        self.relu = nn.ReLU()

    def forward(self, x):
        o = self.fc1(x.float())
        o = self.relu(o)
        return self.fc2(o)

# Model instantiation
```

```
model = OneHotMLP(vocab_size, 2)
```

Com isso, podemos fazer o cálculo manual de acordo com a equação fornecida:

Código

```
# Inicializa um dataloader
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
# Pega o primeiro batch
inputs, y = next(iter(train_loader))
# Inicializa um modelo
model = OneHotMLP(vocab_size, 2)
model.to(device)

# Calcula a saída do model
logit = model(inputs.to(device))
prob = nn.functional.softmax(logit, dim=1)
yhat = prob

# Y é um vetor de 0 ou 1, contém só a classe. Cria um outro vetor derivado dele
# de tamanho 2,sendo que,se o elemento correspondente for 0,o novo será [1, 0].
# Se o elemento original for 1, o novo será [0, 1]
y_2_logitos = torch.zeros((len(y), 2))
y_2_logitos[y == 0, 0] = 1  # Defina [1, 0] nos locais onde o y é 0
y_2_logitos[y == 1, 1] = 1  # Defina [0, 1] nos locais onde o y é 1

# Cálculo da loss:
N, M = prob.shape
loss = (-1/(M*N)) * (y_2_logitos.to(device) * torch.log(yhat)).sum()
print(loss)
```

Saída

```
tensor(0.3459, device='cuda:0', grad_fn=<MulBackward0>)
```

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

Código

```
# Crie uma instância da função de perda CrossEntropyLoss
criterion = nn.CrossEntropyLoss()

# Calcule a loss
loss = criterion(logit.to(device), y.to(device))

print("Loss:", loss.item())
```

Saída

```
Loss: 0.6917370557785034
```

Os resultados não foram os mesmos. Na verdade, os resultados foram idênticos aos calculados usando 1 logito. Há duas possibilidades para essa diferença: ou eu errei na conta no item anterior ou a fórmula fornecida é diferente da fórmula implementada pelo algoritmo nn.CrossEntropyLoss.

Ao pesquisar no Google, encontrei algumas páginas que indicam que a constante multiplicativa é (1/N), e não (1/MN). Por exemplo:

$$CrossEntropyLoss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{i,j} \cdot \log\left(\frac{e^{x_{i,j}}}{\sum_{k=1}^{C} e^{x_{i,k}}}\right)$$

Fonte: https://theoclark.co.uk/posts/loss-functions-in-pytorch.html

$$logloss = -\frac{1}{N}\sum_{i}^{N}\sum_{j}^{M} y_{ij}\log(p_{ij})$$

Fonte: https://www.analyticsvidhya.com/blog/2021/03/binary-cross-entropy-log-loss-for-binary-classification/

$$L\left(y,\hat{y}\right) = -\frac{1}{N}\sum_{i}^{N}\sum_{j}^{C} y_{ij}log\left(\hat{y}_{ij}\right)$$

Fonte: https://www.shiksha.com/online-courses/articles/cross-entropy-loss-function/

Como a diferença dos resultados é exatamente 1/M (0,5) e há essas fórmulas na internet, é possível que a fórmula fornecida esteja errada. Mas também é possível que eu tenha calculado errado.

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.

Alteração na seção V

```
import time

def train_model(model, lr=0.001, num_epochs=5, n_logitos = 1):
    model = model.to(device)
    # Define loss and optimizer
    if n_logitos == 1:
        criterion = nn.BCEWithLogitsLoss()
    else:
        criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=lr)

    loss_por_epoca = []

    print(f'Loss antes de iniciar o treinamento: {calcula_loss(model,
            criterion, n_logitos)}')
    # Training loop
    for epoch in range(num_epochs):
        start_time = time.time()  # Start time of the epoch
        model.train()
        for inputs, labels in train_loader:
            inputs = inputs.to(device)
            labels = labels.to(device)
            # Forward pass
            outputs = model(inputs)
            if n_logitos == 1:
                # Para usar com nn.BCEWithLogitsLoss():
                loss = criterion(outputs.squeeze(), labels.float())
            else:
                # Para usar com nn.CrossEntropyLoss():
                loss = criterion(nn.functional.softmax(outputs, dim=1), labels)
```

```
            # Backward and optimize
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        end_time = time.time()  # End time of the epoch
        epoch_duration = end_time - start_time  # Duration of epoch

        print(f'Epoch [{epoch+1}/{num_epochs}], \
                Loss: {loss.item():.4f}, \
                Elapsed Time: {epoch_duration:.2f} sec')
        loss_por_epoca.append(loss.item())
    return loss_por_epoca

n_logitos = 2
model = OneHotMLP(vocab_size, n_logitos)
train_model(model, lr=0.1, num_epochs=5, n_logitos=2)
```

Saída

```
tensor(0.3459, device='cuda:0', grad_fn=<MulBackward0>)
```

Código na seção VI

```
## evaluation

def eval_model(model, n_logitos):
  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)
          if n_logitos == 1:
              # Para usar com nn.BCEWithLogitsLoss():
              predicted = torch.round(torch.sigmoid(outputs.squeeze()))
          else:
              # Para usar com nn.CrossEntropyLoss():
              probabilities = nn.functional.softmax(outputs, dim=1)
              _, predicted = torch.max(probabilities, 1)

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

      print(f'Test Accuracy: {100 * correct / total}%')

      return 100 * correct / total

eval_model(model, n_logitos)
```

Saída

```
Test Accuracy: 87.46%
87.46
```

# VI Avaliação

## VI.1 Avaliação

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

Há 25.000 amostras no test_data, que são carregadas em 196 batchs:

```
Código
# Número de amostras de test_loader
print(len(test_data))
print(len(test_loader))
Saída
25000
196
```

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

ChatGPT:

Os comandos model.eval() e with torch.no_grad() são usados durante a avaliação de modelos no PyTorch para desativar o cálculo de gradientes e alterar o comportamento do modelo.

- model.eval(): Este método é usado para colocar o modelo em modo de avaliação. Quando um modelo está em modo de avaliação, ele altera o comportamento de certas camadas, como BatchNorm e Dropout. Por exemplo, a camada BatchNorm usa a média e o desvio padrão dos dados de treinamento durante o treinamento, mas durante a avaliação, ele usa média e desvio padrão estimados durante o treinamento. Da mesma forma, a camada Dropout desativa aleatoriamente unidades durante o treinamento para evitar o overfitting, mas durante a avaliação, ele mantém todas as unidades ativas. Portanto, colocar o modelo em modo de avaliação garante que essas camadas se comportem corretamente durante a avaliação.
- with torch.no_grad(): Este é um contexto que desabilita o cálculo de gradientes no PyTorch. Isso é útil durante a avaliação, quando você não está interessado em calcular gradientes. Desabilitar o cálculo de gradientes economiza memória e acelera a computação, pois o PyTorch não precisa manter informações para calcular gradientes durante a avaliação. Usar este contexto também é útil quando você está apenas fazendo inferências com o modelo e não pretende ajustar os pesos com base nos resultados da avaliação.

Em resumo, model.eval() garante que o modelo está no modo de avaliação, alterando o comportamento de certas camadas, e with torch.no_grad() desabilita o cálculo de gradientes durante a avaliação para economizar memória e acelerar a computação. Ambos são comumente usados juntos durante a avaliação de modelos no PyTorch.

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?

Sim. A sigmóide retorna 0.5 se o argumento for 0. Se o argumento for maior que 0, retorna um número maior que 0.5, que é interpretado como classe 1. Se o argumento for menor que 0, retorna um número menor que 0.5, que é interpretado como classe 0. Assim, em vez de calcular a sigmóide, basta verificar se o resultado do modelo é menor ou maior que 0:

```
Código antigo
predicted = torch.round(torch.sigmoid(outputs.squeeze()))
Código novo
predicted = (outputs.squeeze() >= 0).int()
```

## VI.2 Perplexidade como métrica de avaliação

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?

A loss para entropia cruzada calculada em V.1.a foi de ln(2). A perplexidade será **exp(ln(2)) = 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?

No caso de N classes, a loss teórica calculada com a fórmula foi de ln(N)/N. Entretanto, o valor correto (conforme cálculo usando a função nn.CrossEntropyLoss) é ln(N). Assim, temos que a perplexidade para o caso geral é **exp(ln(N)) = N**.

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

Se o modelo sempre acerta, a loss é 0. Nesse caso, a perplexidade é **exp(0) = 1**.

## VI.3 Código para cálculo da perplexidade

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.

Como já havia criado uma função para o cálculo da loss no conjunto de treinamento, apenas alterei a função para receber o conjunto que deve ser usado (treinamento ou teste) e passei a retornar também a perplexidade:

Código

```
import math

# Gerado com ajuda do chatGPT
def calcula_loss_e_perplexidade(model, criterion, n_logitos, loader):
  with torch.no_grad():  # Garante que nenhum gradiente seja calculado
    model.eval()  # Coloca o modelo no modo de avaliação (não treinamento)
    total_loss = 0.0
    total_samples = 0
    for inputs, labels in loader:
        inputs = inputs.to(device)
        labels = labels.to(device)
        # Forward pass
        outputs = model(inputs)
        # Calcula a perda
        if n_logitos == 1:
            #Para usar com nn.BCEWithLogitsLoss():
            loss = criterion(outputs.squeeze().to(device),
labels.float()).to(device)
        else:
            # Para usar com nn.CrossEntropyLoss():
            loss = criterion(nn.functional.softmax(outputs, dim=1), labels)
        # Acumula a perda e o número total de amostras
        total_loss += loss.item() * inputs.size(0)
        total_samples += inputs.size(0)

  loss = total_loss / total_samples
  ppl = math.exp(loss)
  return loss, ppl
```

Agora podemos chamar essa função no laço de avaliação:

Código

```
## evaluation

def eval_model(model, n_logitos):
  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)
          if n_logitos == 1:
              # Para usar com nn.BCEWithLogitsLoss():
              # predicted = torch.round(torch.sigmoid(outputs.squeeze()))
              # Podemos calcular isso de forma mais simples assim:
              predicted = (outputs.squeeze() >= 0).int()
          else:
              # Para usar com nn.CrossEntropyLoss():
              probabilities = nn.functional.softmax(outputs, dim=1)
              _, predicted = torch.max(probabilities, 1)

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

      print(f'Test Accuracy: {100 * correct / total}%')

      # Alterações para calcular a loss e a perplexidade:
      if n_logitos == 1:
          criterion = nn.BCEWithLogitsLoss()
      else:
          criterion = nn.CrossEntropyLoss()
      loss, ppl = calcula_loss_e_perplexidade(model, criterion, n_logitos,
                  test_loader)
      print(f'Loss: {loss}')
      print(f'Perplexidade: {ppl}')
      return 100 * correct / total, loss, ppl

eval_model(model, n_logitos)
```

Saída

```
Test Accuracy: 88.408%
Loss: 0.28558039922714235
Perplexidade: 1.3305340449922043
(88.408, 0.28558039922714235, 1.3305340449922043)
```

A perplexidade após o treinamento foi de ~1.33.

## VI.4 Observando overfitting

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. 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.

Alterei o loop de treinamento para passar a salvar a perplexidade e a loss por época. Fiz os cálculos em 15 épocas mantendo 25mil registros de treinamento:

Código

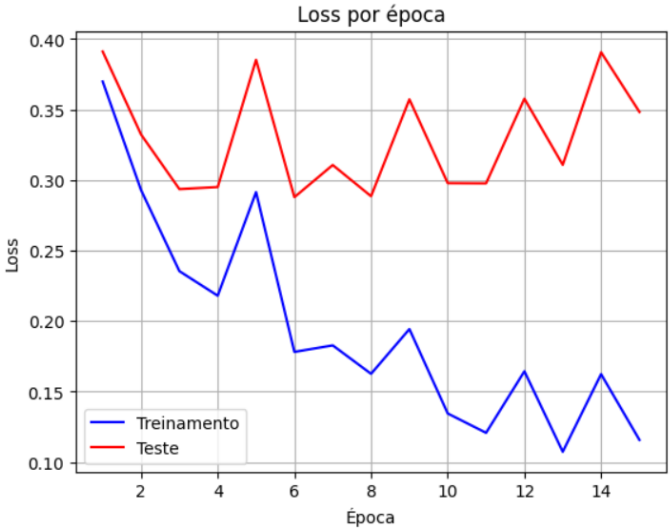
```
import time

def train_model(model, lr=0.001, num_epochs=5, n_logitos = 1):
    model = model.to(device)
    # Define loss and optimizer
    if n_logitos == 1:
        criterion = nn.BCEWithLogitsLoss()
    else:
        criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=lr)

    loss_por_epoca = []

    loss_ppl_treinamento_por_epoca = []
    loss_ppl_teste_por_epoca = []

    print(f'Loss e perplexidade antes de iniciar o treinamento:
        {calcula_loss_e_perplexidade(model, criterion, n_logitos,
                train_loader)}')
    # Training loop
    for epoch in range(num_epochs):
        start_time = time.time()  # Start time of the epoch
        model.train()
        for inputs, labels in train_loader:
            inputs = inputs.to(device)
            labels = labels.to(device)
            # Forward pass
```

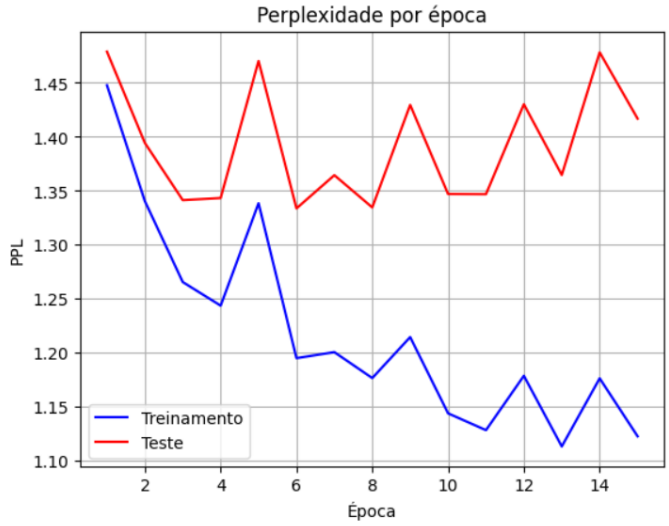
```
            outputs = model(inputs)
            if n_logitos == 1:
                # Para usar com nn.BCEWithLogitsLoss():
                loss = criterion(outputs.squeeze(), labels.float())
            else:
                # Para usar com nn.CrossEntropyLoss():
                loss = criterion(nn.functional.softmax(outputs, dim=1), labels)
            # Backward and optimize
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        end_time = time.time()  # End time of the epoch
        epoch_duration = end_time - start_time  # Duration of epoch

        print(f'Epoch [{epoch+1}/{num_epochs}], \
                Loss: {loss.item():.4f}, \
                Elapsed Time: {epoch_duration:.2f} sec')
        loss_por_epoca.append(loss.item())

        loss_ppl_treinamento_epoca_i = calcula_loss_e_perplexidade(model,
                criterion, n_logitos, train_loader)
        loss_ppl_treinamento_por_epoca.append(loss_ppl_treinamento_epoca_i)
        print(f'Loss e perplexidade treinamento:
                {loss_ppl_treinamento_epoca_i}')

        loss_ppl_teste_epoca_i = calcula_loss_e_perplexidade(model, criterion,
                n_logitos, test_loader)
        loss_ppl_teste_por_epoca.append(loss_ppl_teste_epoca_i)
        print(f'Loss e perplexidade teste: {loss_ppl_teste_epoca_i}')

    return loss_ppl_treinamento_por_epoca, loss_ppl_teste_por_epoca

n_logitos = 1
model = OneHotMLP(vocab_size, n_logitos)
loss_ppl_treinamento_por_epoca, loss_ppl_teste_por_epoca = train_model(model,
lr=0.1, num_epochs=15, n_logitos=n_logitos)
```

Em seguida, pedi pro ChatGPT criar uma função para plotar os valores da perplexidade e loss por época:

Código

```
import matplotlib.pyplot as plt

def plotar_dois_graficos(x, y1, y2, titulo, label1, label2, cor1='blue',
cor2='red', xlabel='X', ylabel='Y', grid=True):
    # Configurações do primeiro gráfico
    plt.figure()
    plt.plot(x, y1, color=cor1, label=label1)

    # Configurações do segundo gráfico
    plt.plot(x, y2, color=cor2, label=label2)

    # Adicionando título e legendas
    plt.title(titulo)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.legend()

    # Adicionando grid se necessário
    if grid:
        plt.grid(True)

    # Exibindo o gráfico
    plt.show()

epocas = range(1, len(loss_ppl_treinamento_por_epoca)+1)
loss_treinamento, ppl_treinamento = zip(*loss_ppl_treinamento_por_epoca)
loss_teste, ppl_teste = zip(*loss_ppl_teste_por_epoca)

plotar_dois_graficos(epocas, loss_treinamento, loss_teste, titulo='Loss por
época', label1='Treinamento', label2='Teste', xlabel='Época', ylabel='Loss')
plotar_dois_graficos(epocas, ppl_treinamento, ppl_teste, titulo='Perplexidade
por época', label1='Treinamento', label2='Teste', xlabel='Época', ylabel='PPL')
```




Como pode ser observado, mesmo a loss/perplexidade de treinamento reduzindo com o passar das épocas, a loss/perplexidade no conjunto de teste começa a aumentar. É um sinal de que o modelo está se adequando aos dados de treinamento e não generalizando (overfitting).