Processo Seletivo para Disciplina IA-024 1S2024 FEEC-UNICAMP
versão atualizada em 15/fev/2024 - errata equação Entropia Cruzada com C classes

Estes exercícios do processo seletivo servem para verificar os conhecimentos dos candidatos sobre conceitos básicos de Machine Learning, Redes Neurais e Processamento de Linguagem Natural, além de experiência de programação avançada Python utilizando as bibliotecas NumPy e PyTorch.

O curso é teórico prático onde cada semana o aluno precisa entregar: Resumo de uma leitura de artigo; Teste inicial no início da aula; e Entrega de um notebook colab. O curso é muito interativo e quase um curso estilo "invertido" - o aluno estuda, faz os exercícios em casa e em classe nós discutimos os exercícios olhando as soluções dos colegas e aprendendo com os acertos e erros. Sabemos que aprendemos mais com os erros do que com os acertos. É um curso colaborativo onde aprendemos com as experiências e soluções dos colegas.

Prazo para Entrega dos Exercícios

Os exercícios devem ser entregues no prazo de 2 semanas, até domingo dia 18 de fevereiro, 23h55.

Descrição e objetivos dos exercícios do processo seletivo.

Iremos passar um código python/pytorch no colab de um modelo preditivo para fazer análise de sentimentos utilizando Bag-of-Words das críticas dos usuários do IMDB, usando uma rede neural MLP muito simples de duas camadas. O objetivo é saber se a crítica do usuário é positiva ou negativa sobre o filme que ele assistiu. A métrica utilizada é a acurácia, isto é, o número de acertos do modelo preditivo dividido pelo número total de predições.
O notebook tem as seguintes partes: I - Vocabulário e Tokenização; II - Dataset; III - Data Loader; IV - Modelo; V - Laço de Treinamento; VI - Avaliação. O notebook tem propositalmente pouquíssimos comentários.

Objetivo da tarefa: Verificar o conhecimento do aluno em aprendizado de máquina por treinamento supervisionado utilizando redes neurais e a minimização da função de perda utilizando técnica do gradiente descendente e a capacidade do aluno em entender esses conceitos na familiarização do código python/pytorch exemplificado no notebook colab oferecido. Espera-se que o aluno entenda em profundidade o que faz cada linha do código do notebook colab.

Utilização do chatGPT:

Com a disponibilidade do chatGPT, os candidatos são incentivados a usarem ao extremo o chatGPT seja para explicar código, escrever código, escrever exemplos de teste, explicar conceitos, enfim tudo o que for necessário.
Estamos procurando alunos que conseguem melhor utilizar o chatGPT para o seu aprendizado e a sua eficiência nas entregas. A tecnologia do chatGPT veio para ficar e é importante que os alunos sejam familiarizados com ele e saibam explorá-lo na sua plenitude. Acreditamos que bons alunos que saibam programar bem e conheçam

programação orientada a objeto e não tenham experiência em redes neurais serão capazes de entender os principais conceitos e conseguir resolver os exercícios com a ajuda do chatGPT.
Lembrar que é normal que o chatGPT erre em muitas das respostas, porém através de solicitação de explicações ou reformulação de perguntas, é possível aprender e fazer um bom uso do chatGPT.
Por exemplo, é possível você colocar trechos do código passado no colab no chatGPT e pedir para ele explicar o código. Por exemplo: "Explique o código python/pytorch a seguir" e o chatGPT irá explicar conceitualmente em alto nível como ele funciona. Entretanto, não é apenas isso que queremos, queremos que você tenha familiaridade com todas as funções do programa. Para isso, você pode pedir ao chatGPT para ele gerar um código simples para mostrar o funcionamento de uma função do pytorch.
Por exemplo: "Faça um exemplo ilustrativo bem simples do uso do nn.Linear. (esta talvez seja uma das classes mais difíceis do notebook todo). Algumas perguntas que podem ajudar: "Qual é a equação matemática da classe nn.Linear? Outra ainda: "Por que nn.Linear só tem a primeira letra em maiúscula?". Queremos também que você entenda a equação da função de Perda utilizada na minimização. Você pode pedir ao chatGPT para ele mostrar equações das funções implementadas no pytorch. Por exemplo: Mostre a equação da função loss do BCEWithLogitsLoss. e modifique o programa para imprimir o valor da loss antes de começar o treinamento e confira com o valor teórico esperado pela equação.

Executando o notebook

O notebook está pronto para ser executado. Se você executá-lo completamente, você deverá obter valores parecidos com estes:
Runtime rodando com CPU apenas

- Tempo de execução de cada época: da ordem de 45 segundos

Runtime rodando com GPU T4:

- Tempo de execução de cada época: da ordem de 29 segundos

Desempenho do modelo:(a cada execução, o resultado será um pouco diferente)

- Loss de treinamento após primeira época: próximo de 0.68
- Loss de treinamento após 5 épocas: caindo muito pouco de 0.68
- Acurácia no conjunto de teste: 54% a 62%

Ao longo dos exercícios, você conseguirá reduzir muito o tempo de execução e aumentar significativamente a acurácia do modelo, ao mesmo tempo que você terá um conhecimento mais aprofundado dos conceitos básicos de treinamento de redes neurais e da programação usando PyTorch.

A primeira coisa a ser feita é diminuir drasticamente o tempo de execução para podermos analisar o código e fazer experimentos com maior eficiência. Para isso, o mais simples é diminuir drasticamente o número de amostras do dataset.
Para podermos fazer isso, é preciso entender bem as seções I e II do notebook:
Vocabulário e Dataset para entender como o dataset IMDB está organizado e como o vocabulário e o dataset serão construídos.

Exercícios a serem entregues:

I - Vocabulário e tokenização

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

Resposta esperada:

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

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

counter = Counter()
count_positivo = 0
count_negativo = 0
tamanho_total = 0
for (label, line) in list(IMDB(split='train')):
   counter.update(line.split())
   if label == 1:
     count_positivo += 1
   else:
     count_negativo += 1
   # print(f'tamanho: {len(line)} de {line}' )
   tamanho_total += len(line.split())

print(f'Total de amostras: {count_negativo+count_positivo}')
print(f'Amostras positivas: {count_positivo}')
print(f'Amostras negativas: {count_negativo}')
print(f'Tamanho médio do texto: {tamanho_total/(count_negativo+count_positivo)}')

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

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

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

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

```
Tamanho médio do texto: 233.7872
```

I.2. As linhas 9 e 10 da célula do vocabulário são linhas típicas de programação python em listas com dicionários com laços na forma compreensão de listas ou *list comprehension* em inglês. Procure analisar e estudar profundamente o uso de lista e dicionário do python. Estude também a função *encode_sentence.*

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.

Resposta esperada:

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

```
print(f'Cinco palavras mais frequentes: {counter.most_common(5)}')
print(f'Cinco palavras menos frequentes: {counter.most_common()[:-6:-1]}')
```

```
Cinco palavras mais frequentes: [('the', 287032), ('a', 155096), ('and', 152664), ('of', 142972),
('to', 132568)]
Cinco palavras menos frequentes: [('Crocker)', 1), ('McKenzie(Barry', 1), ('shearer', 1),
('grossest', 1), ('unemployed...', 1)]
```

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

```
0 (Out Of Vocabulary): assumido no comando get se word não estiver no vocab
```

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

```
Número de tokens desconhecidas: 260617
Número total de ocorrências de tokens desconhecidas: 566141
```

```
def count_unknown_tokens(vocab, list_word):
  """
     Imprime o número de tokens distintas presentes em list_word desconhecidas no vocab
     e o número total de ocorrências delas
  """
  unknown_tokens = set()  # Conjunto para armazenar palavras desconhecidas
  total_occurrences = 0  # Contador para o número total de ocorrências

  for word in list_word:
     if word not in vocab:
       unknown_tokens.add(word)  # Adiciona a palavra desconhecida ao conjunto
       total_occurrences += 1  # Incrementa o contador
     #elif word in unknown_tokens:
     #  total_occurrences += 1  # Incrementa o contador

  print(f"Número de tokens desconhecidas: {len(unknown_tokens)}")
  print(f"Número total de ocorrências de tokens desconhecidas: {total_occurrences}")

count_unknown_tokens(vocab, list_word_treino)
```

Reduzindo o número de amostras para 200

Uma forma simples de reduzir o número de amostras é utilizar o fatiamento de listas para selecionar apenas as primeiras 200 amostras utilizando [:200] na lista do IMDB: `list(IMDB(split='train'))[:200]`.

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

Enunciado do exercício:
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?

porque os 200 primeiros registros de treinamento são todos positivos. E os primeiros 200 dados de teste também. Daí o modelo aprendeu a prever positivo. E os testes apenas testaram se era positivo.

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.

```
class IMDBDatasetBalanced(Dataset):
  def __init__(self, split, vocab):
    self.data = list(IMDB(split=split))
    self.vocab = vocab

    # Separa os exemplos 100 por classe
    positive_examples = [item for item in self.data if item[0] == 1][:100]
    negative_examples = [item for item in self.data if item[0] != 1][:100]

    print('negative')
    for label, input in negative_examples:
      print(label)
      break

    print('positive')
    for label, input in positive_examples:
      print(label)
      break

    # Combina os exemplos das duas classes
    self.data = positive_examples + negative_examples

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

  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)

train_data_lim_200_balanced = IMDBDatasetBalanced('train', vocab_lim_200)
test_data_lim_200_balanced = IMDBDatasetBalanced('test', vocab_lim_200)

label_counts = {}

for _, label in train_data_lim_200_balanced:
    if label.item() not in label_counts:
        label_counts[label.item()] = 0
    label_counts[label.item()] += 1

# Imprime o total por valor em label
for value, count in label_counts.items():
    print(f"Label {value}: {count} exemplos")


OUPTUT
Label 1: 100 exemplos
Label 0: 100 exemplos
```

II - Dataset

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

Em primeiro lugar precisamos entender qual será a entrada da rede neural para decidir se o texto é uma crítica positiva ou negativa. Uma das formas mais simples de construir um modelo preditivo é com base nas palavras utilizadas no texto. A distribuição das palavras de um texto tem alta correlação com o fato do texto estar falando bem ou falando mal de um filme. Certamente é estimativa que possui seus erros, mas é a forma mais simples e eficiente de se fazer uma análise de sentimento ou de maneira geral uma classificação de um texto. Esse método é denominado "Bag of Words". A entrada da rede neural, para cada amostra, será um vetor de comprimento do vocabulário, com valores todos zero, com exceção dos tokens que aparecem no texto da amostra. Esse método de codificação é também denominado "One-Hot". Estude o código da classe `IMDBDataset` fazendo experimentos e perguntas ao chatGPT para entender com profundidade esta classe.

Enunciado do exercício:
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.

```
Label 1: 12500 exemplos
Label 0: 12500 exemplos
```

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

A média de palavras em cada vertor é 133.09548

Em I.1.c o tamanho médio era 233.78 palavras por sentença.
O valor menor 133.09 é explicado por dois motivos:
1. A não contagem de repetições (trata-se de um bag of words binário, com valores 0 ou 1 conforme o token esteja na sentença, e não com quantidade de ocorrências do token, seria uma versão "count").
2. Como o vocabulário só tem 20000 tokens diferentes, uma grande parte de tokens distintas (260617, conforme exercício I.2.C) não foram contadas. Na realidade todas elas foram consideradas como uma única token (out of vocabulary) na posição 0 do tensor.

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

Aumentando a eficiência do treinamento com o uso da GPU T4

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

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

Enunciado do exercício: 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.

(soma dos 2 laços)

Tempo total=0.321; Tempo de movimentação para gpu = 0.02 (7%); Tempo do forward = 0.0011 (3%); Tempo do backward = 0.0014 (4%);

Conclusão

O tempo de percorrimento do dataloader é o maior gargalo, aproximadamente 86%.

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

Esse loop precisa ser otimizado! (ver mais detalhes na solução que se segue em II.2.c.)

```
# Dataset Class with One-hot Encoding
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.

Primeiro troquei o encode_sentence por return_vocab_in_sentence que não passa mais de uma vez em repetições de uma palavra (usado set).

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

encode_sentence("I like Pizza", vocab)
```

Depois, retirado loop

```
    for word in encode_sentence(line, self.vocab):
      X[word] = 1
```

para

```
    X[encode_sentence(line, self.vocab) vocab)] = 1
```

E, por fim, como o conjunto de dados não é muito grande, optei por carregar os tensores no __init__ e, em contrapartida, haverá economia de memória ao não mais armazenar os textos (self.data). Segue código otimizado.

Obs.: fora do escopo deste exercício a avaliação se haverá ou não economia de memória nessa troca. Mas, de tempo, com certeza (cache de valores calculados).

Cógigo final

```
%%time
```

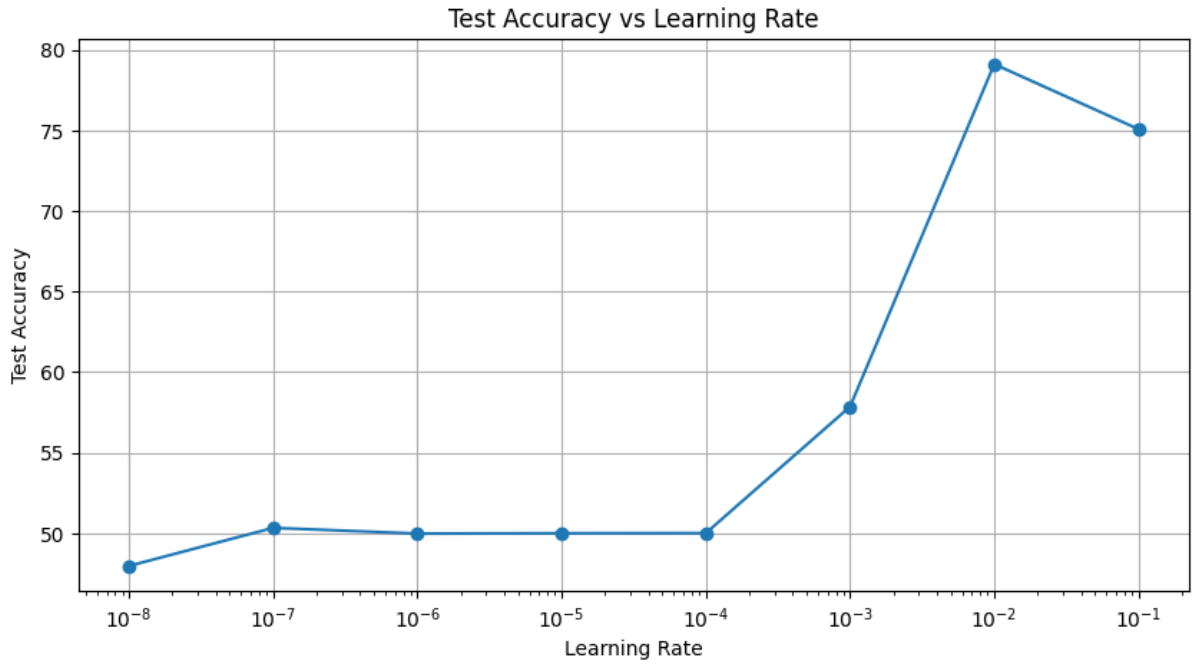
```
from torch.nn.functional import one_hot
# Dataset Class with One-hot Encoding
class IMDBDataset(Dataset):
    def __init__(self, split, vocab):
        data = list(IMDB(split=split))
        self.vocab = vocab
        self.labels = [torch.tensor(1) if item[0] == 1 else torch.tensor(0) for item in data]

        self.sentences = []
        for label, line in data:
            X = torch.zeros(len(self.vocab) + 1)
            X[return_vocab_in_sentence(line, self.vocab)] = 1
            self.sentences.append(X)

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

    def __getitem__(self, idx):
        # print('retornando idx', idx, self.sentences[idx], self.labels[idx])
        return self.sentences[idx], self.labels[idx]

# Load Data with One-hot Encoding
train_data = IMDBDataset('train', vocab)
test_data = IMDBDataset('test', vocab)

Resultado
Tempo total nos 2 laços caiu de 0.321 para 0.034, para cerca de 10%.

Tempo finais (apenas para comparativo dentro do loop):
 Movimentação para gpu = 0.02 (70%);
Tempo do forward = 0.0008 (2.5%);
Tempo do backward = 0.0015 (4,6%);
Percorrimento dataloader: 22,9%
```

Após esta otimização, é esperado que o tempo de processamento de cada época caia tanto para execução em CPU (da ordem de 10 segundos por época) como para GPU (da ordem de 1 a 2 segundos por época). Isso utilizando as 25 mil amostras do dataset IMDB inteiro.

Agora que a execução está bem mais otimizada em tempos de execução, mantenha o dataset completo: 25 mil amostras e vamos analisar um outro fator importante que é a escolha do LR (*Learning Rate*)

Escolhendo um bom valor de LR

Enunciado do exercício: 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.
II.3.a) Gráfico Acurácia vs LR



II.3.b) Valor ótimo do LR

```
best_lr = max(test_accuracies_by_lr, key=test_accuracies_by_lr.get)
print(f'The learning rate with the highest test accuracy is {best_lr}')

OUTPUT
The learning rate with the highest test accuracy is 0.01
```

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

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

Nesta equação:

(θ) - são os parâmetros (ou pesos) do modelo

(η) é a taxa de aprendizado (LR).

$(\nabla_\theta J(\theta))$ é o gradiente da função de perda J com relação aos parâmetros (θ).

O papel do LR no ajuste dos parâmetros do modelo da rede neural é determinar o tamanho do passo em cada iteração do gradiente descendente. Um LR alto pode fazer com que o algoritmo de otimização dê passos maiores e possivelmente pule o mínimo global. Por outro lado, um LR muito baixo pode fazer com que o algoritmo de otimização dê passos muito pequenos, o que pode resultar em um tempo de treinamento muito longo ou o algoritmo pode ficar preso em

um mínimo local. Portanto, a escolha do LR é um compromisso entre a velocidade de treinamento e a capacidade do modelo de encontrar o mínimo global da função de perda.

Otimizando o tokenizador

Agora que a convergência da Loss está melhor, vamos experimentar os parâmetros do tokenizador, isto é, como as palavras estão codificadas em tokens.
Observe novamente o `vocab` criado na parte **I - Vocabulário e Tokenização**. Perceba como as pontuações estão influenciando nos tokens criados e como o uso de letras maiúsculas e minúsculas também podem atrapalhar a consistência dos tokenizador em representar o significado semântico das palavras. Experimente rodar o `encode_sentence` com frases que tenham pontuações e letras maiúsculas e minúsculas. Baseado nessas informações, procure melhorar a forma de tokenizar o dataset.
Enunciado do exercício: Melhores a forma de tokenizar, isto é, pré-processar o dataset de modo que a codificação seja indiferente das palavras serem escritas com maiúsculas ou minúsculas e sejam pouco influenciadas pelas pontuações.
II.4.a) Mostre os trechos modificados para este novo tokenizador, tanto na seção **I –**

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

counter = Counter()
for (label, line) in list(IMDB(split='train')):
  counter.update(generate_list_of_tokens(line))

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

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

encode_sentence("I like Pizza", vocab)

def return_vocab_in_sentence(sentence, vocab):
  return [vocab.get(word, 0) for word in set(generate_list_of_tokens(sentence))] # 0 for OOV

return_vocab_in_sentence("I like Pizza", vocab)

%%time
from torch.nn.functional import one_hot
# Dataset Class with One-hot Encoding
class IMDBDataset(Dataset):
  def __init__(self, split, vocab):
    data = list(IMDB(split=split))
    self.vocab = vocab
```

```
        self.labels = [torch.tensor(1) if item[0] == 1 else torch.tensor(0) for item in data]

        self.sentences = []
        for label, line in data:
            X = torch.zeros(len(self.vocab) + 1)
            X[return_vocab_in_sentence(line, self.vocab)] = 1
            self.sentences.append(X)

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

    def __getitem__(self, idx):
        # print('retornando idx', idx, self.sentences[idx], self.labels[idx])
        return self.sentences[idx], self.labels[idx]

# Load Data with One-hot Encoding
train_data = IMDBDataset('train', vocab)
test_data = IMDBDataset('test', vocab)
print(f'Tamanho de train_data {len(train_data)}')
print(f'Tamanho de test_data {len(test_data)}')
```

Vocabulário, como na seção **II - Dataset**.

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.

Palavras mais/menos frequentes

```
Antes

        Cinco palavras mais frequentes: [('the', 287032), ('a', 155096),
('and', 152664), ('of', 142972), ('to', 132568)]
        Cinco palavras menos frequentes: [('Crocker)', 1),
('McKenzie(Barry', 1), ('shearer', 1), ('grossest', 1), ('unemployed...',
1)]

Agora

        Cinco palavras mais frequentes: [('the', 334730), ('and',
162252), ('a', 161958), ('of', 145326), ('to', 135046)]
        Cinco palavras menos frequentes: [('mckenziebarry', 1),
('grossest', 1), ('dunny', 1), ('fitzgibbon', 1), ('snacka', 1)]
```

(original no caderno notebook)

	Tokens dif	Ocorrencias	Num médio tokens
Original	260617	566141	233.78
Preprocessada	100714	213699	232.83

Justificativa:

Com o novo tokenizador que, entre outras coisas, desconsidera diferenças de caso (upper/lower) e acentos, o número de tokens distintas acaba sendo menor em uma sentença. Por conseguinte, as palavras mais frequentes do vocabulário possuem mais ocorrências nos dados de treino. E, seguindo esse raciocínio, aumentam-se as chances de uma token de um texto estar no vocabulário. Por isso, a redução no número de tokens diferentes fora do vocabulário.
O tamanho médio ficou bem próximo até porque considera o número de ocorrências com repetições, sendo que o novo tokenizador se diferencia em reduzir o número de palavras distintas.

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.

(original no caderno notebook)

```
# Model instantiation
model = OneHotMLP(vocab_size)
model = model.to(device)
criterion = nn.BCEWithLogitsLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)
train_start_time = time.time()  # Start time of the epoch
test_accuracy = train_model(model, criterion, optimizer, num_epochs, train_loader, test_loader)

train_duration = time.time() - train_start_time  # Duration of epoch

print(f'Treino: lr={lr}; test_accuracy={round(test_accuracy,4)}  (train_duration {train_duration:.2f} sec)')
```

```
...  Epoch [1/5],                    Loss: 0.6799,                    Elapsed Time: 1.56 sec
     Epoch [2/5],                    Loss: 0.6358,                    Elapsed Time: 1.52 sec
     Epoch [3/5],                    Loss: 0.6176,                    Elapsed Time: 1.52 sec
     Epoch [4/5],                    Loss: 0.5465,                    Elapsed Time: 1.56 sec
     Epoch [5/5],                    Loss: 0.5116,                    Elapsed Time: 1.53 sec
     Treino: lr=0.1; test_accuracy=81.392  (train_duration 8.98 sec)
```

III - DataLoader

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:

Resposta (original no caderno notebook)

Shuffle	Acurácia (teste)
True	81.4
False	50

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

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

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

** Eficiência Computacional:*
O treinamento em batches permite processar várias amostras de dados ao mesmo tempo.
Isso é mais eficiente do que atualizar os pesos do modelo após cada exemplo individual.
Reduz a sobrecarga computacional, especialmente em GPUs, acelerando o treinamento.

** Estabilidade e Convergência do Treinamento:*
O cálculo do gradiente (derivada) da função de custo em relação aos parâmetros do modelo é mais estável com batches.
O gradiente é uma média das amostras no lote, o que reduz a variância.
Isso ajuda a evitar oscilações e convergência instável durante o treinamento.

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

1. Redução de Viés de Aprendizado (generalização melhorada):
Quando as amostras são apresentadas ao modelo em uma ordem específica, ele pode aprender a depender da sequência.
Por exemplo, se as primeiras amostras forem sempre de uma classe específica, o modelo pode se tornar tendencioso em relação a essa classe.
Embaralhar as amostras garante que o modelo não seja influenciado pela ordem de apresentação.
Embaralhar as amostras torna o treinamento mais robusto e ajuda o modelo a generalizar melhor para dados não vistos.

2. Estabilidade do Treinamento:
O embaralhamento aleatório das amostras introduz uma variabilidade natural no treinamento.
Isso ajuda a evitar que o modelo fique preso em mínimos locais ou em trajetórias de gradiente específicas.
Também ajuda a explorar diferentes partes do espaço de parâmetros.

Em resumo, o embaralhamento das amostras do batch é essencial para garantir que o modelo aprenda de forma imparcial, seja estável durante o treinamento e generalize bem para novos dados.

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

Como a acurácia é calculada sobre todas as previsões, a ordem em que as previsões são feitas não importa. Seja qual for a ordem em que as amostras são processadas, a acurácia final será a mesma.

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.

```
"""
O número de iterações é determinado pelo tamanho do conjunto de dados de treinamento e o
tamanho do lote (batch size).
"""

num_iterations = 0
for _ in train_loader:
   num_iterations += 1

print(f'The train_loader has {num_iterations} iterations.')
print(f"Equivale ao len(train_loader): {len(train_loader)} batches")

OUTPUT
The train_loader has 196 iterations.
Equivale ao len(train_loader): 196 batches
```

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`?

```
Ele pode ser menor do que o batch size. No caso, 40 < 132.

"""
O último lote pode ter menos amostras do que o tamanho do lote.
Isso ocorre quando (N) não é um múltiplo exato de (B).
"""

# Encontre o número total de amostras no conjunto de dados
total_samples = len(train_loader.dataset)

# Calcule o número de lotes
num_batches = len(train_loader)

# Calcule o tamanho do último lote
samples_in_last_batch = total_samples % batch_size

# Imprima os resultados
print(f"Número total de amostras: {total_samples}")
print(f"Número de lotes: {num_batches}")
print(f"Amostras no último lote: {samples_in_last_batch}")

OUTPUT
Número total de amostras: 25000
Número de lotes: 196
Amostras no último lote: 40

"""
Forma alternativa
"""

last_batch_size = 0
```

```
for batch in train_loader:
    last_batch_size = len(batch[0])  # Assuming batch is a tuple of (inputs, labels)

print(f'The last batch has {last_batch_size} samples.')
OUTPUT
The last batch has 40 samples.
```

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.

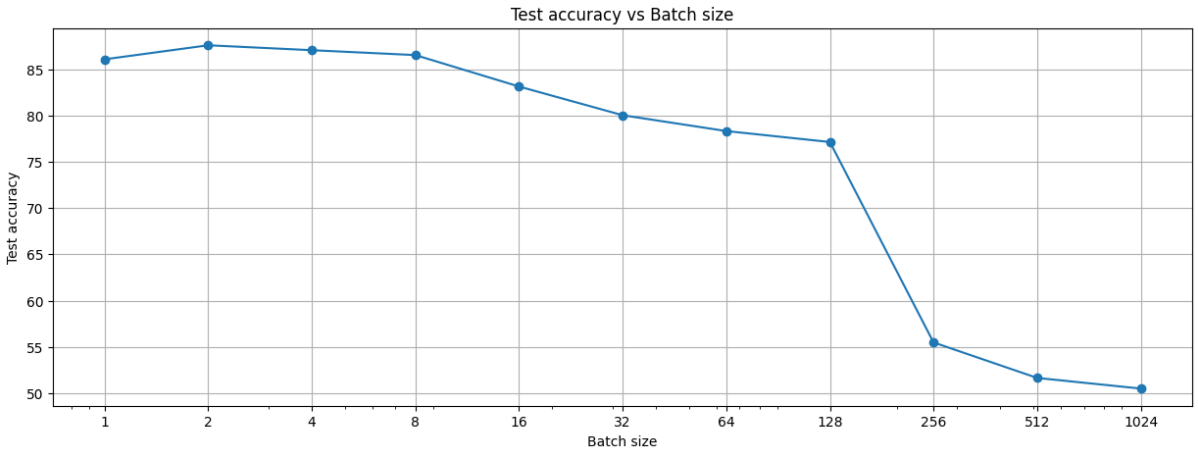
```
# Calculate the ratio R for each batch using a list comprehension
ratios = [torch.sum(labels == 1).item() / len(labels) for _, labels in
train_loader]

# Calculate the average ratio
average_ratio = sum(ratios) / len(ratios)

print(f'Valor de R (average ratio of positive samples) is
{average_ratio:.4f}')
OUTPUT
Valor de R (average ratio of positive samples) is 0.4998
```

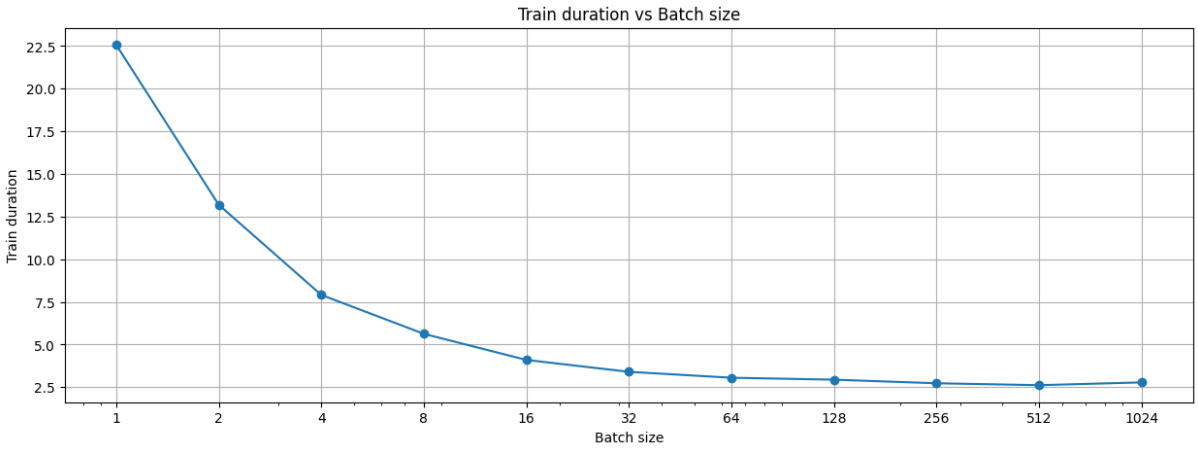
III.2.d) Mostre a estrutura de um dos *batches*. Cada *batch* foi criado no método `__getitem__` do **Dataset**, linha 20. É formado por uma tupla com o primeiro elemento sendo a codificação *one-hot* do texto e o segundo elemento o *label* esperado, indicando positivo ou negativo. Mostre o *shape* (linhas 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))`.

```
# Get the first batch from the train_loader
batch = next(iter(train_loader))

# The batch is a tuple of (inputs, labels)
inputs, labels = batch

# Print the shape and data type of the inputs and labels
print(f'Inputs shape: {inputs.shape}, type: {inputs.dtype}')
print(f'Labels shape: {labels.shape}, type: {labels.dtype}')

OUTPUT
Inputs shape: torch.Size([128, 20001]), type: torch.float32
Labels shape: torch.Size([128]), type: torch.int64
```

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

Alteradas funções para tratar batch_size == 1
outputs.view(-1) e labels.view(-1) irão remodelar outputs e labels para terem uma forma de (n,),
onde n é o número total de elementos em cada tensor. Isso garantirá que outputs e labels tenham a mesma forma,
independentemente do tamanho do batch.





A acurácia ficou maior para batchs menores. Isso pode ser ser explicado:

*1. ****Gradientes mais precisos:**** Quando o tamanho do batch é menor, o gradiente calculado em cada etapa do treinamento é uma estimativa menos precisa do gradiente verdadeiro. Isso pode introduzir mais ruído no processo de treinamento, o que pode, paradoxalmente, ajudar o modelo a evitar mínimos locais e encontrar melhores soluções. Em contraste, quando o tamanho do batch é maior, o gradiente é uma estimativa mais precisa, mas isso pode fazer com que o modelo fique preso em mínimos locais.*

*2. ****Mais atualizações de modelo:**** Quando o tamanho do batch é menor, o modelo é atualizado com mais frequência. Por exemplo, se você tem 1000 exemplos de treinamento, um tamanho de batch de 1 resultará em 1000 atualizações de modelo por época, enquanto um tamanho de batch de 100 resultará em apenas 10 atualizações de modelo por época. Mais atualizações de modelo podem permitir que o modelo aprenda mais a partir dos dados.*

*3. ****Regularização implícita:**** O uso de tamanhos de batch menores também pode ter um efeito de regularização, ajudando a prevenir o overfitting. Isso ocorre porque o ruído introduzido pela estimativa do gradiente com menos exemplos pode ajudar a evitar que o modelo se ajuste demais aos dados de treinamento.*

No entanto, vale a pena notar que embora tamanhos de batch menores possam às vezes resultar em uma acurácia de teste maior, eles também podem tornar o treinamento mais lento (gráfico 2), porque menos exemplos são processados simultaneamente. Além disso, tamanhos de batch muito pequenos podem resultar em estimativas de gradiente muito ruidosas, o que pode tornar o treinamento instável. Portanto, a escolha do tamanho do batch é um compromisso e pode requerer alguma experimentação para encontrar o melhor valor para um determinado problema.

IV - Modelo MLP

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

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

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

Exercícios para experimentar o modelo

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

```
logit = model( input)
```

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

```
train_loader = DataLoader(train_data,
                          batch_size= 8,
                          shuffle=True,
                          num_workers=0,
                          pin_memory=True)
# Model instantiation
model = OneHotMLP(vocab_size)
model = model.to(device)


# Obtenha o primeiro batch do train_loader
inputs, targets = next(iter(train_loader))

# Mova os inputs e targets para o dispositivo onde o modelo está
inputs = inputs.to(device)
targets = targets.to(device).view(-1)

# Passe os inputs pelo modelo
# Isso retorna os logits, que são os valores brutos de saída do modelo.
# Como o modelo ainda não foi treinado, esses valores são inicialmente aleatórios.
logits = model(inputs)

# Aplique a função sigmoid aos logits
# A função sigmoid mapeia qualquer número real para o intervalo (0, 1),
# então as probabilidades resultantes serão valores entre 0 e 1.
# Essas probabilidades representam a saída do modelo: a probabilidade prevista
# de que cada exemplo de entrada pertença à classe positiva.
probabilities = torch.sigmoid(logits).view(-1)

predicted = torch.round(probabilities).view(-1)
total = targets.size(0)

correct = (predicted == targets)
total_correct = correct.sum().item()
accuracy = 100 * total_correct / total

print(f'logits {logits}')
print(f'probabilities {probabilities}')
print(f'predicted {predicted}')
print(f'targets {targets}')
print(f'correct {correct}')
print(f'correct {total_correct} of {total}')
print(f'accuracy {accuracy}')


OUTPUT

logits tensor([[0.0377],
```

[0.0645],
[0.0222],
[0.0132],
[0.0496],
[0.0317],
[0.0464],
[0.0325]], device='cuda:0', grad_fn=<AddmmBackward0>)
probabilities tensor([0.5094, 0.5161, 0.5056, 0.5033, 0.5124, 0.5079, 0.5116, 0.5081],
device='cuda:0', grad_fn=<ViewBackward0>)
predicted tensor([1., 1., 1., 1., 1., 1., 1., 1.], device='cuda:0',
grad_fn=<ViewBackward0>)
targets tensor([0, 1, 1, 0, 1, 1, 0, 0], device='cuda:0')
correct tensor([False, True, True, False, True, True, False, False],
device='cuda:0')
correct 4 of 8
accuracy 50.0

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

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

```
OneHotMLP(
  (fc1): Linear(in_features=20001, out_features=200, bias=True)
  (fc2): Linear(in_features=200, out_features=1, bias=True)
  (relu): ReLU()
)
```

Os pesos da primeira camada podem ser visualizados com:

```
model.fc1.weight
```

e o elemento constante (*bias*) pode ser visualizado com:

```
model.fc1.bias
```

Calcule o número de parâmetros do modelo, preenchendo a seguinte tabela (utilize `shape` para verificar a estrutura de cada parâmetro do modelo):

Resposta (original no caderno notebook):

layer					
	weight	total	bias	total	total
fc1	[200, 20001]	4000200	[200]	200	4000400
fc2	[1, 200]	200	[1]	1	201
total		4000400		201	4000601

V - Treinamento

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

Cálculo da Loss

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

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

Onde:

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

Muitas vezes chamamos y_i de target e $\hat{y}_i$ de prob.

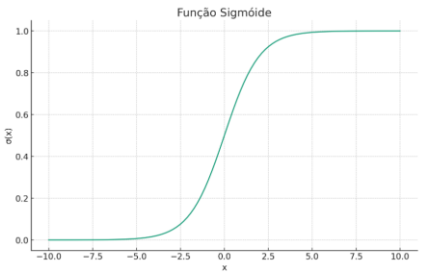
Quando a Loss é zero, significa que o modelo está predizendo tanto as amostras positivas como as amostras negativas com probabilidade de 100%. O objetivo é otimizar o modelo para conseguir minimizar a *Loss* ao máximo.

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

```
output = model( input)
```

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

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



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

```
prob = torch.sigmoid(output)
```

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

Exercícios:

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.

Resposta (original no caderno notebook)

R.: A função de perda de entropia cruzada binária, que é comumente usada para problemas de classificação binária, é definida como:

```
BCELoss = -[y * log(p) + (1 - y) * log(1 - p)]
```

onde `y` é o rótulo verdadeiro (0 ou 1) e `p` é a probabilidade prevista para a classe 1.

Quando o modelo é apenas inicializado e não treinado, e se assumirmos que ele está prevendo uma probabilidade de 0,5 para ambas as classes (o que seria o caso se o modelo estivesse prevendo aleatoriamente), então a função de perda de entropia cruzada binária se simplifica para:

```
BCELoss = -[y * log(0.5) + (1 - y) * log(0.5)]
        = -log(0.5)
        = log(2)
```

O logaritmo natural de 2 é aproximadamente 0.6931. Portanto, o valor teórico da perda quando o modelo não está treinado e está prevendo uma probabilidade de 0,5 para ambas as classes é aproximadamente 0.6931.

Obs.:note que isso é apenas uma aproximação teórica. Na prática, um modelo recém-inicializado pode não prever exatamente 0,5 para todas as entradas, dependendo de como seus pesos são inicializados. Além disso, a perda real também dependerá da distribuição dos rótulos verdadeiros `y` no conjunto de dados.

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.

Valor confere com o teórico

```
# Suponha que você já tenha definido train_data, vocab_size e OneHotMLP

train_loader = DataLoader(train_data,
                          batch_size=128,
                          shuffle=True,
                          num_workers=0,
                          pin_memory=True)

# Instanciação do modelo
model = OneHotMLP(vocab_size)
model = model.to(device)

# Obtenha o primeiro batch do train_loader
inputs, targets = next(iter(train_loader))

# Mova os inputs e targets para o dispositivo onde o modelo está
inputs = inputs.to(device)
targets = targets.to(device).view(-1)
```

```
# Passe os inputs pelo modelo
# Isso retorna os logits, que são os valores brutos de saída do modelo.
# Como o modelo ainda não foi treinado, esses valores são inicialmente aleatórios.
logits = model(inputs)

# Aplique a função sigmoid aos logits
# A função sigmoid mapeia qualquer número real para o intervalo (0, 1),
# então as probabilidades resultantes serão valores entre 0 e 1.
# Essas probabilidades representam a saída do modelo: a probabilidade prevista
# de que cada exemplo de entrada pertença à classe positiva.
probabilities = torch.sigmoid(logits).view(-1)

# Calcule a entropia cruzada manualmente
# A fórmula é: - (y * log(p) + (1 - y) * log(1 - p))
# onde y é a classe real (0 ou 1) e p é a probabilidade prevista
loss = -(targets * torch.log(probabilities) + (1 - targets) * torch.log(1 - probabilities)).mean()

print(f'loss {loss.item()}')

OUTPUT
loss 0.6965450048446655
```

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

```
Sim. Diferenças nas casas decimais provavelmente devido a arredondamentos e representação
numérica.
train_loader = DataLoader(train_data,
                          batch_size= 128,
                          shuffle=True,
                          num_workers=0,
                          pin_memory=True)
# Model instantiation
model = OneHotMLP(vocab_size)
model = model.to(device)
criterion = nn.BCELoss()

# Obtenha o primeiro batch do train_loader
inputs, targets = next(iter(train_loader))

# Mova os inputs e targets para o dispositivo onde o modelo está
inputs = inputs.to(device)
targets = targets.to(device).view(-1)
```

```
# Passe os inputs pelo modelo
# Isso retorna os logits, que são os valores brutos de saída do modelo.
# Como o modelo ainda não foi treinado, esses valores são inicialmente
aleatórios.
logits = model(inputs)

# Aplique a função sigmoid aos logits
# A função sigmoid mapeia qualquer número real para o intervalo (0, 1),
# então as probabilidades resultantes serão valores entre 0 e 1.
# Essas probabilidades representam a saída do modelo: a probabilidade
prevista
# de que cada exemplo de entrada pertença à classe positiva.
probabilities = torch.sigmoid(logits).view(-1)

predicted = torch.round(probabilities).view(-1)
total = targets.size(0)

correct = (predicted == targets)
total_correct = correct.sum().item()
accuracy = 100 * total_correct / total

# Apply the loss function to the probabilities, not the logits
loss = criterion(probabilities, targets.float())

print(f'loss {loss.item()}')

OUTPUT
loss 0.6915096044540405
```

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

```
train_loader = DataLoader(train_data,
                batch_size= 128,
                shuffle=True,
                num_workers=0,
                pin_memory=True)
# Model instantiation
model = OneHotMLP(vocab_size)
model = model.to(device)
criterion = nn.BCEWithLogitsLoss()

# Obtenha o primeiro batch do train_loader
inputs, targets = next(iter(train_loader))

# Mova os inputs e targets para o dispositivo onde o modelo está
inputs = inputs.to(device)
targets = targets.to(device).view(-1)

# Passe os inputs pelo modelo
# Isso retorna os logits, que são os valores brutos de saída do modelo.
# Como o modelo ainda não foi treinado, esses valores são inicialmente aleatórios.
logits = model(inputs)
```

```
# Aplique a função sigmoid aos logits
# A função sigmoid mapeia qualquer número real para o intervalo (0, 1),
# então as probabilidades resultantes serão valores entre 0 e 1.
# Essas probabilidades representam a saída do modelo: a probabilidade prevista
# de que cada exemplo de entrada pertença à classe positiva.
probabilities = torch.sigmoid(logits).view(-1)

predicted = torch.round(probabilities).view(-1)
total = targets.size(0)

correct = (predicted == targets)
total_correct = correct.sum().item()
accuracy = 100 * total_correct / total

loss = criterion(logits.view(-1), targets.float())

print(f'loss {loss.item()}')

OUTPUT
loss 0.6923476457595825
```

Minimização da Loss pelo gradiente descendente

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

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

```
def validar_correcao_loss_esperada_xentropy(model, criterion, train_loader):

    # Forward pass com o modelo inicializado
    # Obtenha o primeiro batch do train_loader
    inputs, targets = next(iter(train_loader))

    # Mova os inputs e targets para o dispositivo onde o modelo está
    inputs = inputs.to(device)
    targets = targets.to(device).view(-1)
```

```
    initial_outputs = model(inputs)  # Inputs devem ser do DataLoader de treinamento
    loss = criterion(initial_outputs.view(-1), targets.float())

    # Valor esperado da Loss
    valor_esperado = 0.6931

    diff = abs(loss.item()-valor_esperado)
    # Verifique se a Loss inicial está próxima do valor esperado
    if diff < 0.01:  # Considerando igualdade até 2 casas decimais
        print(f"A Loss inicial de cross entropy {loss} está correta! Diferença de {diff:.4f}
(arredondamentos)")
        return True
    else:
        print(f"A Loss inicial de cross entropy não corresponde ao valor esperado. Valor atual:
{loss.item():.4f}. Valor esperado para loss: {valor_esperado:.4f}. Diferença de {diff:.4f}")
        return False


def train_model(model, criterion, optimizer, num_epochs, train_loader, test_loader,
verbose:bool=True):

    if validar_correcao_loss_esperada_xentropy(model, criterion, train_loader):
        # Training loop
        for epoch in range(num_epochs):

            (...)
```

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

Realmente não corresponde à inicial, pois os parâmetros do modelo fora atualizados e ela diminuiu (houve aprendizado).
Para recomeçar em modo aleatório é necessário reiniciar os parâmetros, recriando-se o modelo com o comando model = OneHotMLP(vocab_size).

Modificando a rede para gerar dois logitos no lugar de 1

Existe uma forma alternativa de implementar um modelo binário utilizando 2 logitos, um para dar a probabilidade da classe positiva e outro para a classe negativa. Para isso, modifique a camada de saída da rede para gerar 2 logitos no lugar de apenas 1 logito. Agora, para converter os logitos em probabilidade, é necessário utilizar a função Softmax, que é dada pela equação:

$$\text{Softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}},$$

onde:

- $\text{Softmax}(z)_i$ é o i-ésimo elemento do vetor resultante após aplicar a função Softmax ao vetor z.
- e é a constante de Euler (aproximadamente 2.71828).
- z_i é o i-ésimo elemento do vetor de entrada z.
- $\sum_{j=1}^{K} e^{z_j}$ é a soma de todas as exponenciais dos elementos do vetor de entrada z.

A função Softmax é usada para prever mais de 2 classes, mas também pode ser usada para o nosso caso de 2 classes. Quando existem C classes, utiliza-se C logitos na saída da rede neural. O Softmax converte estes C logitos em C probabilidades de modo que a soma destas probabilidades é sempre igual a 1, não importando os valores dos logitos que podem assumir quaisquer valores, negativos ou positivos.

O valor da Loss para o caso de C classes e N amostras no batch, é dado por:

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

Exercícios:

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

```
class OneHotMLPClasses(nn.Module):
    def __init__(self, vocab_size, num_classes):
        super(OneHotMLPClasses, self).__init__()
        self.fc1 = nn.Linear(vocab_size + 1, 200)
        self.fc2 = nn.Linear(200, num_classes)  # Duas saídas para as classes positiva e negativa
        self.relu = nn.ReLU()

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

# Suponha que você já tenha definido train_data, vocab_size e OneHotMLP

train_loader = DataLoader(train_data,
                          batch_size=128,
                          shuffle=True,
                          num_workers=0,
                          pin_memory=True)

# Instanciação do modelo
model = OneHotMLPClasses(vocab_size,2)
model = model.to(device)
```

```
# Obtenha o primeiro batch do train_loader
inputs, targets = next(iter(train_loader))

# Mova os inputs e targets para o dispositivo onde o modelo está
inputs = inputs.to(device)
targets = targets.to(device)

# Passe os inputs pelo modelo
# Isso retorna os logits, que são os valores brutos de saída do modelo.
# Como o modelo ainda não foi treinado, esses valores são inicialmente aleatórios.
logits = model(inputs)

# Calculate the softmax manually
exp_logits = torch.exp(logits)
sum_exp_logits = torch.sum(exp_logits, dim=1, keepdim=True)
probabilities = exp_logits / sum_exp_logits

#print(f'targets {targets}')
#print(f'logits {logits}')
#print(f'probabilities {probabilities}')

# Calcule a entropia cruzada manualmente
# A fórmula é: - (y * log(p) + (1 - y) * log(1 - p))
# onde y é a classe real (0 ou 1) e p é a probabilidade prevista
# loss = -(targets * torch.log(probabilities) + (1 - targets) * torch.log(1 - probabilities)).mean()


# Convert targets to one-hot encoding
targets_one_hot = torch.nn.functional.one_hot(targets)

# Calculate the cross-entropy loss manually
# torch.nn.functional.one_hot(targets) converte targets para a codificação one-hot.
# A fórmula -(targets_one_hot * torch.log(probabilities)).sum() / targets.size(0)
# calcula a entropia cruzada multiclasse, que é a soma negativa das probabilidades
# logarítmicas previstas para as classes verdadeiras, dividida pelo número de amostras.

# note que isso pressupõe que targets é um tensor de rótulos de classe e que probabilities
# é um tensor de probabilidades previstas que soma 1 ao longo da dimensão das classes.
loss = -(targets_one_hot * torch.log(probabilities)).sum() / targets.size(0)

print(f'loss {loss.item()}')

OUTPUT
loss 0.6964499950408936
```

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)

```
# Suponha que você já tenha definido train_data, vocab_size e OneHotMLP
train_loader = DataLoader(train_data,
                          batch_size=128,
```

```
                shuffle=True,
                num_workers=0,
                pin_memory=True)

# Instanciação do modelo
model = OneHotMLPClasses(vocab_size,2)
model = model.to(device)

# Obtenha o primeiro batch do train_loader
inputs, targets = next(iter(train_loader))

# Mova os inputs e targets para o dispositivo onde o modelo está
inputs = inputs.to(device)
targets = targets.to(device)

# Passe os inputs pelo modelo
# Isso retorna os logits, que são os valores brutos de saída do modelo.
# Como o modelo ainda não foi treinado, esses valores são inicialmente aleatórios.
logits = model(inputs)

# A função softmax mapeia probabilidades
probabilities = torch.softmax(logits, dim=1)

#print(f'targets {targets}')
#print(f'logits {logits}')
#print(f'probabilities {probabilities}')

# Calcule a entropia cruzada manualmente
# A fórmula é: - (y * log(p) + (1 - y) * log(1 - p))
# onde y é a classe real (0 ou 1) e p é a probabilidade prevista
# loss = -(targets * torch.log(probabilities) + (1 - targets) * torch.log(1 - probabilities)).mean()


# Convert targets to one-hot encoding
targets_one_hot = torch.nn.functional.one_hot(targets)

# Calculate the cross-entropy loss manually
# torch.nn.functional.one_hot(targets) converte targets para a codificação one-hot.
# A fórmula -(targets_one_hot * torch.log(probabilities)).sum() / targets.size(0)
# calcula a entropia cruzada multiclasse, que é a soma negativa das probabilidades
# logarítmicas previstas para as classes verdadeiras, dividida pelo número de amostras.

# note que isso pressupõe que targets é um tensor de rótulos de classe e que probabilities
# é um tensor de probabilidades previstas que soma 1 ao longo da dimensão das classes.
loss = -(targets_one_hot * torch.log(probabilities)).sum() / targets.size(0)

print(f'loss {loss.item()}')
OUTPUT
loss 0.6906419992446899
```

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

Sim. Diferenças nas casas decimais provavelmente devido a arredondamentos e representação numérica.

```
# Suponha que você já tenha definido train_data, vocab_size e OneHotMLP

train_loader = DataLoader(train_data,
                          batch_size=128,
                          shuffle=True,
                          num_workers=0,
                          pin_memory=True)

# Instanciação do modelo
model = OneHotMLPClasses(vocab_size,2)
model = model.to(device)

# Obtenha o primeiro batch do train_loader
inputs, targets = next(iter(train_loader))

# Mova os inputs e targets para o dispositivo onde o modelo está
inputs = inputs.to(device)
targets = targets.to(device)

# Passe os inputs pelo modelo
# Isso retorna os logits, que são os valores brutos de saída do modelo.
# Como o modelo ainda não foi treinado, esses valores são inicialmente aleatórios.
logits = model(inputs)

# A função softmax mapeia probabilidades
probabilities = torch.softmax(logits, dim=1)

# Instantiate the loss function
criterion = nn.CrossEntropyLoss()

# Calculate the loss
loss = criterion(logits, targets)

print(f'loss {loss.item()}')

OUTPUT
loss 0.6907879710197449
```

V.3.c) 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.
Código alterado

```
def evaluate(model, test_loader):
  ## evaluation
  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 outputs.shape[1] == 1: # gera 1 logito
                predicted = torch.round(torch.sigmoid(outputs)).view(-1)
            else: # gera mais de um logito, um para cada classe
                predicted = torch.argmax(outputs, dim=1).view(-1)
            total += labels.size(0)
            correct += (predicted == labels.view(-1)).sum().item()

        test_accuracy = 100 * correct / total

        return test_accuracy
def validar_correcao_loss_esperada_xentropy(model, criterion, train_loader):

    # Forward pass com o modelo inicializado
    # Obtenha o primeiro batch do train_loader
    inputs, targets = next(iter(train_loader))

    # Mova os inputs e targets para o dispositivo onde o modelo está
    inputs = inputs.to(device)
    targets = targets.to(device).view(-1)

    initial_outputs = model(inputs)  # Inputs devem ser do DataLoader de treinamento

    if initial_outputs.shape[1] == 1: # gera 1 logito
        loss = criterion(initial_outputs.view(-1), targets.view(-1).float())
    else: # gera mais de um logito, um para cada classe
        loss = criterion(nn.functional.softmax(initial_outputs, dim=1), targets)

    # Valor esperado da Loss
    valor_esperado = 0.6931

    diff = abs(loss.item()-valor_esperado)
    # Verifique se a Loss inicial está próxima do valor esperado
    if diff < 0.01:  # Considerando igualdade até 2 casas decimais
        print(f"A Loss inicial de cross entropy {loss} está correta! Diferença de {diff:.4f}
(arredondamentos)")
        return True
    else:
        print(f"A Loss inicial de cross entropy não corresponde ao valor esperado. Valor atual:
{loss.item():.4f}. Valor esperado para loss: {valor_esperado:.4f}. Diferença de {diff:.4f}")
        return False

def train_model(model, criterion, optimizer, num_epochs, train_loader, test_loader,
verbose:bool=True):
```

```
if validar_correcao_loss_esperada_xentropy(model, criterion, train_loader):
    # Training loop
    for epoch in range(num_epochs):
        epoch_start_time = time.time()  # Start time of the epoch
        model.train()
        for cnt_batch, (inputs, labels) in enumerate(train_loader):
            # print(f"Batch {cnt+1}")
            # mover para gpu
            inputs = inputs.to(device)
            labels = labels.to(device)
            # Forward pass
            outputs = model(inputs)
            if verbose and cnt_batch==0:
                # Verifique as dimensões dos tensores
                print("Dimensões dos outputs:", outputs.shape)
                print("Dimensões dos labels:", labels.shape)

            if outputs.shape[1] == 1: # gera 1 logito
                loss = criterion(outputs.view(-1), labels.view(-1).float())
            else:  # gera mais de um logito, um para cada classe
                loss = criterion(nn.functional.softmax(outputs, dim=1), labels)

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

        epoch_duration = time.time() - epoch_start_time  # Duration of epoch

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

    # Evaluate the model and record the test accuracy
    test_accuracy = evaluate(model, test_loader)

    return test_accuracy
```

VI - Avaliação

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

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

```
print(f'O total de amostras avaliadas em teste é: {len(test_loader.dataset)} em
{len(test_loader)} batches')
OUTPUT
O total de amostras avaliadas em teste é: 25000 em 3125 batches
```

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

*1. ****model.eval():**** Quando você chama `model.eval()`, você está essencialmente definindo o modelo (nn.Module) para o modo de avaliação. Isso tem efeitos em certas camadas do seu modelo, como Dropout e BatchNorm, que se comportam de maneira diferente durante o treinamento e durante a avaliação. Por exemplo, durante o treinamento, a camada Dropout irá aleatoriamente zerar algumas das entradas, mas durante a avaliação (ou seja, quando o modelo está em modo `.eval()`), a camada Dropout não alterará suas entradas. Da mesma forma, a camada BatchNorm usará estatísticas de execução durante o treinamento, mas usará estatísticas acumuladas durante a avaliação. Embora o modelo possa não possuir essas camadas, há sempre a possibilidade de ele ser evoluído. Outra motivação é a legibilidade do código: deixa claro para qualquer pessoa que esteja lendo o seu código que essa parte do código está fazendo a avaliação do modelo, não o treinamento. Tem também a compatibilidade com outras bibliotecas de aprendizado profundo ou funções do PyTorch que podem ser afetadas com essa informação do modelo.*

*2. ****with torch.no_grad():**** Em PyTorch, cada operação em tensores que têm `requires_grad=True` irá criar um histórico de computação que permite calcular gradientes usando backpropagation. No entanto, durante a avaliação do modelo, você geralmente não precisa de gradientes, porque você não está atualizando os pesos do modelo. O uso de `with torch.no_grad():` desativa a criação desse histórico de computação, o que pode reduzir o uso de memória e acelerar os cálculos. Isso é especialmente útil durante a avaliação do modelo, quando você normalmente só precisa passar os dados através do modelo e calcular a perda ou as métricas de avaliação, sem precisar atualizar os pesos do modelo.*

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, existe uma maneira mais simples de calcular a classe predita sem a necessidade de usar a função torch.sigmoid. A função torch.sigmoid é usada para mapear os valores de saída do modelo para o intervalo entre 0 e 1, que pode ser interpretado como a probabilidade da classe positiva. No entanto, se você está apenas interessado na classe predita e não na probabilidade, você pode simplesmente usar a função torch.round diretamente nos valores de saída do modelo.

Aqui está como você pode fazer isso:

```
predicted = torch.round(outputs).view(-1)
```

Neste caso, os valores de saída do modelo que são maiores que 0 serão arredondados para 1 (classe positiva) e os valores que são menores ou iguais a 0 serão arredondados para 0 (classe negativa).

No entanto, é importante notar que esta abordagem só é válida se você estiver usando uma função de perda que já inclua a função sigmoid, como a Binary Cross-Entropy with Logits Loss (BCEWithLogitsLoss) no PyTorch. Se você estiver usando uma função de perda que não inclui a função sigmoid, como a Binary Cross-Entropy Loss (BCELoss), você ainda precisará aplicar a função sigmoid aos valores de saída do modelo antes de arredondá-los.

Perplexidade como métrica de avaliação

Em teoria da informação, a perplexidade (PPL) é dada por

$$PPL = e^{CE}$$

onde CE é a Cross Entropy, que utilizamos na Loss do treinamento. A base *e* utilizada para a exponenciação deve ser compatível com a base utilizado no logaritmo da cross entropia. Como utilizamos logaritmo natural para a entropia cruzada, devemos aqui usar o *e*. Se a entropia cruzada usasse a base 2, a perplexidade seria 2 elevado à entropia cruzada.

Eu, particularmente gosto de usar a perplexidade em vez de usar a entropia cruzada pelo motivo que ficará explícito nos exercícios a seguir:

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?

O Valor é 2

np.log(2)
OUTPUT
0.6931471805599453

*print(f'O valor é {round(math.e**np.log(2),3)}')*
OUTPUT
O valor é 2.0

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

Resposta (original no caderno notebook)

R.: Seria N.

A perplexidade é uma medida comumente usada para avaliar modelos de linguagem. Ela é definida como a potência inversa da probabilidade média do modelo de prever a classe correta. Em outras palavras, a perplexidade de um modelo é o número de escolhas igualmente prováveis que ele efetivamente tem ao fazer uma previsão.

Se um modelo está fazendo previsões aleatórias para um problema de classificação com N classes, então a probabilidade de prever a classe correta para qualquer amostra é 1/N. Portanto, a perplexidade para o caso aleatório seria N.

Aqui está a justificativa matemática para isso:

A perplexidade (PPL) é definida como a exponencial da entropia cruzada (CE), que é uma medida da diferença entre duas distribuições de probabilidade. A fórmula da perplexidade é:

$$PPL = e^{CE}$$

A entropia cruzada para o caso aleatório é:

$$CE = -\sum(1/N)log(1/N) = logN$$

Portanto, a perplexidade para o caso aleatório é:

$$PPL = e^{logN} = N$$

Isso significa que, para um modelo que faz previsões aleatórias para um problema de classificação com N classes, a perplexidade é N. Em outras palavras, o modelo tem efetivamente N escolhas igualmente prováveis ao fazer uma previsão.

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

Se um modelo de classificação é capaz de prever a classe correta para todas as amostras com 100% de probabilidade, então a perplexidade do modelo é 1.

Aqui está a justificativa matemática.

Se o modelo está prevendo a classe correta com 100% de probabilidade, então a distribuição de probabilidade prevista pelo modelo é a mesma que a distribuição de probabilidade verdadeira. Nesse caso, a entropia cruzada é 0, porque a entropia cruzada é mínima quando as duas distribuições p(x) e q(x) (calculada pelo modelo) são iguais.

A entropia cruzada é definida como:

CE(p,q)} = - ∑ p(x) log (q(x))

*Nesse caso, para a classe correta, temos p(x) = 1 e q(x) = 1. Portanto, o termo correspondente na soma da entropia cruzada é -1 * log(1) = 0, porque o logaritmo de 1 é 0. Para todas as outras classes, temos p(x) = 0. Portanto, os termos correspondentes na soma da entropia cruzada são 0 * log(q(x)) = 0, porque qualquer número multiplicado por 0 é 0. Portanto, a entropia cruzada é a soma de muitos termos 0, o que resulta em 0.*

Portanto, a perplexidade quando o modelo acerta todas as classes com 100% de probabilidade é:

PPL = {CE}^0 = 1

Isso significa que, para um modelo perfeito que acerta todas as classes com 100% de probabilidade, a perplexidade é 1. Em outras palavras, o modelo tem efetivamente uma única escolha ao fazer uma previsão.

Se você respondeu corretamente as 3 questões acima, já é possível entender que a perplexidade é muito mais fácil de entender o seu significado do que o valor da Loss como entropia cruzada.

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

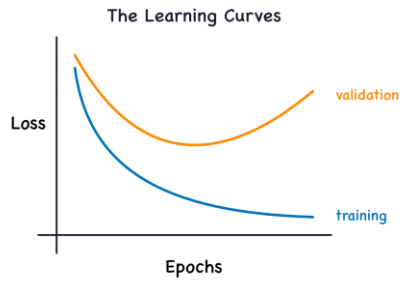
```
def evaluate(model, test_loader, criterion):
  ## evaluation
  model.eval()

  with torch.no_grad():
    correct = 0
    total = len(test_loader.dataset)
    total_loss = 0
    for inputs, labels in test_loader:
      inputs = inputs.to(device)
      labels = labels.to(device)
      outputs = model(inputs)

      if outputs.shape[1] == 1: # gera 1 logito
        predicted = torch.round(torch.sigmoid(outputs)).view(-1)
        loss = criterion(outputs.view(-1), labels.view(-1).float())
```

```
            else: # gera mais de um logito, um para cada classe
                predicted = torch.argmax(outputs, dim=1).view(-1)
                loss = criterion(nn.functional.softmax(outputs, dim=1), labels)
            total_loss += loss.item()
            correct += (predicted == labels.view(-1)).sum().item()

        test_accuracy = 100 * correct / total
        test_loss = total_loss / len(test_loader)
        test_perplexity = torch.exp(torch.tensor(test_loss))
        return test_accuracy, test_perplexity
```

Observando Overfitting

Dizemos que o treinamento está *overfitting* quando treina-se tanto nas mesmas amostras, de modo quase a decorá-lo e quando o modelo fizer a predição em um outro conjunto de amostras, ele não consegue "generalizar" o conhecimento aprendido no treinamento. Uma forma de detectar se o treinamento está entrando no overfitting é calcular, durante o laço de treinamento, tanto a loss de minimização no conjunto de treino, porém ao final de cada época, calcular a loss ou alguma métrica relacionada ao conjunto de teste ou validação.

Último exercício:
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.



A resposta pode ser melhor visualizada no código. Daí replicarei aqui como imagem. Código alterado.

```python
def train_model(model, criterion, optimizer, num_epochs, train_loader, test_loader, verbose:bool=True):

    if validar_correcao_loss_esperada_xentropy(model, criterion, train_loader):
        # Training loop
        total_data = len(train_loader.dataset)
        results = []

        for cnt_epoch, epoch in enumerate(range(num_epochs)):
            epoch_start_time = time.time()  # Start time of the epoch
            model.train()

            total_loss = 0
            correct = 0

            for cnt_batch, (inputs, labels) in enumerate(train_loader):
                # print(f"Batch {cnt+1}")
                # mover para gpu
                inputs = inputs.to(device)
                labels = labels.to(device)
                # Forward pass
                outputs = model(inputs)
                if verbose and cnt_epoch == 0 and cnt_batch==0:
                    # Verifique as dimensões dos tensores
                    print("Dimensões dos outputs:", outputs.shape)
                    print("Dimensões dos labels:", labels.shape)



                if outputs.shape[1] == 1: # gera 1 logito
                    predicted = torch.round(torch.sigmoid(outputs)).view(-1)
                    loss = criterion(outputs.view(-1), labels.view(-1).float())
                else: # gera mais de um logito, um para cada classe
                    predicted = torch.argmax(outputs, dim=1).view(-1)
                    loss = criterion(nn.functional.softmax(outputs, dim=1), labels)

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


                # acumulando por batch
                total_loss += loss.item()
                correct += (predicted == labels.view(-1)).sum().item()


            epoch_train_accuracy = 100 * correct / total_data
            epoch_train_loss = total_loss / len(train_loader)
            epoch_train_perplexity = torch.exp(torch.tensor(epoch_train_loss))


            epoch_duration = time.time() - epoch_start_time  # Duration of epoch

            # Evaluate the model and record the test accuracy
            test_accuracy, test_perplexity = evaluate(model, test_loader, criterion)

            if verbose:
                print(f'Epoch [{epoch+1}/{num_epochs}], '
                      f'train loss: {loss.item():.4f}, '
                      f'train accuracy: {epoch_train_accuracy:.4f}, '
                      f'train perplexity: {epoch_train_perplexity.item():.4f}, '
                      f'test accuracy: {test_accuracy:.4f}, '
                      f'test perplexity: {test_perplexity.item():.4f}, '
                      f'Elapsed Time: {epoch_duration:.2f} sec')



            # Adicionar os resultados da época à lista de resultados
            results.append({
                'epoch': epoch + 1,
                'train_accuracy': epoch_train_accuracy,
                'train_loss': epoch_train_loss,
                'train_perplexity': epoch_train_perplexity.item(),
                'test_accuracy': test_accuracy,
                'test_perplexity': test_perplexity.item()
            })

        return results
```

```python
def print_train_results(train_results,metric_name):
    # Extrair os valores de test_accuracy e train_accuracy
    epochs = [result['epoch'] for result in train_results]

    test_accuracies = [result['test_'+metric_name] for result in train_results]
    train_accuracies = [result['train_'+metric_name] for result in train_results]

    # Criar o gráfico
    plt.figure(figsize=(10, 5))
    plt.plot(epochs, test_accuracies, label='Test '+metric_name)
    plt.plot(epochs, train_accuracies, label='Train '+metric_name)
    plt.xlabel('Epoch')
    plt.ylabel(metric_name.capitalize())
    plt.title(metric_name.capitalize() + ' Test vs Train ')
    plt.legend()
    plt.grid(True)
    plt.show()
```

Cálculos efetuados

```
test_loader = DataLoader(test_data,
                         batch_size= 16,
                         shuffle=False,
                         num_workers=0,
                         pin_memory=True)
print(f"test_loader tem {len(test_loader)} batches")

train_loader = DataLoader(train_data,
                        batch_size= 16,
                        shuffle=True,
                        num_workers=0,
                        pin_memory=True)
print(f"train_loader tem {len(train_loader)} batches")
```

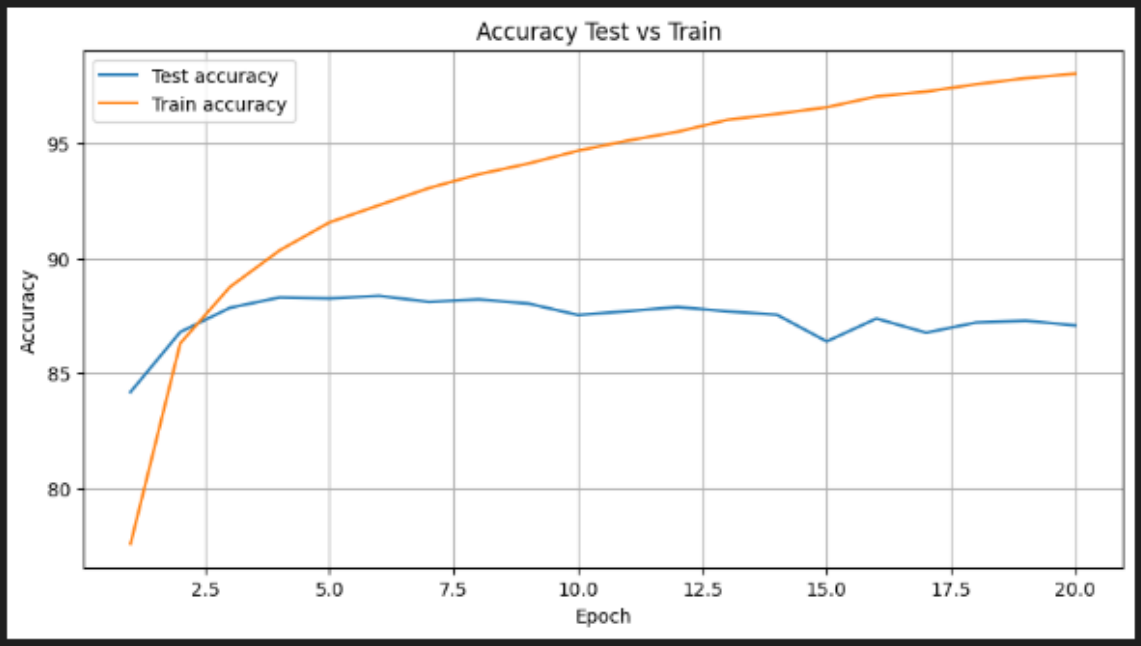
```
test_loader tem 1563 batches
train_loader tem 1563 batches
```

```
# Model instantiation
model = OneHotMLP(vocab_size)
model = model.to(device)
criterion = nn.BCEWithLogitsLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)
train_results = train_model(model, criterion, optimizer, 20, train_loader, test_loader, verbose=True)
```

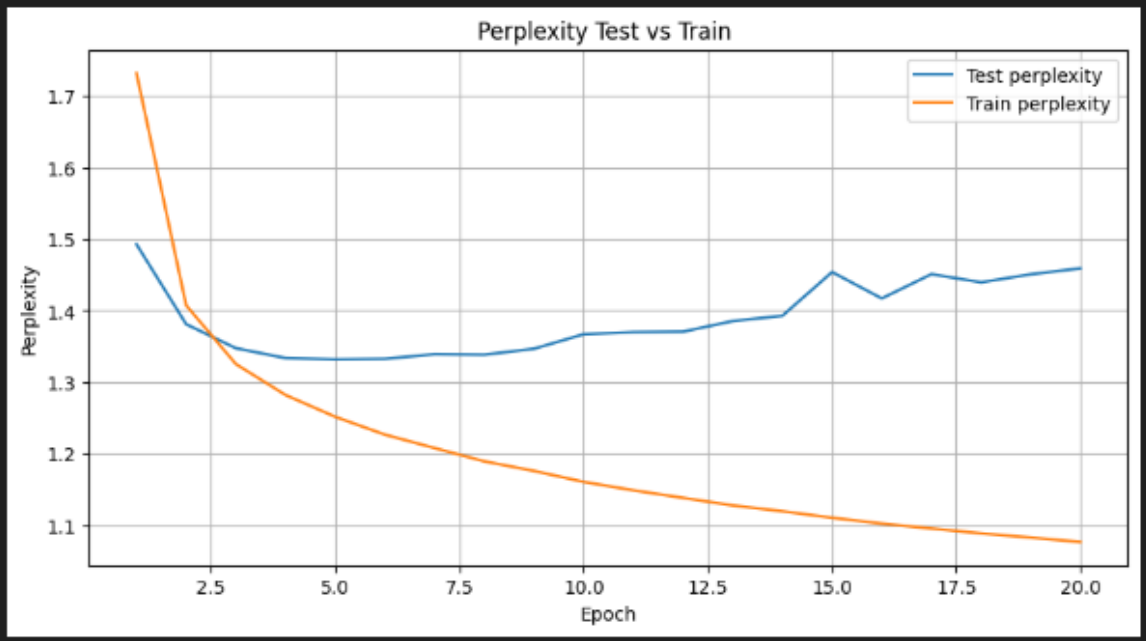
```
A Loss inicial de cross entropy 0.6915335059165955 está correta! Diferença de 0.0016 (arredondamentos)
Dimensões dos outputs: torch.Size([16, 1])
Dimensões dos labels: torch.Size([16])
Epoch [1/20], train loss: 0.3140, train accuracy: 77.5600, train perplexity: 1.7316, test accuracy: 84.1800, test perplexity: 1.4926, Elapsed Time: 3.09 sec
Epoch [2/20], train loss: 0.2274, train accuracy: 86.3000, train perplexity: 1.4073, test accuracy: 86.7920, test perplexity: 1.3809, Elapsed Time: 3.07 sec
Epoch [3/20], train loss: 0.2168, train accuracy: 88.7560, train perplexity: 1.3252, test accuracy: 87.8480, test perplexity: 1.3474, Elapsed Time: 3.02 sec
Epoch [4/20], train loss: 0.2547, train accuracy: 90.3520, train perplexity: 1.2820, test accuracy: 88.3000, test perplexity: 1.3336, Elapsed Time: 2.98 sec
Epoch [5/20], train loss: 0.2032, train accuracy: 91.5680, train perplexity: 1.2515, test accuracy: 88.2560, test perplexity: 1.3318, Elapsed Time: 3.03 sec
Epoch [6/20], train loss: 0.0511, train accuracy: 92.3200, train perplexity: 1.2266, test accuracy: 88.3680, test perplexity: 1.3325, Elapsed Time: 3.02 sec
Epoch [7/20], train loss: 0.1377, train accuracy: 93.0640, train perplexity: 1.2078, test accuracy: 88.1040, test perplexity: 1.3390, Elapsed Time: 3.02 sec
Epoch [8/20], train loss: 0.1034, train accuracy: 93.6640, train perplexity: 1.1893, test accuracy: 88.2120, test perplexity: 1.3383, Elapsed Time: 3.06 sec
Epoch [9/20], train loss: 0.0838, train accuracy: 94.1320, train perplexity: 1.1759, test accuracy: 88.0320, test perplexity: 1.3466, Elapsed Time: 3.09 sec
Epoch [10/20], train loss: 0.0174, train accuracy: 94.6920, train perplexity: 1.1606, test accuracy: 87.5320, test perplexity: 1.3669, Elapsed Time: 3.08 sec
Epoch [11/20], train loss: 0.0737, train accuracy: 95.1320, train perplexity: 1.1488, test accuracy: 87.7000, test perplexity: 1.3698, Elapsed Time: 3.03 sec
Epoch [12/20], train loss: 0.1002, train accuracy: 95.5160, train perplexity: 1.1382, test accuracy: 87.8800, test perplexity: 1.3704, Elapsed Time: 2.99 sec
Epoch [13/20], train loss: 0.0756, train accuracy: 96.0360, train perplexity: 1.1277, test accuracy: 87.6920, test perplexity: 1.3853, Elapsed Time: 3.08 sec
Epoch [14/20], train loss: 0.0048, train accuracy: 96.2960, train perplexity: 1.1195, test accuracy: 87.5440, test perplexity: 1.3927, Elapsed Time: 3.03 sec
Epoch [15/20], train loss: 0.2453, train accuracy: 96.5840, train perplexity: 1.1104, test accuracy: 86.3800, test perplexity: 1.4538, Elapsed Time: 3.12 sec
Epoch [16/20], train loss: 0.0604, train accuracy: 97.0560, train perplexity: 1.1023, test accuracy: 87.3800, test perplexity: 1.4170, Elapsed Time: 3.13 sec
Epoch [17/20], train loss: 0.2016, train accuracy: 97.2680, train perplexity: 1.0954, test accuracy: 86.7600, test perplexity: 1.4509, Elapsed Time: 3.07 sec
Epoch [18/20], train loss: 0.0362, train accuracy: 97.5840, train perplexity: 1.0886, test accuracy: 87.2040, test perplexity: 1.4394, Elapsed Time: 3.08 sec
Epoch [19/20], train loss: 0.0167, train accuracy: 97.8520, train perplexity: 1.0828, test accuracy: 87.2840, test perplexity: 1.4507, Elapsed Time: 2.99 sec
Epoch [20/20], train loss: 0.0672, train accuracy: 98.0480, train perplexity: 1.0766, test accuracy: 87.0800, test perplexity: 1.4590, Elapsed Time: 3.07 sec
```

Gráficos dos resultados

```
print_train_results(train_results,'accuracy')
```




```
print_train_results(train_results,'perplexity')
```

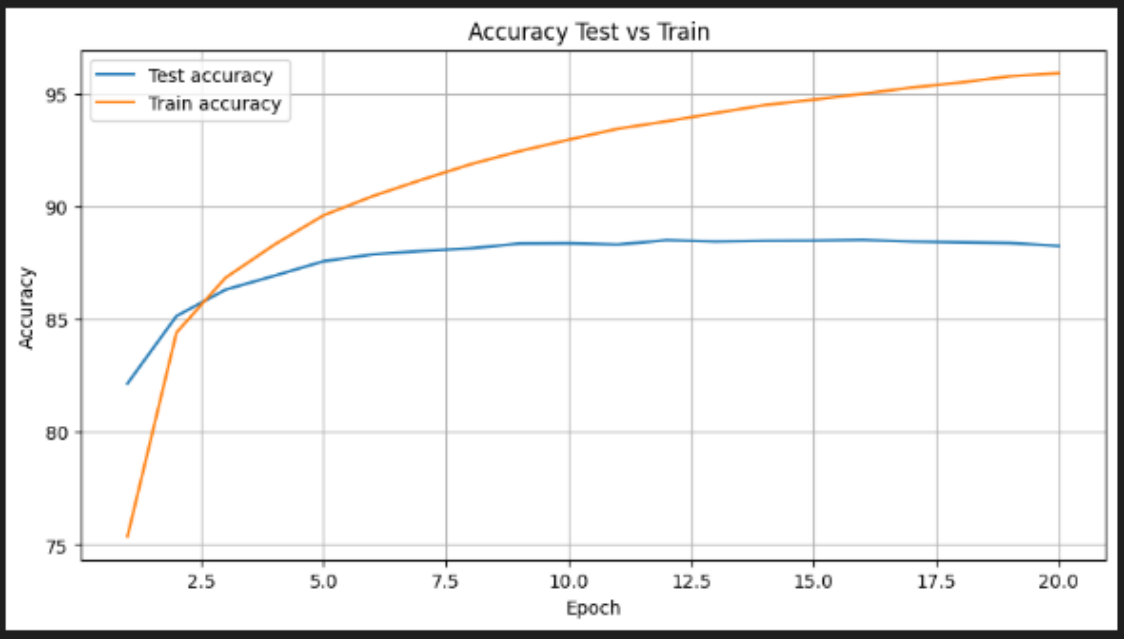


Experimentando o código com o modelo que gera 2 classes (para validação do código e vizualização do overfitting nesse caso também). Com 'verbose=False" para não imprimir os valores durante o treinamento.

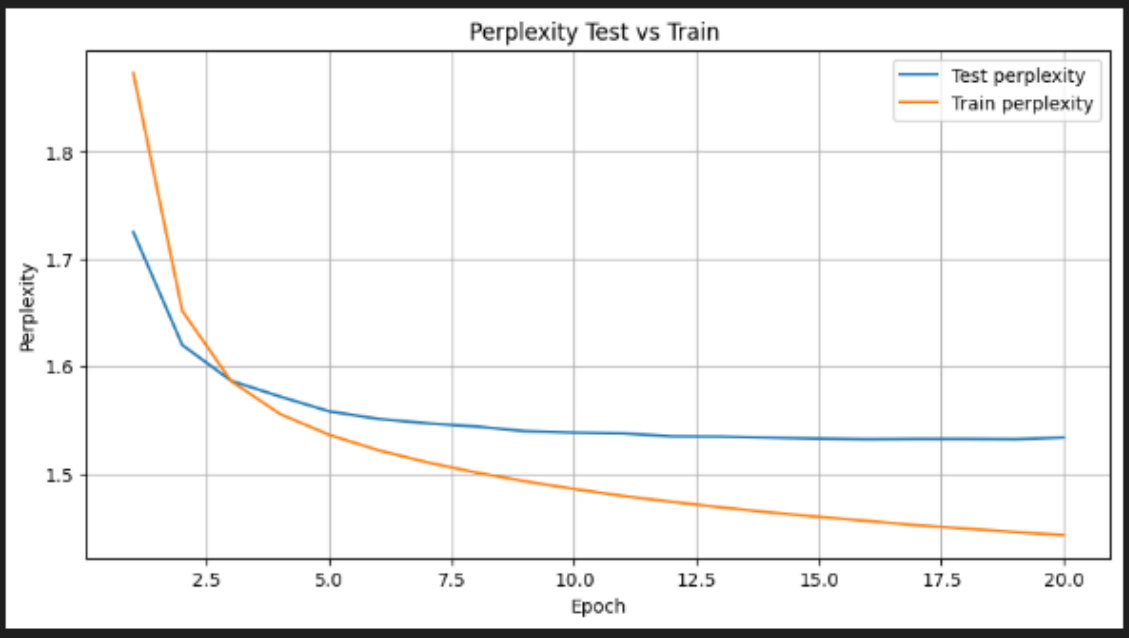
```
model = OneHotMLPClasses(vocab_size,2)
model = model.to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)
train_results = train_model(model, criterion, optimizer, 20, train_loader, test_loader, verbose=False)
```

```
A Loss inicial de cross entropy 0.6912642121315002 está correta! Diferença de 0.0018 (arredondamentos)
```

```
print_train_results(train_results,'accuracy')
```




```
print_train_results(train_results,'perplexity')
```



Visualizando o overfit

Uma forma de validar o modelo é ver se ele está diminuindo a loss para um batch apenas. Se considerarmos como teste todos os dados, o overfit será logo percebido.

```python
def create_dataloader_getting_records(data_loader, num_records):
    """
    Retorna um novo DataLoader que contém apenas os primeiros num_records de data_loader.dataset
    """
    # Cria um subconjunto do conjunto de dados original
    subset = Subset(data_loader.dataset, indices=range(num_records))

    # Cria um novo DataLoader a partir do subconjunto
    new_data_loader = DataLoader(subset, batch_size=data_loader.batch_size, shuffle=False)

    return new_data_loader
```

```python
train_loader_1000 = create_dataloader_getting_records(train_loader,1000)
```

```python
print(f'train_loader_1000 tem {len(train_loader_1000)} batches e {len(train_loader_1000.dataset)} records')
```

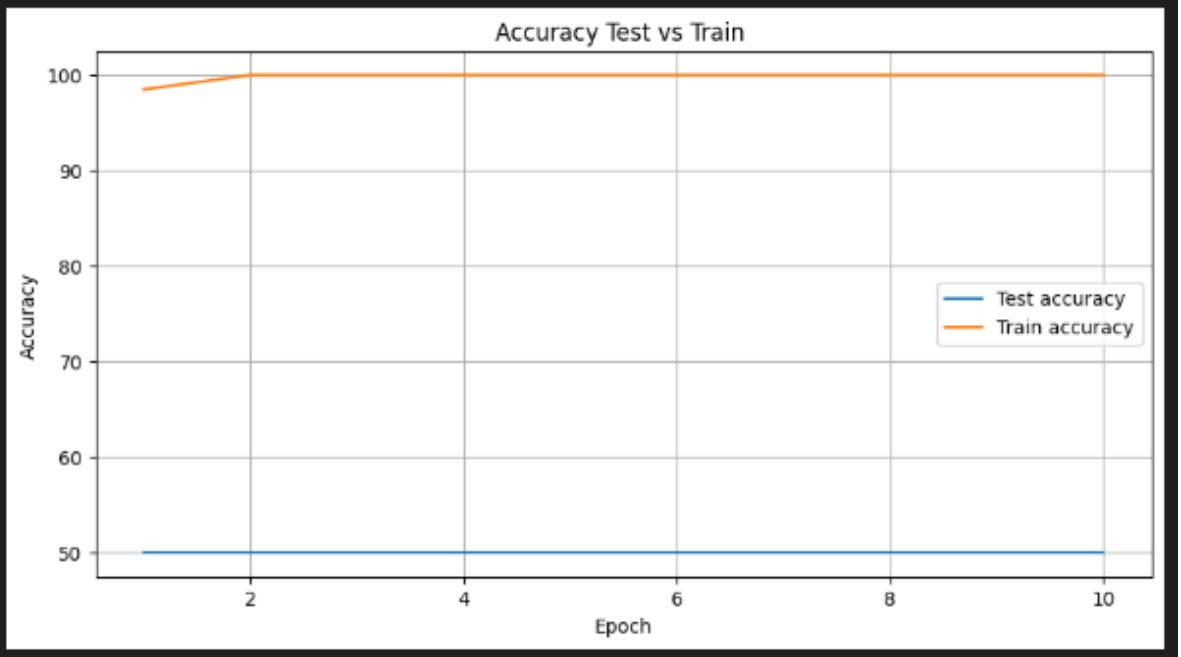
```
train_loader_1000 tem 63 batches e 1000 records
```

```python
model = OneHotMLPClasses(vocab_size,2)
model = model.to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)
train_results = train_model(model, criterion, optimizer, 10, train_loader_1000, test_loader, verbose=True)
```

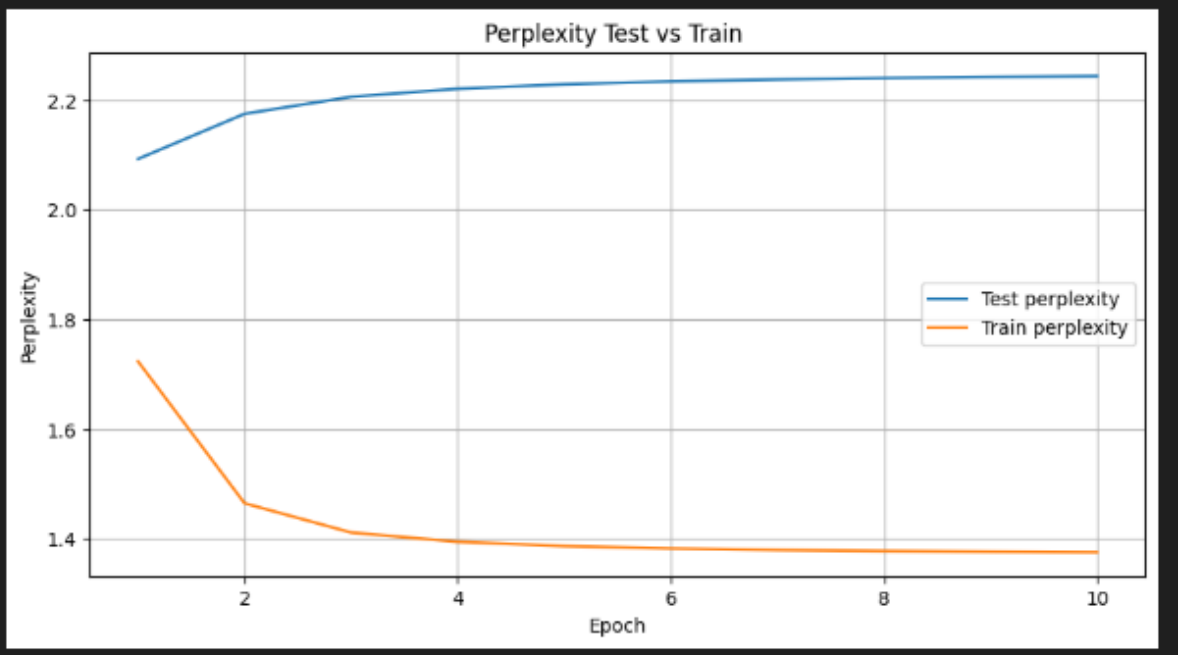
```
A Loss inicial de cross entropy 0.6932174563407898 está correta! Diferença de 0.0001 (arredondamentos)
Dimensões dos outputs: torch.Size([16, 2])
Dimensões dos labels: torch.Size([16])
Epoch [1/10], train loss: 0.4418, train accuracy: 98.5000, train perplexity: 1.7234, test accuracy: 50.0000, test perplexity: 2.0923, Elapsed Time: 0.11 sec
Epoch [2/10], train loss: 0.3635, train accuracy: 100.0000, train perplexity: 1.4644, test accuracy: 50.0000, test perplexity: 2.1747, Elapsed Time: 0.11 sec
Epoch [3/10], train loss: 0.3410, train accuracy: 100.0000, train perplexity: 1.4110, test accuracy: 50.0000, test perplexity: 2.2057, Elapsed Time: 0.11 sec
Epoch [4/10], train loss: 0.3315, train accuracy: 100.0000, train perplexity: 1.3940, test accuracy: 50.0000, test perplexity: 2.2203, Elapsed Time: 0.11 sec
Epoch [5/10], train loss: 0.3265, train accuracy: 100.0000, train perplexity: 1.3863, test accuracy: 50.0000, test perplexity: 2.2286, Elapsed Time: 0.11 sec
Epoch [6/10], train loss: 0.3235, train accuracy: 100.0000, train perplexity: 1.3819, test accuracy: 50.0000, test perplexity: 2.2338, Elapsed Time: 0.11 sec
Epoch [7/10], train loss: 0.3215, train accuracy: 100.0000, train perplexity: 1.3792, test accuracy: 50.0000, test perplexity: 2.2373, Elapsed Time: 0.11 sec
Epoch [8/10], train loss: 0.3201, train accuracy: 100.0000, train perplexity: 1.3773, test accuracy: 50.0000, test perplexity: 2.2399, Elapsed Time: 0.11 sec
Epoch [9/10], train loss: 0.3191, train accuracy: 100.0000, train perplexity: 1.3759, test accuracy: 50.0000, test perplexity: 2.2419, Elapsed Time: 0.11 sec
Epoch [10/10], train loss: 0.3183, train accuracy: 100.0000, train perplexity: 1.3749, test accuracy: 50.0000, test perplexity: 2.2434, Elapsed Time: 0.13 sec
```

Gráficos do overfit

```
print_train_results(train_results,'accuracy')
```




```
print_train_results(train_results,'perplexity')
```



Pelos gráficos, percebe-se que o modelo está aprendendo (métricas melhoram nos dados de treinamento), mas está "overfitando". Ele está como que se especializando nos dados de treinamento. E como os dados de teste são genéricos, ele não alcança bons reultados em teste.

Comentários finais:

Se você conseguiu chegar até aqui, fazendo todos os exercícios, parabéns. Espero que você tenha aprendido vários conceitos importantes de treinamento de redes neurais. O valioso é que tudo o que foi visto aqui é válido tanto para modelos de alguns milhares de parâmetros como visto aqui até os modelos LLM da ordem de bilhões de parâmetros. Dominar técnicas de treinamento de modelos deep learning é possível apenas com muita experiência de programação e conceitos sólidos da teoria, que não é muita. É a teoria de minimização de funções pelo método do gradiente descendente. Ficou faltando entender como o gradiente é calculado, que é algo que o pytorch e outros ambientes similares conseguiram simplificar e deixá-lo quase imperceptível para o programador.
Espero que você tenha aprendido com esses exercícios. Eles são uma amostra do que estudaremos e da forma como estudaremos no curso IA-024.
Existem vários conceitos muito importantes que não tratamos aqui. Conceito de embedding,
Nesse curso não aceitamos alunos ouvintes, apenas alunos comprometidos com os exercícios e o aprendizado colaborativo com os colegas. Assim, a participação de todos será fundamental para o sucesso do curso.

Formas da Entrega

Os exercícios devem ser entregues em arquivo PDF junto com um notebook com a versão final do código, comentada e com um desempenho bem melhor, ainda utilizando a mesma técnica (BagOfWords) e rede neural de 2 camadas e mesmo número de neurônios.

Os links do arquivo PDF e do Google colab notebook deve ser entregues neste formulário:
https://forms.gle/xAGtt55fYpHW23eZA
Entregue a versão final do Google colab, contendo as modificações sugeridas ao longo dos exercícios que você conseguiu fazer.

Avaliação:

Esse é um processo seletivo para escolher alunos especiais para o curso IA-024 onde existem 50 candidatos inscritos. Serão selecionados os alunos que obtiverem as melhores avaliações nas respostas dos exercícios. É esperado que sejam selecionados da ordem de 10 a 15 alunos especiais.