**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, e**m 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

Gráfico, Gráfico de linhas

Descrição gerada automaticamente

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.

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

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)

Tabela

Descrição gerada automaticamente

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)

Texto

Descrição gerada automaticamente

**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)

Tabela

Descrição gerada automaticamente

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.

Gráfico, Gráfico de linhas

Descrição gerada automaticamente

Gráfico, Gráfico de linhas

Descrição gerada automaticamente

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):

Tabela

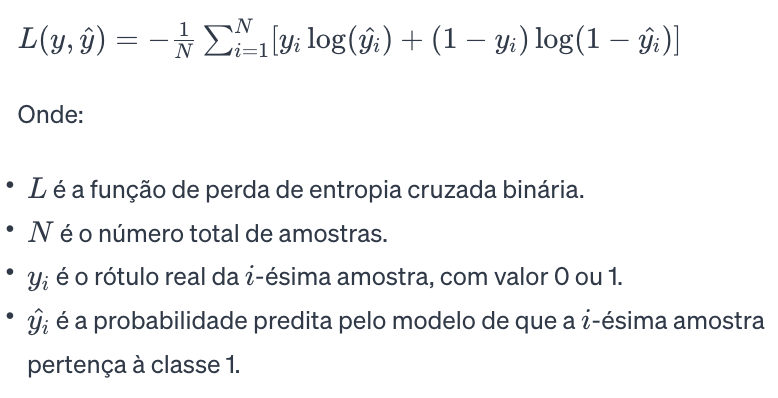
Descrição gerada automaticamente

**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:



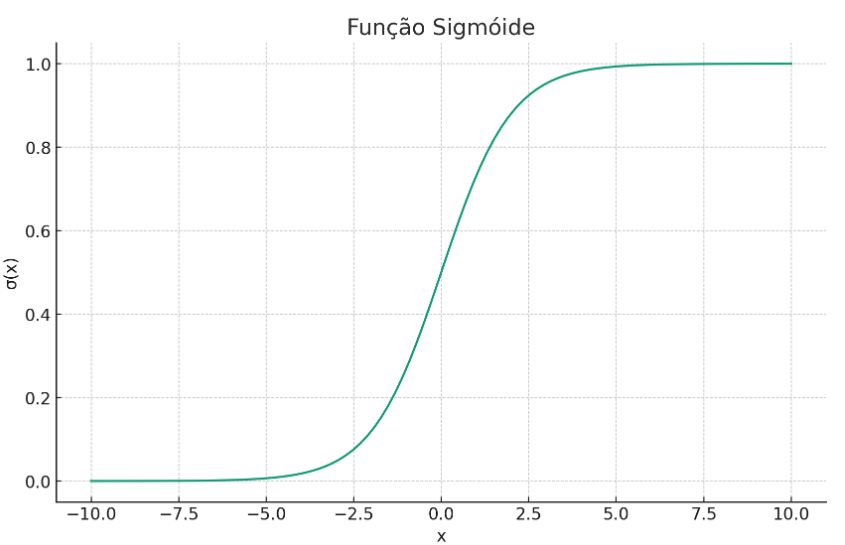
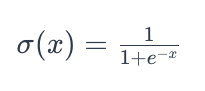
Muitas vezes chamamos de target e de prob.

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:



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)

Texto

Descrição gerada automaticamente

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?

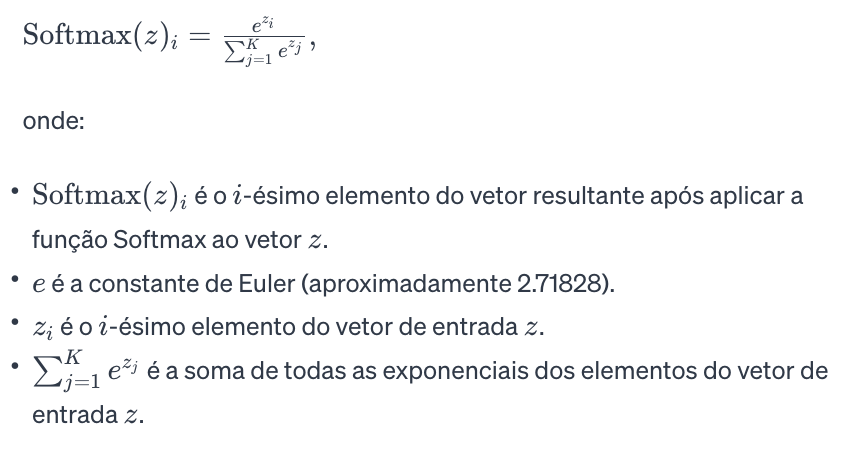
Realmente não corresponde à inicial, pois os parâmetros do modelo fora atualizados e ela diminuiu (houve aprendizado).

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**

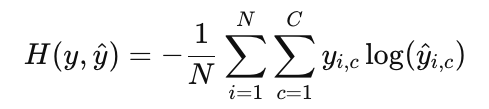
Existe uma forma alternativa de implementar um modelo binário utilizando 2 logitos, um para dar a probabilidade da classe positiva e outro para a classe negativa. Para isso, modifique a camada de saída da rede para gerar 2 logitos no lugar de apenas 1 logito.

Agora, para converter os logitos em probabilidade, é necessário utilizar a função Softmax, que é dada pela equação:



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:



**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:

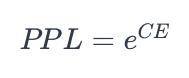
        predicted = torch.round(outputs).view(-1)

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



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)

Texto

Descrição gerada automaticamente

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))

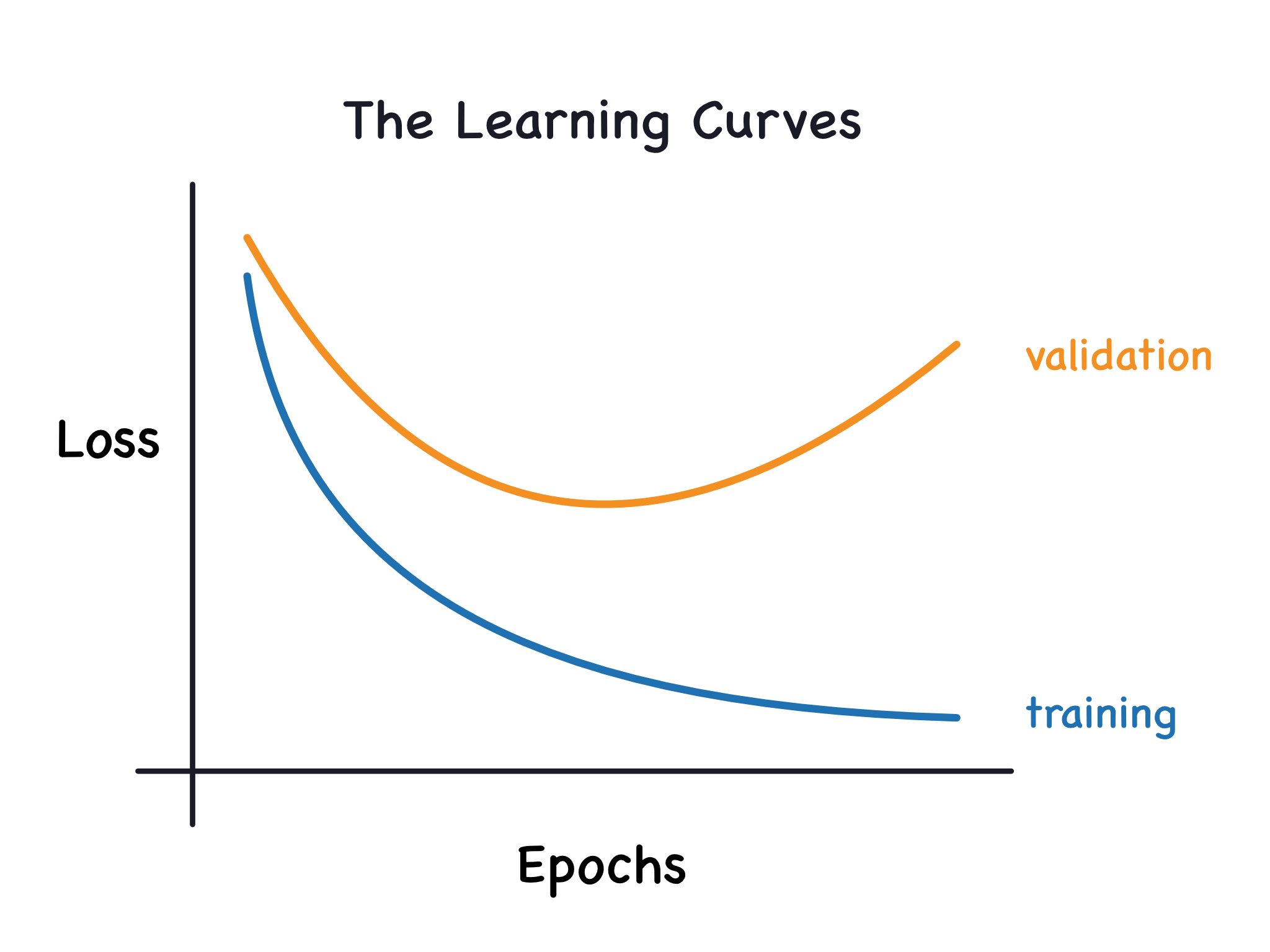
        return test\_accuracy, test\_perplexity

**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.

Texto

Descrição gerada automaticamente

Cálculos efetuados

Texto

Descrição gerada automaticamente

Gráficos dos resultados

Gráfico, Gráfico de linhas

Descrição gerada automaticamente

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.

Interface gráfica do usuário

Descrição gerada automaticamente

**Visualizando** o overfit

Texto

Descrição gerada automaticamente

Gráficos do overfit

Interface gráfica do usuário

Descrição gerada automaticamente com confiança baixa

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.