Exercícios Preliminares IA-024 1S2024 FEEC-UNICAMP

Nome: Caio Petrucci dos Santos Rosa **RA:** 248245

Resolução dos Exercícios

I - Vocabulário e tokenização

Exercício 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.

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

A modificação está contida no Notebook 1.

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

O número de amostras positivas, negativas e totais são, respectivamente, **12500**, **12500** e **25000**.

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

O comprimento médio dos textos (reviews) das amostras, em palavras, é: **233.7872**.

Exercício I.2:

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

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

As cinco palavras mais frequentes do vocabulário são: “the”, “a”, “and”, “of”, “to”. Já as cinco menos frequentes do vocabulário são: “age-old”, “place!”, “Bros”, “tossing”, “nation,”.
O código utilizado para encontrar estas palavras está contido no Notebook 1.

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

Após ordenar as palavras por frequência no texto e selecionar as 2000 (ou qualquer seja o tamanho do vocabulário) mais frequentes nos textos do conjunto de treinamento, foi atribuído uma codificação para cada uma delas: um número, de 1 até 2000, atribuindo o código 1 à palavra mais frequente de todas, o código 2 à segunda palavra mais frequente, e assim em diante para as 2000 palavras.
Assim, para toda palavra fora do vocabulário, foi atribuído o código 0 para aqueles tokens que não estão inclusos no vocabulário. Essa atribuição do código de unknown token é feita na função encode_sentence, a partir do trecho *vocab.get(word, 0)*, dado que a função *get* do *dict* retorna o segundo argumento passado como o valor default (que nesse caso é 0) caso a string *word* não seja um chave do dicionário vocab.

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

O número de unknown tokens encontrados no conjunto de treinamento é: **260617**.
O código utilizado para calcular este número de token não reconhecidos está contido no Notebook 1.

Exercício I.3:

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?

Após limitar os conjuntos de treinamento e de teste em todo o notebook para apenas as 200 primeiras amostras dos conjuntos completos, podemos perceber que a acurácia do modelo é de aproximadamente 100%. Isso ocorre pois, tanto no conjunto de treinamento quanto no de teste, todas as primeiras 200 amostras são da classe **negative** e não há nenhuma amostra cuja classe seja **positive**, ou seja, esses conjuntos não são balanceados e possuem apenas uma classe dentre todas as 200 amostras.
Assim, a tarefa de classificação se torna muito simples pois o modelo acerta sempre apenas “chutando” **negative**. Para fazer um treinamento mais eficaz e que realmente faça com que o modelo aprenda algo, é necessário utilizar um conjunto de amostras balanceadas dentre as classes.
O código utilizado para este exercício está contido no Notebook 1.

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.

O código modificado está contido no Notebook 1.

II - Dataset

Exercício II.1:

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.

O número de amostras positivas e negativas do dataset se manteve o mesmo ao mudar a forma de realizar o processamento das amostras. Sendo assim, o número de amostras positivas e de amostras negativas foi, respectivamente, **12500** e **12500**.
O código utilizado para fazer essa contagem está contido no Notebook 1.

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

O número médio de palavras codificadas em cada vetor one-hot foi: **111.81108**. Podemos reparar que esse valor diminuiu quando consideramos o número médio de palavras calculados a partir do texto cru, sem ser codificado como vetores one-hot, que foi 233.7872.
O código utilizado para fazer essa contagem está contido no Notebook 1.

II.1.c) e explique a diferença.

Essa diminuição no número médio de palavras por amostra se deve ao fato de que, ao codificar o texto como um vetor one-hot, a palavras duplicadas, ou seja que aparecem mais de uma vez em uma amostra de texto, são contadas apenas uma vez enquanto anteriormente estávamos contando todas as ocorrências das palavras nas amostras. Portanto, o número médio de palavras por amostra codificada, como um vetor one-hot, é menor ou igual ao número médio de palavras calculado quando não há este processamento, pois as palavras com mais de uma ocorrência são contadas diversas vezes.

Exercício II.2:

II.2.a)

Tempo do laço:	**1.0930 sec**
Tempo do forward	**0.4538 sec**
Tempo do backward	**0.1968 sec**

Conclusão: A demora principal está na execução da linha: 13 → *for inputs, targets in train_loader:*.

II.2.b) Trecho que precisa ser otimizado. (Esse é um problema bem mais difícil) A dica aqui é que precisa muito de conceitos de iterador e programação orientada a objetos.

O trecho do código que precisa ser otimizado é a implementação da classe *IMDBDataset*, mais especificamente a implementação do método *__getitem__*. Isso se deve ao fato de que, na linha 13, o *DataLoader train_loader*, por se tratar de um iterador, carrega os dados ao longo da execução do laço de treinamento e, ao carregar esses dados do *dataset*, a função *__getitem__* é chamada a cada iteração. Esse comportamento faz com que todas as amostras do *dataset* sejam codificadas para vetores *one-hot* em tempo de treinamento, em vez de serem pré-processadas. Além de aumentar o tempo de execução do laço de treinamento, essa abordagem é ineficiente pois uma mesma amostra será codificada uma vez por época, resultando em diversos processamentos para a mesma amostra conforme o número de épocas cresce.

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

Para otimizar esse código, bastou alterar a implementação da classe *IMDBDataset*, de forma que o pré-processamento e a codificação das amostras para vetores *one-hot* fossem realizados no momento de instanciação do objeto do *dataset*, em vez de no momento de acesso por índice/indexação que ocasiona da chamada do método *__getitem__*. Assim, apenas movi a parte de pré-processamento contida no método *__getitem__* para dentro do método *__init__* e realizei a codificação para todas as amostras utilizando *list comprehension*.
Assim, o código alterado está contido no Notebook 1.

Exercício II.3:

II.3.a) Gráfico Acurácia vs LR

Para encontrarmos uma boa escolha de Learning Rate (LR), testei 75 valores diferentes uniformemente distribuídos no intervalo [1e-5, 1e-1], que foram gerados a partir do código *np.linspace(1e-5, 1e-1, num=75)*. Dessa forma, treinei e avaliei o modelo para cada possibilidade de LR e, após as 75 acurácias decorrentes deste processo, plotei a acurácia em função da LR utilizando o framework *matplotlib*:

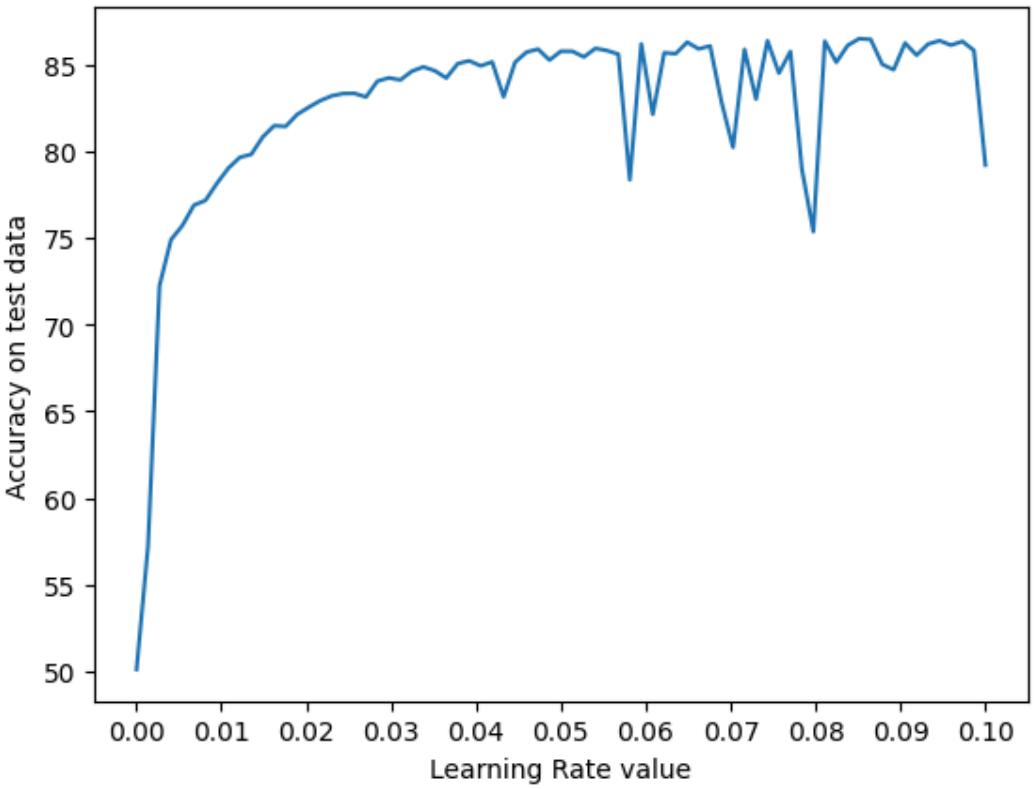


Figura 1: Gráfico Acurácia vs LR

O código utilizado para fazer esta análise está contido no Notebook 1.

II.3.b) Valor ótimo do LR (para isso é desejável que o LR ótimo que forneça maior acurácia no conjunto de testes seja maior que usar um LR menor e um LR maior que o LR ótimo.)

Após avaliar 75 valores possíveis de LR, o valor de LR que resultou na melhor acurácia foi: 0.06892202702702703 com 86.34% de acurácia.
Porém, ao analisarmos a curva plotada no gráfico acima, podemos notar que, para valores de LR maiores do que 0.05, o valor da acurácia se torna mais instável, ou seja, apesar de obterem bons resultados, conforme o valor da LR vai se tornando cada vez mais próximo de 1e-1, a acurácia resultante começa a se tornar cada vez mais instável e a ter uma maior variabilidade.
Dado isso, considerei o valor ótimo de LR como **0.05**, que, após avaliação do modelo, resultou em uma acurácia de **85.228%**.

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.

A equação utilizada no gradiente descendente é a seguinte:

$$\theta^{(i+1)} := \theta^{(i)} - \alpha \nabla J(\theta^{(i)})$$

que é equivalente a:

$$\theta_j^{(i+1)} := \theta_j^{(i)} - \alpha \frac{\partial}{\partial \theta_j^{(i)}} J(\theta^{(i)})$$

Nesta equação, o $\theta^{(i)}$ é o vetor dos parâmetros do modelo na i-ésima iteração do laço de treinamento, o $J(\theta^{(i)})$ é a função de custo em função dos parâmetros do modelo e, por fim, o hiperparâmetro α é a Learning Rate de treinamento do modelo.
Se considerarmos uma função $J(\theta)$ convexa, o gradiente $\nabla J(\theta)$ é um vetor que, partindo do ponto θ avaliado, aponta para o sentido contrário de onde fica o mínimo de $J(\theta)$. Portanto, $-\nabla J(\theta)$ aponta para o mínimo da função e $\theta' = \theta - \nabla J(\theta)$ é um valor de θ, idealmente, mais próximo do mínimo da função custo.
Portanto, nesse caso, a Learning Rate α está diretamente relacionado a quanto o gradiente do custo irá influenciar os parâmetros (*weights*) da rede, ao longo do treinamento, ou seja, quanto maior o α, maior o "salto" de variação dos parâmetros, que será feito a cada batch, em direção ao mínimo da função custo $J(\theta)$. Resumidamente, o Learning Rate é, como o próprio nome diz, a **taxa de velocidade de minimização da função de custo** $J(\theta)$, ou seja, a **taxa de velocidade de aprendizado do modelo**.

Exercício II.4:

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

Para melhorar o tokenizador, apenas alterei a forma como o vocabulário é construído, na célula inicial do notebook, e como as palavras são codificadas na função *encode_sentence*, a partir da construção de uma função *normalize_string*. Para normalizar cada palavra, tanto na construção do vocabulário quanto no pré-processamento do *dataset*: primeiramente realizei transformei caracteres Unicode em caracteres ASCII similares (o que trata caracteres que contém acentos) utilizando o projeto Unidecode (https://pypi.org/project/Unidecode/), depois realizei um *strip* removendo todos caracteres de pontuação (obtidos através da string *string.punctuation*) e finalizei o processamento utilizando a função *string.lower* para deixar todos os seus caracteres do texto minúsculos.
Assim, os trechos modificados estão no Notebook 1.
É possível notar que, por centralizar as alterações de tokenização na implementação da função *encode_sentence* e *normalize_string*, não foi necessário alterar nenhum código da 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.

Após esta técnica diferente de tokenização, o número de unknown tokens diminuiu muito e o vocabulário, consequentemente se tornou muito mais proporcionalmente abrangente. A comparação entre os valores antigos e o atual de unknown tokens pode ser feita a partir da tabela abaixo:

	Tokenização antiga	Tokenização nova
Unknown Tokens	260617	112077

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.

Executando o notebook inteiro, o novo valor de acurácia obtido após a melhoria do tokenizador foi: **87.752%.**

III - DataLoader

Em primeiro lugar anote a acurácia do notebook com as melhorias de eficiência de rodar em GPU, com ajustes de LR e do tokenizador. Em seguida mude o parâmetro `shuffle` na construção do objeto `train_loader` para False e execute novamente o notebook por completo e meça novamente a acurácia:

Shuffle	Acurácia
True	87.752%
False	50.0%

Tabela 1 - Acurácia considerando a variação do Shuffle

Exercício III.1:

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

O tamanho do *batch* utilizado no treinamento de redes neurais tem um impacto direto na eficácia e eficiência deste processo de otimização e de minimização da função custo. Assim, vale destacar que a relação entre o método de gradiente descendente e o *batch size* ocorre principalmente em 3 variações:

1. *Batch Gradient Descent*: Nesta variação, o cálculo da loss e, consequentemente, do gradiente leva em consideração todas as amostras no conjunto. Dessa forma, o método de gradiente descendente realiza apenas um passo para cada época e o tamanho do *batch* é o tamanho do conjunto de dados de treinamento. Esse método pode levar a uma convergência mais estável para um mínimo global, em funções de custo convexas, porém se mostra extremamente inviável e computacionalmente caro conforme o tamanho do conjunto de dados cresce. Além disso, esse método não contribui diretamente para a capacidade de generalização do modelo treinado.
2. *Stochastic Gradient Descent (SGD)*: Este método de gradiente descendente computa a loss, o gradiente e atualiza os parâmetros para cada amostra de treinamento, ou seja, com um *batch size* igual a 1. Isso torna o *SGD* muito mais rápido do que o *Batch Gradient Descent*, especialmente para grandes conjuntos de dados, e pode ajudar a escapar de mínimos locais da função de perda devido à sua natureza estocástica. No entanto, a trajetória de otimização é mais ruidosa ou

irregular, o que pode levar a uma convergência mais lenta ou a uma convergência para um mínimo local, em vez do global.

3. *Mini-batch Gradient Descent*: É um método que combina o *Batch Gradient Descent* e o *Stochastic Gradient Descent*, dividindo o conjunto de dados em pequenos lotes/*batches* (de tamanho maior do que 1 e menor do que o números de amostras no conjunto) e atualizando os parâmetros para cada *batch*. Essa abordagem proporciona um equilíbrio entre a eficiência computacional e a estabilidade da convergência, de forma que o treinamento se beneficie tanto da rapidez do *SGD* quanto da estabilidade do *Batch Gradient Descent*. Assim, para o treinamento, é necessário fazer uma boa escolha de *batch size* dado este hiperparâmetro influencia tanto a velocidade de convergência quanto a qualidade do modelo final.

Portanto, entendendo a decorrências das duas primeiras variações do método de *gradient descent* podemos dizer que o uso do *batch* através da terceira técnica, o *Mini-batch Gradient Descent*, possui duas principais vantagens no treinamento:

1. Melhor **qualidade do passo de convergência** durante o treinamento e melhor **capacidade de generalização** do modelo resultante, já que o uso de *batches* possibilita que o modelo avalie um lote de amostras com diversas classes antes de calcular o gradiente e, assim tenha mais estabilidade durante o treinamento,ao mesmo tempo que, como ocorre no *SGD*, pode introduzir ruídos e desbalanceamentos em *batches* menores, o que evita uma convergência extremamente suave e, assim, contribui para a capacidade de generalização do modelo e para evitar o *overfitting*.
2. Melhor eficiência computacional do treinamento, pois possibilita que mais amostras sejam processadas simultaneamente e permite uma maior paralelização, o que tira maior proveito de componentes de hardware como GPUs.

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

Considerando os pontos discutidos no exercício anterior, realizar o embaralhamento do conjunto de dados antes de realizar a divisão dos *batches* contribui ainda mais com todas as vantagens descritas no item 1 do exercício acima.
Isso ocorre pois, ao introduzir uma aleatoriedade aos *batches*, a diversidade das amostras contidas em um mesmo *batch* é maior e isso possibilita com que o modelo avalie **diferentes** classes durante uma iteração do laço de treinamento, mas também evita um desbalanceamento extremo entre classes contidas em um mesmo *batch* ou que até uma classe não esteja presente em diversos *batches*.

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

Ao alterar o `shuffle=False` no instanciamento do objeto `test_loader`, não ocorre mudança na acurácia do modelo pois o modelo não está sendo treinado ao ser avaliado no conjunto de treinamento, ou seja, seus parâmetros já estão congelados. Sendo assim, não faz diferença a ordem em que as amostras sejam avaliadas, desde que todas sejam avaliadas.

Exercício III.2:

III.2.a) Faça um laço no objeto `train_loader` e meça quantas iterações o Loader tem. Mostre o código para calcular essas iterações. Explique o valor encontrado.

O valor de iterações do `train_loader` encontrado foi: **196**. Isso ocorre pois o número de iterações a serem realizadas será igual a *numero_de_amostras/batch_size* arredondado para cima, já que a cada iteração *batch_size* amostras são processadas, com exceção da última iteração que pode acabar processando um número menor que o *batch size* de amostras.
O código utilizado está contido no Notebook 1.

III.2.b) Imprima o número de amostras do último batch do `train_loader` e justifique o valor encontrado? Ele pode ser menor que o `batch_size`?

O número de amostras no último batch do conjunto de treinamento foi: **40**. Isso ocorre pois, se o número de amostras no conjunto não é um múltiplo de `batch_size`, sobram algumas amostras que são processadas em um batch menor do que os outros. Podemos observar isso matematicamente nos seguintes cálculos:
$number\_of\_samples/batch\_size = 25000/128 = 195,3125$ e $128 \times 0,3125 = 40$.
O código utilizado está contido no Notebook 1.

III.2.c) Calcule R, a relação do número de amostras positivas sobre o número de amostras no *batch* e no final encontre o valor médio de R, para ver se o data loader está entregando *batches* balanceados. Desta vez, em vez de fazer um laço explícito, utilize *list comprehension* para criar uma lista contendo a relação R de cada amostra no batch. No final, calcule a média dos elementos da lista para fornecer a resposta final.

O valor de R encontrado foi: **0.5018**. Isso mostra que, após o embaralhamento das amostras no conjunto de treinamento, os *batches* ficaram relativamente balanceados, no quesito quantidade de classes, o que era esperado considerando que a quantidade de classes **positive** e **negative** é a mesma no conjunto de treinamento como um todo.
Para esses cálculos, o código utilizado, que usa *list comprehension* conforme pedido no enunciado, está contido no Notebook 1.

III.2.d) Mostre a estrutura de um dos *batches*. Cada *batch* foi criado no método `__getitem__` do **Dataset**, linha 20. É formado por uma tupla com o primeiro elemento sendo a codificação *one-hot* do texto e o segundo elemento o *target* esperado, indicando positivo ou negativo. Mostre o *shape* (linhas e colunas) e o tipo de dado (*float* ou *integer*), tanto da entrada da rede como do *target* esperado. Desta vez selecione um elemento do batch do train_loader utilizando as funções `next` e `iter`: `batch = next(iter(train_loader))`.

A partir da execução do código contido no Notebook 1, podemos observar que a estrutura do *batch* se dá por uma tupla de dois elementos:

- `inputs`: um *tensor* de *shape* (128, 20001) e seu *dtype* é *torch.float32*. A primeira dimensão do *tensor* representa cada amostra do *batch* e, portanto, tem o tamanho

da dimensão igual ao *batch size*. Já a segunda a dimensão do *tensor* diz respeito às amostras do conjunto de dados codificadas no formato *one-hot encoding* com o tamanho do vetor de acordo com o tamanho vocabulário abrangido pelo modelo, logo, possui um tamanho de 2001, representando as 2000 palavras do vocabulário mais 1 token desconhecido. Conforme a documentação do PyTorch (https://pytorch.org/docs/stable/generated/torch.zeros.html), o *dtype* do tensor foi atribuído automaticamente pelo PyTorch em sua criação que utilizou a função *tensor.zeros* e atribuiu o tipo *default*, que pode ser alterado a partir da função *torch.set_default_type*;

- `targets`: um *tensor* de *shape* (128) e seu *dtype* é *torch.int64*. A única dimensão do *tensor* diz respeito a cada amostra do *batch*, considerando que o *target* de cada amostra em si é expressa apenas por um número inteiro que pode ser 0 ou 1. Nesse caso, o *dtype* do *tensor* foi inferido em sua criação a partir dos dados passados, cujo os valores possíveis são apenas 0 e 1, para o construtor, segundo a documentação do PyTorch (https://pytorch.org/docs/stable/generated/torch.tensor.html).

Exercício III.3:

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 512, preencha a tabela a seguir e comente sobre os resultados.

	batch_size = 1	batch_size = 128	batch_size = 512
Loss final da época 1	0.0147	0.4746	0.6717
Loss final da época 5	0.0053	0.2773	0.4236

Analisando os resultados, podemos perceber que, quando o *batch_size* é apenas 1 amostra, é possível observar uma rápida diminuição da loss, alcançando um valor de praticamente 0 na época 5. Isso ocorre porque, com um *batch_size* tão pequeno, o modelo atualiza seus pesos após processar cada amostra, o que permite uma convergência muito rápida. No entanto, isso também pode levar a uma alta variabilidade e uma convergência mais instável e irregular durante o treinamento, pois cada atualização é baseada apenas em uma única amostra, tornando o modelo muito sensível ao ruído dos dados. Como discutido no exercício anterior, esse método é conhecido como *Stochastic Gradient Descent (SGD).*
Já quando o *batch_size* é 128, o modelo demonstra um comportamento de aprendizagem mais estável e regular durante o treinamento. O valor de loss decresce de forma consistente, mas não tão rapidamente se comparado aos valores obtidos quando o *batch_size* é 1. Isso ocorre porque, ao avaliar e atualizar o modelo com 128 amostras por vez, o efeito de cada amostra individual é suavizado, levando a atualizações de peso mais estáveis e confiáveis. Esse tamanho de *batch* pode ser considerado um bom meio-termo, pois combina eficiência computacional e outras vantagens do uso de *batches*, discutidas também no exercício acima, e não apresenta os mesmos problemas que um *batch_size* pequeno como 1.
E para o *batch_size* maior ainda, igual a 521, é possível notar que o valor de loss diminui de forma mais lenta. Isso pode ser atribuído ao fato de que grandes *batches* tendem a suavizar a curva de convergência na função de custo, o que pode levar a atualizações de peso

menores e, consequentemente, “saltos” menores em direção ao mínimo da função de custo. Embora isso possa resultar em uma convergência mais estável e menos sensível ao ruído dos dados, também pode retardar significativamente o processo de aprendizagem e potencialmente levar a mínimos locais da função de custo.

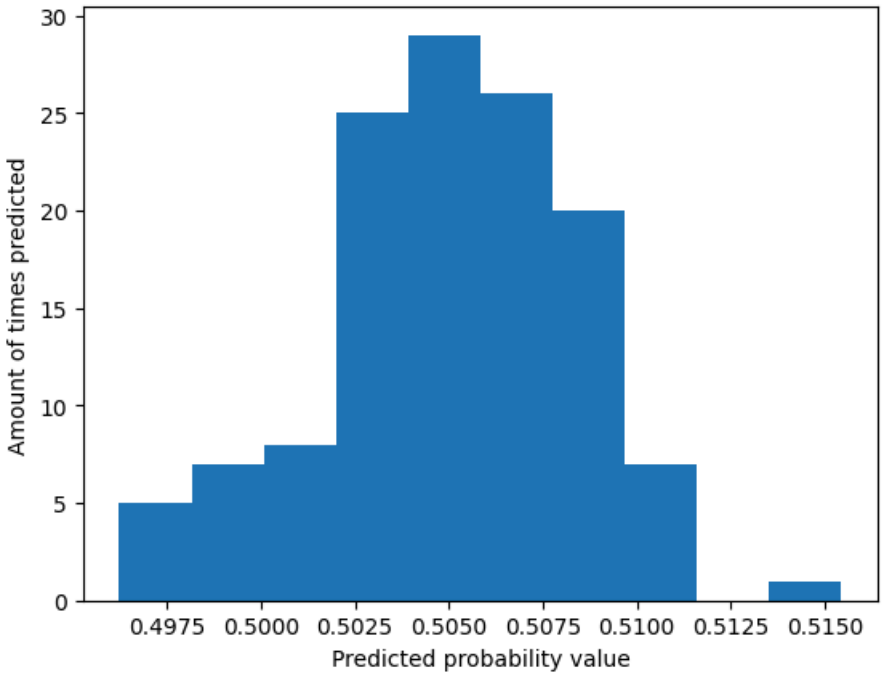
IV - Modelo MLP

Exercício IV.1:

IV.1.a) Faça a predição do modelo utilizando um batch do train_loader: extraia um batch do `train_loader`, chame de `(inputs, targets)`, onde `input` é a entrada da rede e `target` é a classe (positiva ou negativa) esperada. Como a rede está com seus parâmetros (*weights*) aleatórios, o logito de saída da rede será um valor aleatório, porém a chamada irá executar sem erros: `logits = model( inputs)` aplique a função sigmoidal ao logito para convertê-lo numa probabilidade de valor entre 0 e 1.

Para realizar a avaliação conforme o enunciado, o código implementado está contido no Notebook 1.
Assim, analisando os resultados, foram obtidos os seguintes valores relacionados às predições realizadas para o primeiro *batch* do conjunto de treinamento:

- **Valor médio** de predição: **0.5051;**
- **Valor mínimo** de predição: **0.4962;**
- **Valor máximo** de predição: **0.5154;**
- **Desvio padrão** dos valores: **0.0034;**
- **Histograma** dos valores de predição:

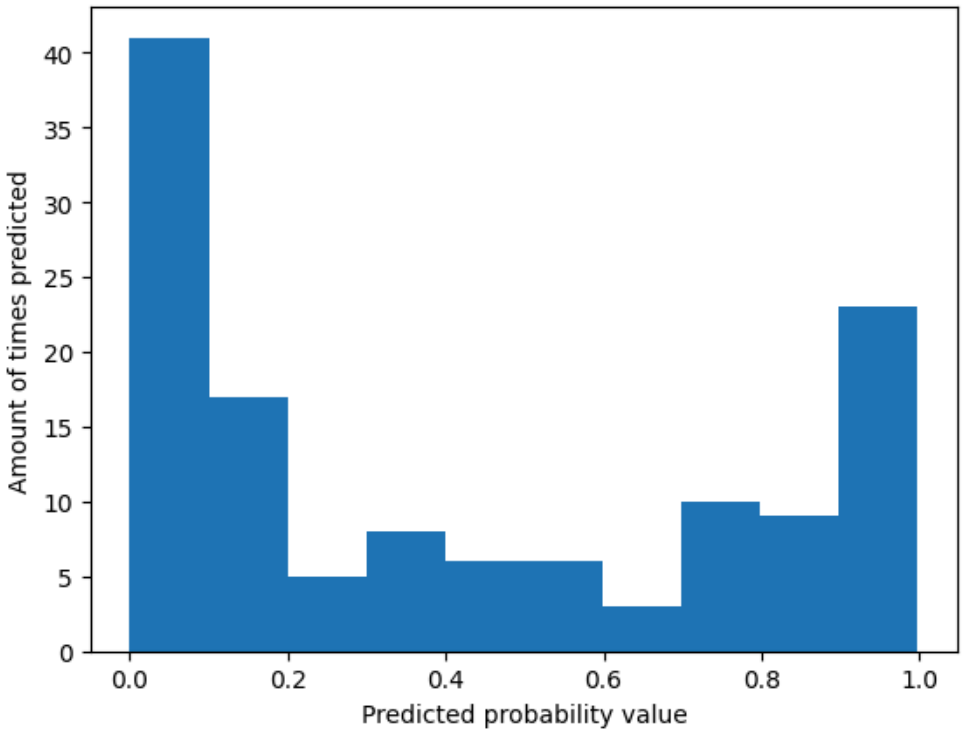


IV.1.b) Agora, treine a rede executando o notebook todo e verifique se a acurácia está alta. Agora repita o exercício anterior, porém agora, compare o valor da probabilidade encontrada com o `target` esperado e verifique se ele acertou. Você pode considerar que

se a probabilidade for maior que 0.5, pode-se dar o target 1 e se for menor que 0.5, o target 0. Observe isso que é feito na linha 11 da seção **VI - Avaliação**.
Calcule então a acurácia do modelo treinado no primeiro batch dos dados e mostre o código para calcular essa acurácia do primeiro batch e a acurácia obtida:

Similarmente, os resultados foram analisados da mesma forma que no exercício anterior e foram obtidos os seguintes valores relacionados às predições realizadas para o primeiro *batch* do conjunto de treinamento:

- **Valor médio** de predição: **0.4138**;
- **Valor mínimo** de predição: **0.0000**;
- **Valor máximo** de predição: **0.9979**;
- **Desvio padrão** dos valores: **0.3660**.
- **Histograma** dos valores de predição:



IV.1.c) Número de parâmetros do modelo

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

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

Tabela 2: Número de parâmetros (weights e biases) do modelo.

Para calcular a quantidade de parâmetros do modelo, por camada e ao todo, e o *shape* dos *tensors* de parâmetros em cada camada, foi utilizado o código que está contido no Notebook 1.
Analogamente, para calcular o número de parâmetros dentro de um *tensor*, também poderia ser utilizada a função *torch.numel* em vez de multiplicar os valores contidos dentro do *shape* do *tensor*.

V - Treinamento

Exercício V.1:

V.1.a) Qual é o valor teórico da Loss quando o modelo não está treinado, mas apenas inicializado? Isto é, a probabilidade predita tanto para a classe 0 como para a classe 1, é sempre 0,5 ? Justifique. Atenção: na equação da Entropia Cruzada utilize o logaritmo natural.

Para calcularmos o valor teórico da loss quando o modelo não está treinado e apenas inicializado, basta calcular a equação da Entropia Cruzada e considerando que $\hat{y}_i = 0,5$ sempre. Assim, desenvolvendo os cálculos temos:

$$
\begin{aligned}
L(y, 0,5) &= -\frac{1}{N}\sum_{i=1}^{N} y_i log_e(0,5) + (1 - y_i) log_e(0,5) \\
&= -\frac{1}{N}\sum_{i=1}^{N} log_e(0,5)(y_i + 1 - y_i) \\
&= -\frac{1}{N} N log_e(0,5) \\
&= -log_e(0,5) \\
&= 0,69314718056
\end{aligned}
$$

Portanto, temos que o valor teórico inicial da loss é **0,69314718056**, o que é uma consequência do modelo sempre atribuir a probabilidade de 0,5 para todas as classes.

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

Foi feita a avaliação do modelo **não treinado e apenas inicializado**, no primeiro batch do conjunto de treinamento, a partir do seguinte código contido no Notebook 1.
Então, a loss nesse batch foi calculada a partir de uma função *bce_loss* implementada com operações da equação descrita anteriormente, cujo código está também contido no Notebook 1.
E o valor obtido para a loss foi **0.6936115026473999**. Esse resultado foi extremamente próximo do esperado e mostrou um comportamento de predições praticamente aleatórias

por parte do modelo, considerando o valor teórico inicial para a loss calculado no exercício anterior.
Porém, esse comportamento, apesar de cumprir com o calculado teoricamente, não ocorre sempre pois, na prática, o valor inicial da loss pode variar dependendo dos valores aleatórios que são atribuídos aos parâmetros do modelo durante a inicialização e também das técnicas de inicialização que são utilizadas, como *Xavier Initialization*, *He Initialization* e outras.

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.**

Após pesquisar pela documentação da classe *BCELoss* do PyTorch, utilizei a função e avaliei a loss das predições a partir do seguinte código contido no Notebook 1.
E, como esperado, o resultado calculado foi o mesmo ao obtido no exercício anterior: **0.6936115026473999**.

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

Repetindo o mesmo exercício, fiz a implementação da função *bce_with_logits_loss* e avaliei o resultado obtido utilizando tanto a função implementada quanto a classe *nn.BCEWithLogitsLoss*, conforme o código contido no Notebook 1.
Analisando os resultados, vemos que os valores calculados foram os mesmos das funções executadas anteriormente (*bce_loss* e *nn.BCELoss*): **0.6936115026473999**.

Exercício V.2:

V.2.a) Modifique a célula do laço de treinamento de modo que a primeira Loss a ser impressa seja a Loss com o modelo inicializado (isto é, sem nenhum treinamento), fornecendo a Loss esperada conforme os exercícios feitos anteriormente. Observe que desta forma, fica fácil verificar se o seu modelo está correto e a Loss está sendo calculada corretamente.
Atenção: Mantenha esse código da impressão do valor da Loss inicial, antes do treinamento, nesta célula, pois ela é sempre útil para verificar se não tem nada errado, antes de começar o treinamento.
Atenção: A Loss antes da época 0 deve ser calculada em todas as 25.000 amostras.

Loss antes da época 0	0.6938

Loss após época 0	0.5262
Loss após época 5	0.2710

O código alterado baseado na célula do laço de treinamento está contido no Notebook 1.

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?

Para que o treinamento comece novamente com um modelo aleatório, é necessário inicializá-lo novamente, ou seja, atribuir valores aleatórios (seguindo a devida estratégia de inicialização de parâmetros) para todos os seus parâmetros novamente. Isso é possível de duas maneiras: chamando uma função de inicialização de parâmetros para cada camada do modelo, passando-as como argumento; ou, de uma outra forma mais simples e relativamente eficiente, instanciando novamente o objeto utilizado, nesse caso o *model*, da classe *OneHotMLP*, pois, instanciando novamente o objeto, a variável irá criar um novo objeto na memória do programa e o objeto antigo, com os seus parâmetros possivelmente já treinados, serão sobrescritos.
Para isso, basta executar uma célula que instancie novamente o objeto *model* da classe *OneHotMLP*. Uma célula de código que faz isso é a célula que contém a própria declaração da classe *OneHotMLP*:

```python
class OneHotMLP(nn.Module):
  def __init__(self, vocab_size):
    super(OneHotMLP, self).__init__()

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

    self.relu = nn.ReLU()

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

# Model instantiation
model = OneHotMLP(vocab_size)
```

Ou uma célula que contenha apenas o seguinte código:

```python
# Instantiate new model
model = OneHotMLP(vocab_size)
```

VI - Avaliação

Exercício VI.1:

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

O número de amostras que está sendo considerado na seção de avaliação e existem no conjunto de teste é: **25000**, que é a mesma quantidade de amostras no conjunto de treinamento.

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

O comando `model.eval()` é responsável por colocar o modelo em modo de avaliação, o que resulta na alteração do comportamento de certas camadas da rede (se existam no modelo) que agem de forma diferente durante o período de treinamento e o período de inferência ou de avaliação. Dentre as camadas afetadas por esse modo de avaliação, estão:

- **Camadas de dropout**: Durante o treinamento, essas camadas "desligam" aleatoriamente as saídas, ou seja, zeram as ativações, de alguns neurônios e servem como uma técnica de regularização. No entanto, durante a avaliação, todas as saídas dos neurônios são mantidas para garantir uma inferência consistente.
- **Camadas de *batch normalization* (BatchNorm)**: Durante o treinamento, essas camadas normalizam a saída utilizando a média e o desvio padrão do *batch* atual. Durante a avaliação, essas camadas passam a utilizar a média e o desvio padrão acumulados durante o processo de treinamento, o que proporciona uma inferência estável.

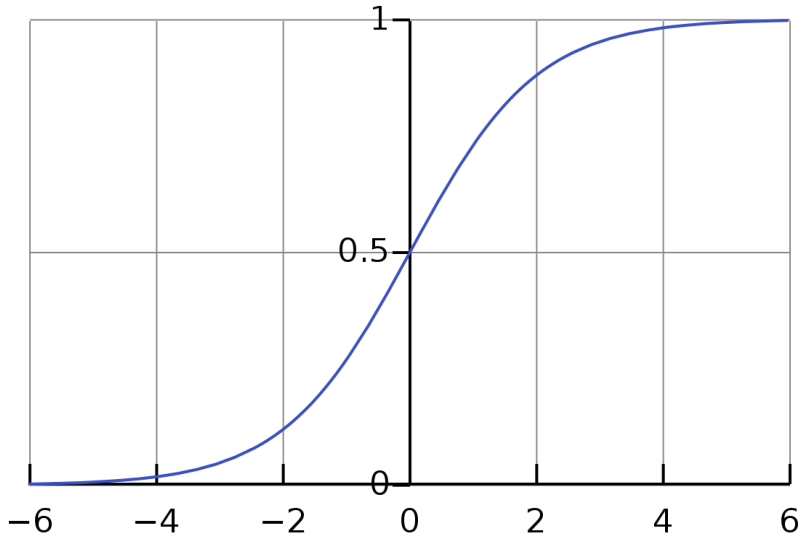
Já o comando `with torch.no_grad()` é responsável por gerar um contexto em que o cálculo de gradientes é desativado. Durante o treinamento, o cálculo dos gradientes é necessário para a atualização dos pesos e a execução do método de otimização/minimização por gradiente descendente. Porém, devido ao fato de que, durante a avaliação e a inferência, os parâmetros do modelo não são alterados, realizar esse cálculo de gradientes é desnecessário e utilizar `with torch.no_grad()` reduz o uso de memória, evita o armazenamento de operações para cálculos de gradiente e, consequentemente, acelera o processo de inferência. Assim, essa função é particularmente útil para tarefas de inferência em modelos grandes.
Portanto, é comum combinar ambos os comandos para realizar o processo de avaliação do modelo ou para utilizá-lo para inferência, otimizando os recursos computacionais e garantindo que os resultados sejam consistentes e não afetados por camadas que têm comportamentos diferentes durante o treinamento.

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

Considerando os seguintes aspectos sobre o processamento das predições de probabilidade para transformá-las em predições de classe:

- Para que uma predição seja 0, é necessário apenas que a probabilidade atribuída pelo modelo seja menor do que 0,5;
- Para que uma predição seja 1 é necessário que a probabilidade seja maior ou igual a 0,5.

E considerando que a curva descrita pela função Sigmoid é:



Uma forma mais simples de processar as predições é utilizar *list comprehension* com a seguinte ideia: *predicted = [1 if logit >= 0 else 0 for logit in logits]*.
Assim, o código contendo essa simplificação está contido no Notebook 1.
Executando o código e comparando os resultados, ambos indicaram uma acurácia de: **87.364%**.

Exercício VI.2:

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?

Considerando que a loss teórica de um modelo aleatório de 2 classes é $CE = -log_e(0,5)$, o valor da perplexidade será:

$$PPL = e^{CE} = e^{-log_e(0,5)} = \frac{1}{e^{log_e(0,5)}} = \frac{1}{0,5} = 2$$

$$\therefore PPL = 2$$

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?

Sabendo que a fórmula da Cross-Entropy é $CE = -\sum_{i=1}^{N} y_i \log_e(\hat{y}_i)$ para N classes, sabemos que, de maneira análoga ao calculado no exercício V.1.a, o valor teórico da loss nesse caso será $CE = -log_e(\frac{1}{N})$ e, similarmente ao exercício anterior, a perplexidade nesse caso será:

$$PPL = e^{-log_e(\frac{1}{N})} = \frac{1}{e^{log_e(\frac{1}{N})}} = \frac{1}{\frac{1}{N}} = N$$

$$\therefore PPL = N$$

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

Quando o modelo acerta todas as classes, com 100% de probabilidade, o valor da loss será $CE = 0$, logo, a perplexidade do modelo será $PPL = e^0 = 1 \therefore PPL = 1.$

Exercício VI.3:

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.
Modificando o código conforme o enunciado, as alterações do código estão no Notebook 1.

VI.3.b) Incorpore agora o código da seção VI - Avaliação, durante o laço de treinamento, de modo que após cada época de treinamento, os seguintes valores sejam calculados:

- Laço de treinamento:
    - Loss no conjunto de treinamento
    - Perplexidade no conjunto de treinamento

- Laço de validação:
    - Loss no conjunto de teste
    - Perplexidade no conjunto de teste
    - Acurácia no conjunto de teste

Assim, o código com as alterações no laço de treinamento, para também calcular a perplexidade no conjunto de treinamento, está contido no Notebook 1.