INSTITUTO SUPERIOR TÉCNICO

MESTRADO BOLONHA EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

APRENDIZAGEM PROFUNDA

Homework 1

Grupo 61 - Lab 10

Autores:	*Número:*
Manuel Ruivo	87061
Luís Coelho	90127
Carlos Gomes	98417

12 de janeiro de 2022

Índice

Questão 1

Ao longo desta questão pretendemos avaliar a convexidade de algumas funções de perdas nomeadamente as funções de perda binária e logística multinomial.

Alínea 1.1 - Função Sigmóide

Partindo da função sigmóide, foi nos pedido para provar a igualdade $\sigma'(z) = \sigma(z)(1-\sigma(z))$. Apresentamos de seguida os cálculos efectuados:

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

$$\sigma'(z) = \frac{-(1+e^{-z})'}{(1+e^{-z})^2}$$

$$\frac{e^{-z}}{(1+e^{-z})(1+e^{z})} = \frac{1}{(1+e^{-z})} \cdot \frac{(e^{-z})}{(1+e^{-z})}$$

Dado que :

$$1-\sigma(z) = 1 - \frac{1}{1+e^{-z}} = \frac{1+e^{-z}-1}{1+e^{-z}} = \frac{e^{-z}}{1+e^{-z}} \tag{2}$$

Fica provado que:

$$\sigma'(z) = \sigma(z)(1-\sigma(z)) \tag{3}$$

Alínea 1.2 - Concavidade das *loss function*

Foi nos pedido para provar a concavidade (em z) da função de perda binária para uma situação de regressão logística. Assumindo y=1, obtemos:

$$\begin{aligned} L'(z;y=1) &= \frac{\partial}{\partial z}\Big(-log(\sigma(z))\Big) \\ &= -\frac{\sigma'(z)}{\sigma(z)} \\ &= -\frac{\sigma(z)(1-\sigma(z))}{\sigma(z)} \\ &= \sigma(z) - 1 \\ &= -\frac{e^{-z}}{1+e^{-z}} \end{aligned}$$

Vejamos agora a segunda derivada e o seu sinal para averiguar a concavidade.

$$\begin{aligned} L''(z;y=1) &= \frac{\partial}{\partial z}\Big(L'(z;y=1)\Big) \\ &= \frac{\partial}{\partial z}\Big(\sigma(z)-1\Big) \\ &= \sigma'(z) \\ &= \sigma(z)(1-\sigma(z)) \\ &= \frac{e^{-z}}{(1+e^{-z})^2} \end{aligned}$$

Podemos observar que a segunda derivada é composta por uma fracção cujos numerador e denominador assumem valores sempre positivos, provando assim que a função $L(z, y = 1)$ é convexa (em função de z) em todo o seu domínio.

Alínea 1.3 - *Softmax*

Consideremos agora o caso multi-classes. Sabendo que a função *softmax* é dada por:

$$softmax(z_j) = \frac{exp(z_j)}{\sum_{k=1}^{K} exp(z_k)}, j = 1, ..., k \tag{4}$$

Vejamos como obter a matriz Jacobiana desta função.

Por simplicidade, consideremos a seguinte notação: $S(z_j) = softmax(z_j)$ e $S' = \frac{\partial S(z_j)}{\partial z_k}$. Considerando a (j,k)-ésima entrada da matriz Jacobiana, é necessário avaliar as seguintes situações:

Para $j = k$ temos,

$$\begin{aligned} S' &= \frac{exp(z_j) \cdot \sum_k exp(z_k)}{\Big(\sum_k exp(z_k)\Big)^2} - \frac{exp(z_k) \cdot exp(z_k)}{\Big(\sum_k exp(z_k)\Big)^2} \\ &= S(z_j) - \big[S(z_k)\big]^2 \end{aligned}$$

Para $j \neq k$ temos,

$$\begin{aligned} S' &= 0 - \frac{exp(z_j) \cdot exp(z_k)}{\Big(\sum_k exp(z_k)\Big)^2} \\ &= -S(z_j) \cdot S(z_k) \end{aligned}$$

Podemos então concluir que a matriz Jacobiana de dimensões $K \times K$ assumirá as seguintes expressões para a (j,k)-ésima posição:

$$S' = \begin{cases} S(z_j) - S(z_k)^2, para\, j = k \\ -S(z_j) \cdot S(z_k), para\, j \neq k \end{cases} \tag{5}$$

Alínea 1.4 - *Multinomial Logistic Loss*

Mantendo a notação da alínea anterior, sabemos que a função *Multinomial logistic loss* é definida como $\Rightarrow L(z; y = j) = -log(S(z_j))$. Assim, à semelhança do que se fez para no 1.3., para obter o gradiente precisamos de considerar os seguintes casos:

Quando $j = k$,

$$\begin{aligned} \nabla(j = k) &= \frac{\partial}{\partial z}\Big(-log(S(z_j))\Big) \\ &= -\frac{1}{S(z_j)} \cdot S' \\ &= S(z_j) - 1 \end{aligned}$$

Quando $j \neq k$,

$$\begin{aligned}\nabla(j \neq k) &= \frac{\partial}{\partial z}\Big(-log(S(z_j))\Big) \\ &= -\frac{1}{S(z_j)} \cdot S' \\ &= S(z_k)\end{aligned}$$

Temos assim que $\nabla S(z) = \begin{cases} S(z_j) - 1, \text{ para j = k} \\ S(z_k), \text{ para j} \neq k \end{cases}$

Vejamos agora o caso da segunda derivada isto é da Matriz Hessiana. Desta vez, podemos considerar o valores $j = k$ e $j \neq k$ de um só vez visto que o resultado é equivalente.

$$H = \frac{\partial}{\partial z_k}\Big(\nabla S(z_j)\Big) = S(z_j) - S(z_j)^2$$

Podemos então observar que a matriz Hessiana assume valores sempre positivos, levando a função de perdas em causa a ser convexa em z.

Alínea 1.5 - Concavidade em relação a (W,b)

Provámos na questões anteriores que a função *multinomial logistic loss (MLL)* é uma função convexa. Adicionalmente, observando a composição de z, verificamos que este é uma combinação linear dos parâmetros (W,b):

$$z = W\phi(x) + b$$

Podemos então concluir que a função MLL é também ela convexa em relação a (W,b), de acordo com a seguinte propriedade - *the composition of an affine map g:* $R^n \Rightarrow R^m$ *with a convex function* $f : R^m \Rightarrow R$ *is convex.* - disponível no enunciado desta questão.

Relativamente aos casos em que z não é uma combinação linear de (W,b), a concavidade da função de perdas da MLL, neste caso, não é garantida.

Questão 2

Alínea 2.1 - Erro Quadrático Médio

Considerando agora a função de erro quadrático médio definida abaixo, queremos provar a sua concavidade em relação a (W,b).

$$L(z;y) = \frac{1}{2}(z-y)^T(z-y)$$

Assumindo $e = z - y$, temos:

$$L(e) = \frac{1}{2}e^2$$

Podemos por isso fazer:

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial e} \cdot \frac{\partial e}{\partial z} = e \cdot 1 = e$$

Repondo as variáveis originais temos que:

$$\frac{\partial L}{\partial z} = z - y$$

Agora para a segunda derivada:

$$\frac{\partial}{\partial z}\left(\frac{\partial L}{\partial z}\right) = \frac{\partial}{\partial z}(z-y) = 1 > 0$$

Provamos assim que a função do erro quadrático ($L(z;y)$) é convexa em função de z. Contudo, pela prova realizada na questão anterior podemos também afirmar que $L(z;y)$ é convexa em relação a (W,b), dado que z é uma combinação linear destes parâmetros.

Alínea 2.2 - *Ames Housing*

Nesta questão, o objectivo é utilizar ferramentas de regressão linear para estimar o valor de centenas de imóveis, baseado num conjunto de *features* disponibilizados através do conjunto de dados *Ames Housing*.

Alínea 2.2 a - Modelo Linear

Começámos por definir a solução fechada para o problema de regressão, implementando o seguinte par de equações.

$$\hat{y} = wX + b \tag{6}$$

onde os pesos w, podem ser estimados analiticamente, aplicando:

$$w = (X^T X)^{-1} X^T y \tag{7}$$

Adicionalmente, um termo de regularização foi adicionado para garantir a existência de matriz inversa.

De seguida completou-se um modelo linear. Para o treino, considerou-se o erro quadrático médio como *loss function* a optimizar ao longo de 150 épocas, utilizando *Stochastic Gradient Descent* (SGD) e um *learning rate* de 10^{-3}. Os resultados obtidos para o conjunto de treino e teste estão presentes na Figura 1.

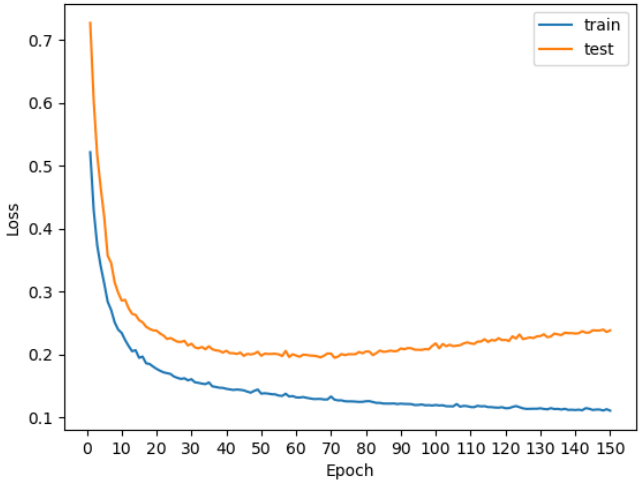

Figura 1: *A evolução da função de perdas (Loss) para um modelo linear, ao longo de 150 épocas.*

É possível observar que as perdas para a função de treino são sempre inferiores às de teste. Este resultado é expectável dado que os dados de treino são usados para a rede aprender, havendo assim uma adaptabilidade natural a estes dados. Lembrando que o objectivo do algoritmo de regressão linear é reduzir as perdas para o conjunto de treino, verificamos que isso acontece monotonamente. Contudo, o mesmo não acontece para o conjunto de teste. Podemos por isso dizer que se verifica a existência do fenómeno conhecido como *overfitting*, em que o modelo treinado se adapta em demasia aos dados que lhe foram dados para treinar, ficando menos capaz de representar dados não vistos anteriormente. Apesar de haver diversas formas de mitigar este problema, como o *Early Stopping* entre outros, ficam fora do espectro deste relatório. No final do treino, o valor das perdas para o conjunto de treino e de teste foram, respectivamente, 0.111 e 0.238.

Por último, averiguou-se a evolução da "distância" entre os pesos obtidos através do treino do modelo de regressão linear e os pesos obtidos analiticamente. O resultados desta evolução está expresso na Figura 2.

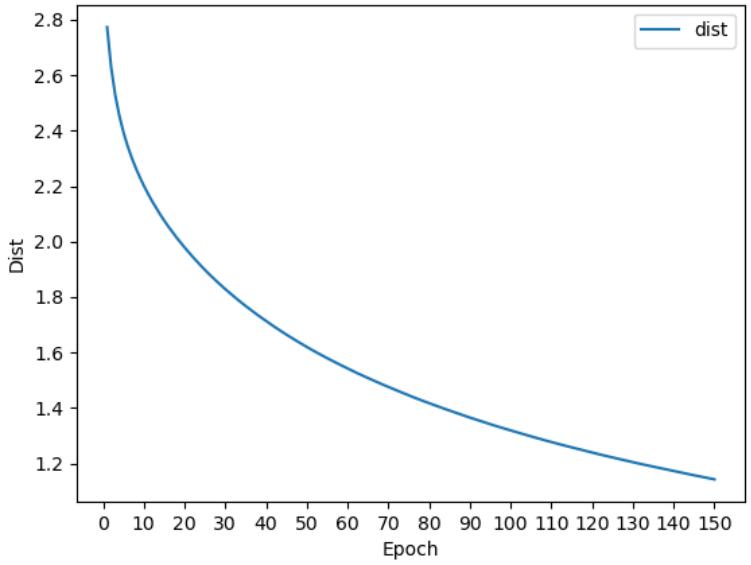

Figura 2: *Distância entre os pesos aprendidos e os obtidos analiticamente ao longo de 150 épocas.*

Podemos observar que a cada iteração/época, o modelo treinado converge para os valores obtidos de forma analítica, como seria de esperar. Adicionalmente, este gráfico ajuda-nos a perceber o aparecimento de *overfitting*, uma vez que o pesos do modelo treinado tendem para os pesos analíticos, estes que estão 100% moldados aos dados de treino.

Alínea 2.2 b - Rede Neuronal

Desta vez, a ferramenta a utilizar é uma rede neuronal, com uma camada escondida de 150 unidades e com uma *ReLU* como função de activação não linear. Também aqui consideramos o erro quadrático médio como função de perdas. Os pesos da rede neuronal foram inicializados de forma a seguir uma distribuição normal, $w_0 \sim N(0.1, 0.1^2)$ e os *bias* nulos . O treino foi feito ao longo de 150 épocas, com um *learning rate* de 10^{-3}, uma vez mais utilizando SGD. Avaliamos a evolução da função de perda para os conjuntos de treino e teste, cujos resultados estão presentes na Figura 3.

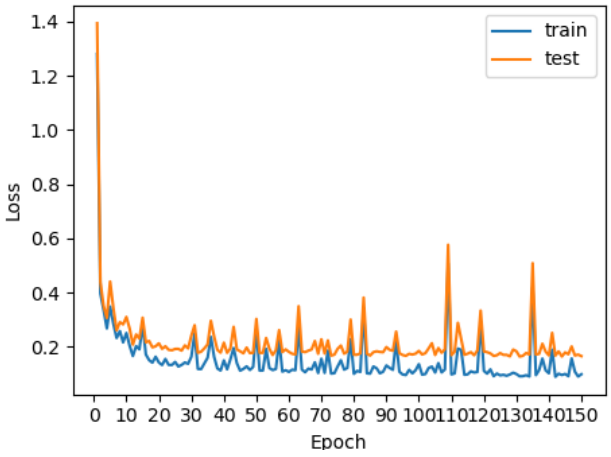

Figura 3: *A evolução da função de perdas (Loss) para a rede neuronal, ao longo de 150 épocas*

Podemos observar, um valor de perdas inicial muito elevado, isto dá-se devido á inicialização aleatória dos pesos da rede. Rapidamente, com a passagem das épocas, os pesos convergem, reduzindo significativamente o valor das perdas. No final do treino, o valor das perdas para o conjunto de treino e de teste foram, respectivamente, 0.099 e 0.147. Em comparação com o modelo linear, a rede neuronal prova convergir muito mais rapidamente, atingindo melhores resultados para ambos os conjuntos de treino e teste.

É de se notar que as fortes variações da função de perda se devem ao SGD, com *batch size* 1. O uso deste *optimizer* com um *batch size* tão pequeno torna a função de perdas muito susceptível às flutuações dos dados que entram no modelo, por serem outliers ou por possuírem padrões pouco habituais, por exemplo. Uma maneira de diminuir o efeito destas variações seria aumentar o *batch size*. No caso extremo, usando um *Full Batch Gradient Descent*, o efeito destas flutuações seria imperceptível, porque os dados entrariam todos em conjunto no modelo, no mesmo *batch*.

Questão 3

Neste exercício é pedido que implementemos um classificador linear de imagens do *Fashion-MNIST dataset*.

Alínea 3.1

Alínea 3.1.a - *Perceptron*

Nesta alínea, devemos implementar um único *perceptron* para resolver o problema, tendo iniciado o mesmo com a implementação da função *update_weights* fornecida no código base. Esta função segue as regras de classificação para várias classes (multi-classe) para um *perceptron*, em que o peso é aumentado quando a predição é correta e diminuído quando é incorreta. Após 20 épocas de treino, o *perceptron* alcança uma precisão de cerca de 70% para o *set* de teste (o valor mais baixo das 20 épocas), tendo alcançado o valor máximo de cerca de 82% na 13ª época, sendo o mesmo observável na Figura 4.

```
y_hat = np.argmax(self.W.dot(x_i))
if y_hat != y_i:
    # Perceptron update.
    self.W[y_i, :] += x_i # Right Class
    self.W[y_hat, :] -= x_i # Wrong Class
return
```

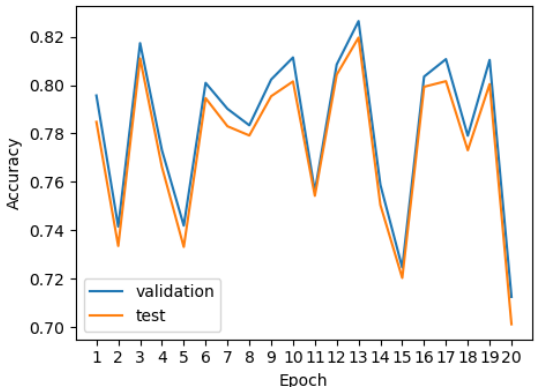

Figura 4: *Precisão do perceptron ao longo de 20 épocas, para os sets de validação (a azul) e de teste (a laranja)*

Alínea 3.1.b - Regressão Logística

Nesta alínea, devemos repetir o mesmo exercício que na alínea anterior, utilizando a regressão logística sem regularização e o *stochastic gradient descent* como algoritmo de optimização, com um *learning rate* de 0.001.

Através da observação da Figura 5 podemos constatar que a precisão aumentou gradualmente ao longo das épocas, iniciando com um valor de cerca de 81.3% e terminando com cerca de 84%.

Podemos concluir que houve uma melhoria significativa nas previsões, uma vez que na alínea anterior o valor oscila constantemente enquanto na presenta alínea a previsão cresce em sentido lato ao longo do tempo.

```
        # Multi-class Logistic Regression
        label_scores = self.W.dot(x_i)[:, None] # Label scores according to the model (num_labels
            x 1).
        y_one_hot = np.zeros((np.size(self.W, 0), 1)) # One-hot vector with the true label
            (num_labels x 1).
        y_one_hot[y_i] = 1
        label_probabilities = np.exp(label_scores) / np.sum(np.exp(label_scores))# Softmax
            function.
        self.W += learning_rate * (y_one_hot - label_probabilities) * x_i[None, :] # SGD update
        return
```

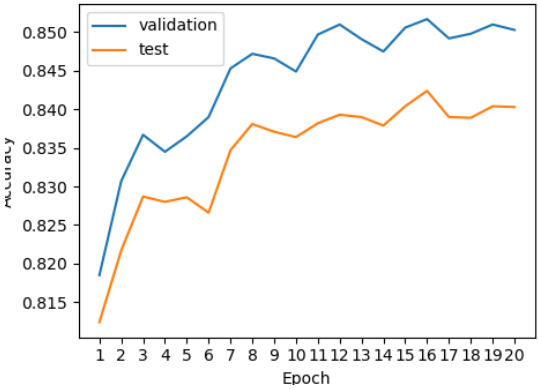

Figura 5: *Precisão do regressão logística ao longo de 20 épocas, para os sets de validação (a azul) e de teste (a laranja)*

Alínea 3.2

Alínea 3.2.a - Camadas lineares vs camadas não-lineares

O uso de múltiplas camadas com função de ativação não-linear permite uma melhor adaptação ao *dataset*, especialmente se este não for linearmente separável, uma vez que as funções lineares não são desejáveis para este tipo de *datasets*, ou seja, as funções não-lineares estabelecem relações mais ajustadas com os dados, devido à sua maior complexidade.

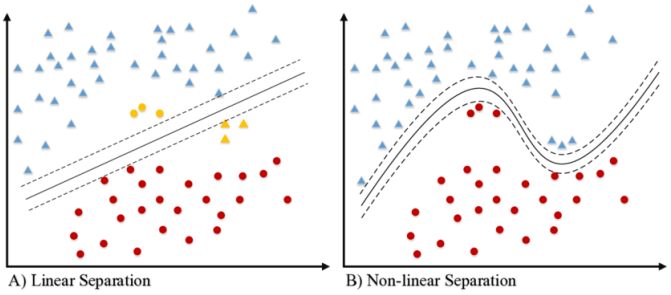

Figura 6: *Separação de um dataset linearmente inseparável (1)*

Uma vez que as funções lineares têm derivada constante, esta não apresenta qualquer relação com a entrada, impedindo de retroceder e entender que pesos nos neurónios de entrada podem fornecer uma melhor previsão. Assim, funções lineares não podem aumentar a sua capacidade de computação através do uso de múltiplas camadas nem aplicar retropropagação, apresentado-se com as duas principais vantagens das funções de ativação não-lineares face às lineares (para além da melhor adaptação a *datasets* não separáveis linearmente).

A combinação de múltiplas camadas lineares tem o mesmo efeito que o uso de uma única camada linear, uma vez que o aumento do número de camadas apenas aumenta o poder de expressão do algoritmo quando as ditas camadas não são lineares (por exemplo, tangente hiperbólica, logística). Em suma, uma camada com activação linear tem o mesmo efeito que múltiplas camadas com activação linear, como pode ser constatado nas imagens abaixo.

Figura 7: *Redes neuronais com 1 (cima) e 3 (baixo) hidden layers lineares.*

Alínea 3.2.b - *MLP*

Nesta alínea é pedido que se implemente uma *multi-layer perceptron* (MLP) com uma hidden layer usando o *gradient backpropagation* como algoritmo de treino. A função de ativação usada é a *Rectified Linear Unit* (ReLU), são usados também 200 *hidden units* e a função de perda usada é a *cross-entropy*. Os pesos e os *bias* são iniciados usando uma curva de Gauss com média e variância iguais a 0,1.

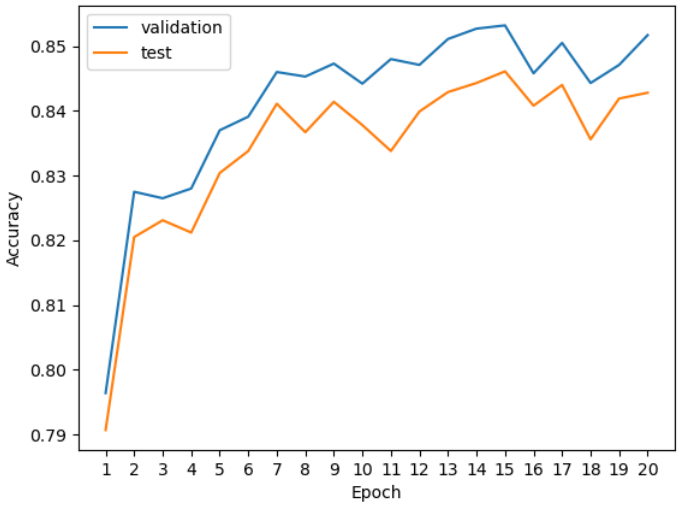

Figura 8: *Precisão da MLP ao longo de 20 épocas, para os sets de validação (a azul) e de teste (a laranja)*

Questão 4

Nesta alínea era pedida a implementação de modelos de aprendizagem profunda com *backpropagation* de gradientes, à semelhança do pedido na questão 3, usando um *framework* com diferenciação automática. Para resolver estas questões usou-se o *framework Pytorch* e o *skeleton code* fornecido no ficheiro hw1-q4.py.

Alínea 4.1 - Modelo Linear com Regressão Logística

Primeiro era pedida a implementação de um modelo linear com regressão logística, mais concretamente dos métodos **train_batch()**, **__init__()** (da classe **LogisticRegression**) e **forward()** (também da classe **LogisticRegression**) presentes no ficheiro h1-q4.py. A sua implementação pode ser visualizada de seguida.

```
class LogisticRegression(nn.Module):
    def __init__(self, n_classes, n_features, **kwargs):
        super().__init__()
        self.layer = nn.Linear(n_features,n_classes)
        return
    def forward(self, x, **kwargs):
        y = self.layer(x)
        return y
def train_batch(X, y, model, optimizer, criterion, **kwargs):
    # clear the gradients
    optimizer.zero_grad()
    # compute the model output
    yhat = model(X)
    # calculate loss
    loss = criterion(yhat, y)
    # credit assignment
```

```
        loss.backward()
        # update model weights
        optimizer.step()
        # return the numeric value of loss
        return loss.item()
```

De modo a avaliar o desempenho do modelo, era pedido para treiná-lo com os *learning rates* 0.001, 0.01 e 0.1. Os resultados podem ser visualizados na Tabela 1.

Tabela 1: *Testing accuracies registadas no Modelo Linear com Regressão Logística para diferentes Learning Rates.*

Learning Rate	0.1	0.01	0.001
Testing Accuracy	79.59%	83.33%	83.38%

De entre os learning rates propostos, concluiu-se que a melhor configuração para o modelo é aquela que foi treinada com *learning rate* de 0.001, por ter a maior *testing accuracy*, de 83.38%, em comparação com as restantes. A evolução da *validation accuracy* e *testing loss* para o modelo com maior *testing accuracy* pode ser encontrada nas Figuras 9 e 10, respectivamente.

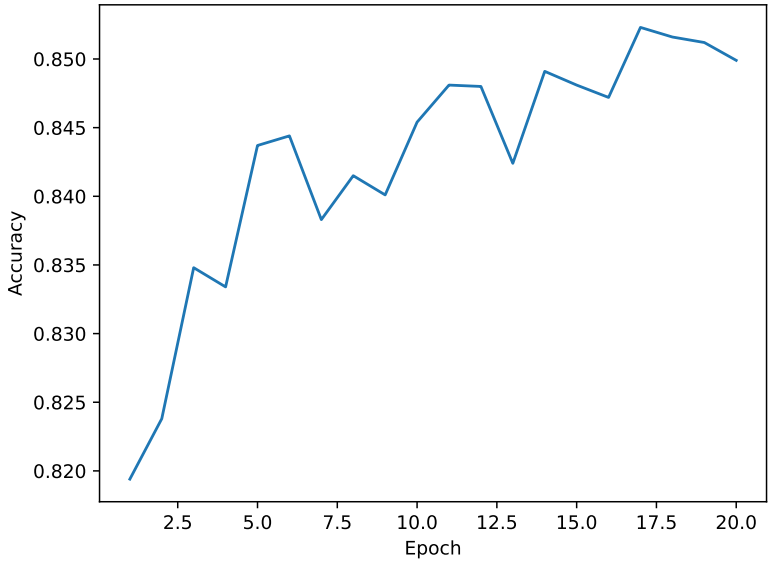

Figura 9: *Evolução da validation accuracy do Modelo Linear com Regressão Logística ao longo de 20 épocas.*

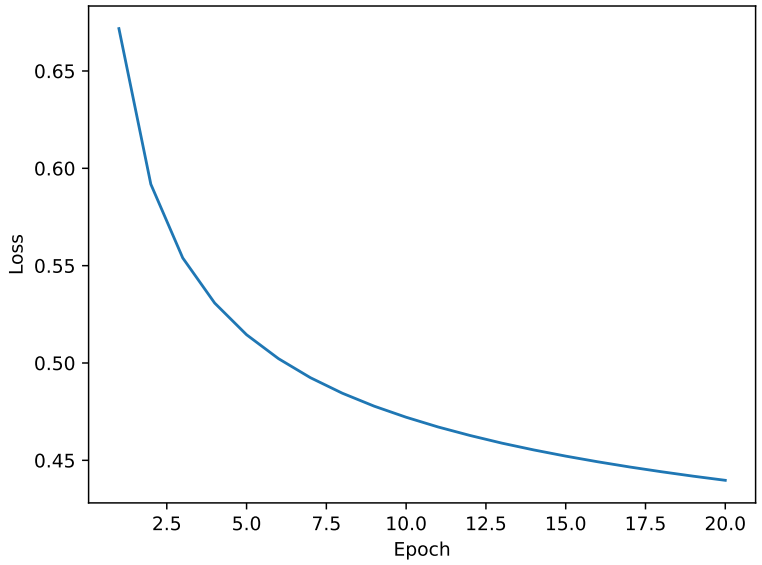

Figura 10: *Evolução da training loss do Modelo Linear com Regressão Logística ao longo de 20 épocas.*

A existência de um *learning rate* menor permite um ajuste mais pequeno do valor dos pesos a cada época, levando a um modelo com previsões mais precisas, com o custo de demorar mais épocas para convergir. No entanto, caso se defina um *learning rate* muito pequeno, corre-se o risco de fazer com que o modelo convirja para um mínimo local da *loss function* usada que não seja o mínimo global da função, ficando preso no mesmo, ou então também pode nem chegar de todo a convergir. Dito isto, tal não se verificou para o *learning rate* escolhido de 0.001. Para o caso do *learning rate* 0.1 registou-se uma convergência mais rápida, num número mais reduzido de épocas, mas para um valor não ideal o que se traduziu num pior desempenho, como se pôde observar na Tabela 1. Esta relação entre o *learning rate* e a convergência do modelo encontra-se descrita na Figura 11.

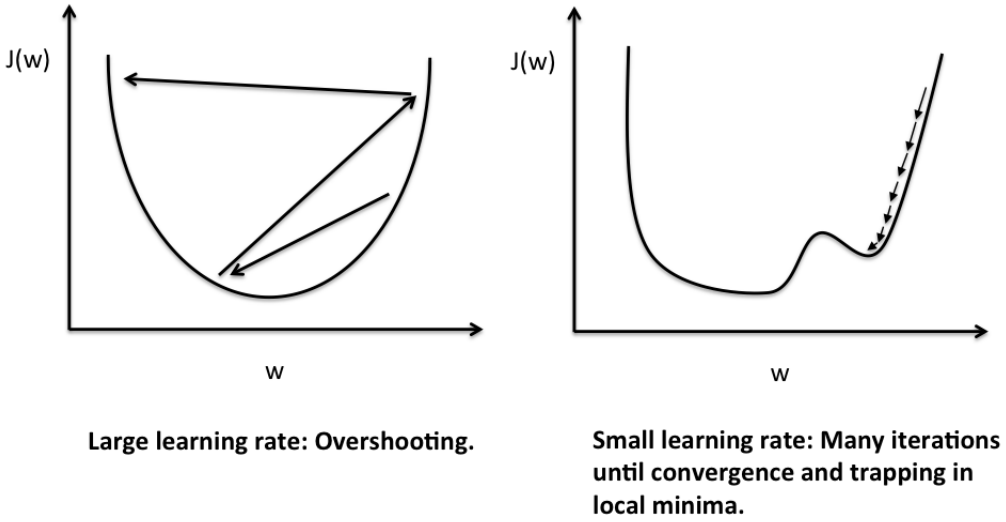

Figura 11: *Relação do learning rate de um modelo com a sua convergência para um mínimo da função de perda (2).*

Alínea 4.2 - Rede Neuronal *Feed-Foward* de 1 Camada

Na segunda alínea era pedido para se implementar uma rede neuronal *feed-foward* com uma única camada (mesma da Questão 3) usando regularização por *dropout*. Para construir o modelo pretendido, modificaram-se os métodos **__init__()** e **foward()** da classe **FeedfowardNetwork**. O código desenvolvido pode ser visto abaixo.

```
class FeedforwardNetwork(nn.Module):
    def __init__(
            self, n_classes, n_features, hidden_size, layers,
            activation_type, dropout, **kwargs):

        super(FeedforwardNetwork, self).__init__()

        if (activation_type == "relu"):
            activation_func = nn.ReLU()
        else :
            activation_func = nn.Tanh()

        if (layers == 2):
            self.layers = nn.Sequential(
                nn.Linear(n_features, hidden_size),
                nn.Dropout(p=dropout),
                activation_func,
                nn.Linear(hidden_size, hidden_size),
                nn.Dropout(p=dropout),
                activation_func,
                nn.Linear(hidden_size, n_classes),
            )
        elif (layers == 3):
            self.layers = nn.Sequential(
                nn.Linear(n_features, hidden_size),
                nn.Dropout(p=dropout),
                activation_func,
                nn.Linear(hidden_size, hidden_size),
                nn.Dropout(p=dropout),
                activation_func,
                nn.Linear(hidden_size, hidden_size),
                nn.Dropout(p=dropout),
                activation_func,
                nn.Linear(hidden_size, n_classes),
            )
        else:
            self.layers = nn.Sequential(
                nn.Linear(n_features, hidden_size),
                nn.Dropout(p=dropout),
                activation_func,
                nn.Linear(hidden_size, n_classes),
            )

    def forward(self, x, **kwargs):
        output = self.layers(x)
        return output
```

Era pedido para se avaliar o desempenho desta rede neuronal para diferentes configurações de *hyperparameters*, nomeadamente no que toca ao tamanho das *hidden layers*, função de ativação usada ou *learning rate* por exemplo, de

modo a se obter a melhor configuração possível, utilizando como critério a precisão de teste. Os resultados obtidos para cada configuração podem ser encontrados na Tabela 2.

Tabela 2: *Testing accuracies registadas na Rede Neuronal para diferentes configurações de hyperparameters.*

Hyperparameter						Testing Accuracy
Learning Rate	**Hidden Size**	**Dropout**	**Batch Size**	**Activation Function**	**Optimizer**	
0.1	200	0.3	1	ReLU	SGD	12.41%
0.01	200	0.3	1	ReLU	SGD	86.62%
0.001	200	0.3	1	ReLU	SGD	88.02%
0.01	100	0.3	1	ReLU	SGD	85.99%
0.01	200	0.5	1	ReLU	SGD	85.28%
0.01	200	0.3	1	tanh	SGD	85.41%
0.01	200	0.3	1	ReLU	Adam	39.80%

Como se pode observar pela Tabela 2, a configuração com melhor desempenho corresponde à configuração com o *learning rate* a 0.001, onde se registou uma *testing accuracy* de 88.02%. Comparando este valor, com o registado na questão 4.1., verifica-se o aumento de desempenho no uso de Redes Neuronais, ao invés de Modelos Lineares. Esta diferença de desempenho resulta da maleabilidade das Redes Neuronais, que conseguem estabelecer relações não lineares nos preditores, tendo como custo o maior número de parâmetros a estimar e a necessidade de um maior conjunto de dados de treino para chegar a uma precisão satisfatória.

Observando a variação da *testing accuracy* com a mudança do tamanho da *hidden layer*, conclui-se que a precisão do modelo é melhor quando este usa uma *hidden layer* maior, pelo menos para este problema de classificação de imagem em questão. Caso se usem poucas *hidden units*, o modelo pode ficar *underfitted*, pois não existem neurónios/unidades suficientes para representarem os padrões dos dados, resultando num modelo muito generalista e com pouca precisão. Um aumento no número de *hidden units*, permite enriquecer o preditor do modelo, levando a um aumento da sua precisão, tal como se verificou. No entanto, caso se aumente em demasia o *hidden size*, corre-se o risco de a rede neural se especializar em demasia nas características do conjunto de dados de treino, causando *overfitting* e levando a um aumento da precisão de treino e a um decréscimo elevado da precisão de teste. Deste modo é sempre importante estudar o tipo de dados que vão ser aplicados sobre o modelo ao se definir o *hidden size*, e qualquer outro *hyperparameter*.

A Tabela 2 também demonstra que, um aumento do *dropout* de 0.3 para 0.5, resulta num decréscimo da precisão do modelo. O *dropout* corresponde à percentagem de neurónios que são desligados aleatoriamente durante cada iteração do treino do modelo. Deste modo, para estes dados, um *dropout* mais elevado corresponde a remover mais neurónios durante o treino do modelo e, como tal, a generalizar mais o modelo, aproximando-o da situação de *underfitting*, por *overregularization*. Para esta configuração o uso de um *dropout* de 0.5 acaba por causar *underfitting* no modelo, no entanto, caso se tratasse de um modelo com mais camadas, esse valor já poderia ser mais apropriado.

No que toca a funções de ativação, a *ReLU* provou ser mais adequada comparativamente com a *tanh* para este *dataset*. Isto pode ter sido devido à função de ativação *tanh* ser mais complexa computacionalmente e de sofrer de *vanishing gradients* na *backpropagation*, fruto do facto de a *tanh* restringir o seu *output* ao intervalo [-1;1], o que torna a aprendizagem mais lenta do que no caso da *ReLU*.

Por fim, apesar de o ADAM ser um *optimizer* com uma convergência muito mais rápida face ao SGD, o SGD generaliza melhor que o ADAM, resultando por isso numa melhor precisão. Para além disso, de acordo com (3), o SGD é um *optimizer* uniformemente estável para *loss functions* convexas, como a *Cross-Entropy loss function*, o que resulta numa redução óptima do erro de generalização.

A evolução da *training loss* e *validation accuracy* da configuração considerada ótima (*learning rate* de 0.001) pode ser encontrada na Figura 12 e Figura 13, respetivamente.

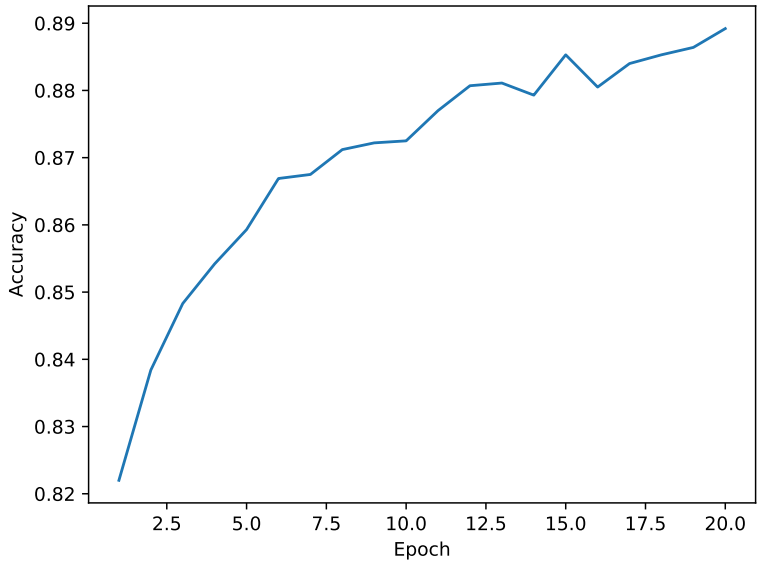

Figura 12: *Evolução da validation accuracy do Rede Neuronal ideal ao longo de 20 épocas.*

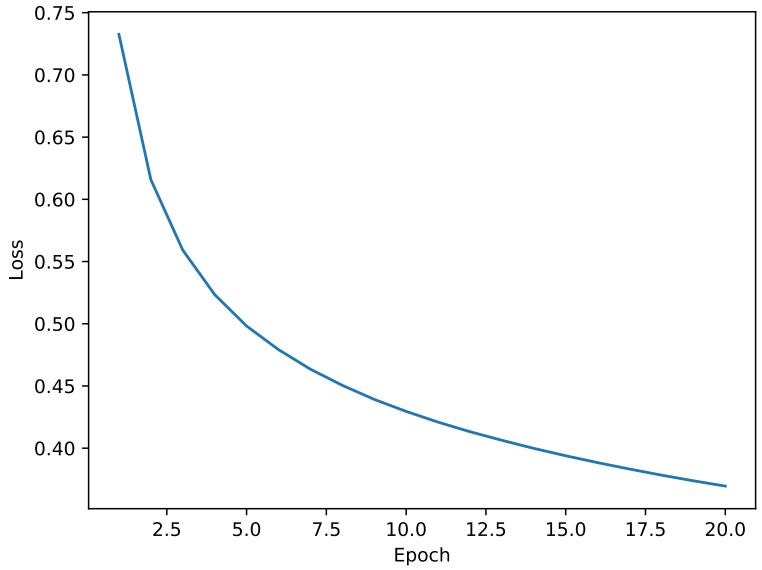

Figura 13: *Evolução da training loss da Rede Neuronal ideal ao longo de 20 épocas.*

Conclui-se portanto que, de todos os *hyperparameters* que se podem afinar, o *learning rate* é o que mais impacto tem na precisão do modelo.

Alínea 4.3 - Rede Neuronal *Feed-Foward* de 2 e 3 Camadas

Na terceira alínea da questão 4, era pedido para se analizar a precisão do modelo da alínea anterior, considerando os *hyperparameters* padrão, e fazendo variar o número de *hidden layers*, de modo a retirar conclusões sobre a influência que o número de camadas tem no desempenho da rede neuronal. Usando o código apresentado na alínea anterior, chegaram-se às seguintes *testing accuracies* para as diferentes configurações (Tabela 3):

Tabela 3: *Testing accuracies registadas na Rede Neuronal para diferentes números de hidden layers.*

Hyperparameters							Testing Accuracy
Hidden Layers	**Learning Rate**	**Hid. Size**	**Dropout**	**Batch Size**	**Act. Func.**	**Opt.**	
2	0.01	200	0.3	1	ReLU	SGD	86.39%
3	0.01	200	0.3	1	ReLU	SGD	85.94%

Como se pode observar na Tabela 3, considerando os *hyperparamaters* padrão, registou-se para a configuração do modelo com duas *hidden layers* a maior *testing accuracy*, de 86.39%, sendo por isso esta a configuração ideal. Este comportamento pode ser justificado tendo em consideração que a cada camada adicionada ao modelo são adicionados mais neurónios, permitindo retirar mais *features* dos dados de treino. Isto torna o modelo menos generalista, e corre-se o risco de este entrar em *overfitting*, resultando num decréscimo da *testing accuracy*. Isto justifica a perda de precisão com a introdução de uma terceira *hidden layer*. Eventualmente, esta perda de precisão com o aumento do número de camadas poderia ter sido corrigida com o afinamento de outros *hyperparameters*, nomeadamente do *dropout*.

Na Figura 14 e na Figura 15 pode ser visualizada a evolução com o decorrer das épocas da *validation accuracy* e da *training loss* da configuração ideal (2 *hidden layers*), respectivamente.

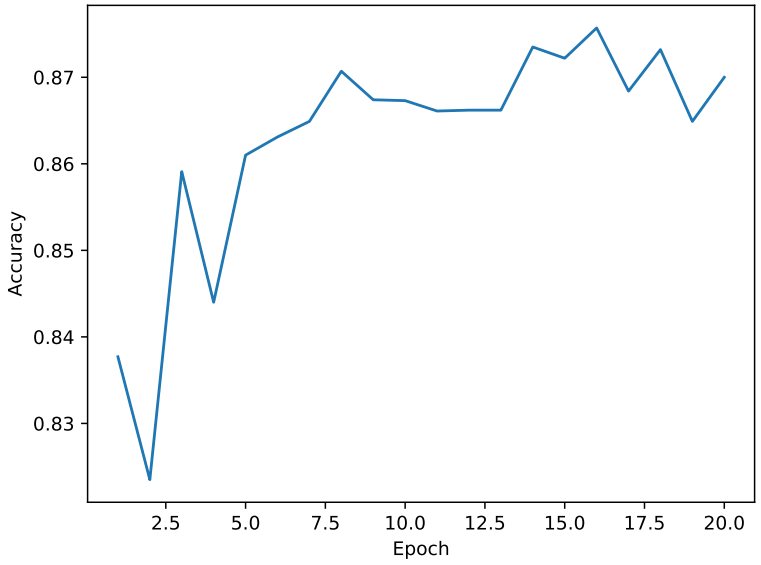

Figura 14: *Evolução da validation accuracy do Rede Neuronal ideal com duas hidden layer2 ao longo de 20 épocas.*

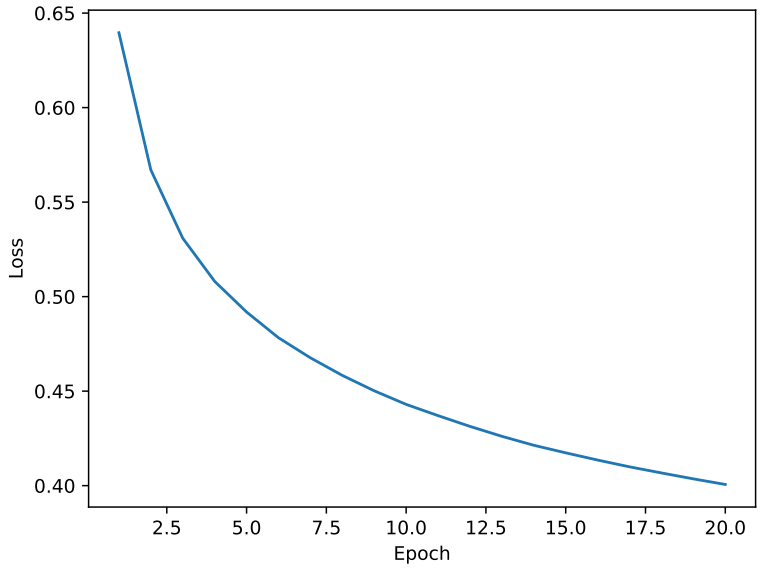

Figura 15: *Evolução da training loss da Rede Neuronal ideal com duas hidden layers ao longo de 20 épocas.*

Agradecimentos

Aos grupos 13, 39 e 40, com os quais se discutiram resultados de modo a confirmar o correto funcionamento das implementações do nosso grupo, assim como a todas as dúvidas respondidas no *Piazza*, as quais guiaram o nosso trabalho.

Bibliografia

[1] Kowsari, K. (2020, abril) "Diagnosis and Analysis of Celiac Disease and Environmental Enteropathy on Biopsy Images using Deep Learning Approaches"

[2] Single Layer Neural Networks - Perceptron Learning Rate: https://sebastianraschka.com/images/blog/2015/singlelayer_neural_networks_files/perceptron_learning_rate.png

[3] Hardt, M., Recht, B., & Singer, Y. (2016, junho). "Train faster, generalize better: Stability of stochastic gradient descent". In International Conference on Machine Learning (pp. 1225–1234). PMLR.