TÉCNICO LISBOA

Aprendizagem Profunda

MEEC

Homework 1

Autores:

André Alexandre Costa Santos - 96152 andresantos1.alf@gmail.com
João Bernardo Vieira Pinto dos Santos - 96237 joaobsantos2001@tecnico.ulisboa.pt

Grupo 3

2022/2023 – 1º Semestre, P2

Conteúdo

1 Introdução

Neste primeiro trabalho de Aprendizagem Profunda, foi proposta a implementação de vários algoritmos de aprendizagem, sendo esta realizada a partir do mais baixo nível de modo a compreender o funcionamento intrínseco dos referidos algoritmos. Foi também proposta a utilização de determinados toolkits de *machine learning* de modo a aprofundar o conhecimento destes.

2 Questão 1

Na primeira questão, foi proposto o desenvolvimento de um *linear classifier* para um problema simples de classificação de imagens presentes num *dataset*, utilizando apenas algebra linear. Foi necessário realizar a implementação de diversos métodos, descritos nas seguintes secções.

Questão 1.1. a) Perceptron

Inicialmente foi implementado o método *Perceptron*. Neste, é realizado o cálculo de $\hat{y}$, utilizando o vetor w (weights), cujo valor é inicializado a zero e uma determinada entrada, neste caso, as imagens. Após o cálculo de ŷ, o valor obtido deste é comparado com o valor esperado (y) e, caso se verifique uma discrepância, o vetor w é atualizado. Esta lógica foi implementada da seguinte maneira:

```
class Perceptron(LinearModel):
    def update_weight(self, x_i, y_i, **kwargs):
        """
        x_i (n_features): a single training example
        y_i (scalar): the gold label for that example
        other arguments are ignored
        """
        y_hat = self.predict(x_i)

        if y_hat != y_i:
            self.W[y_i] +=  x_i
            self.W[y_hat ] -=  x_i
```

Figura 1: Implementação do método *update weight* da classe Perceptron

Como é possível ver através da interpretação do código apresentado acima, quando os valores de $\hat{y}$ e y são diferentes, os valores nas respetivas posições no vetor w são atualizados, utilizando uma soma para o valor real e uma subtração para o valor calculado, com *Xi*, com o objetivo de recompensar o valor correto e punir o valor errado.

Na seguinte figura apresenta-se o resultado obtido:

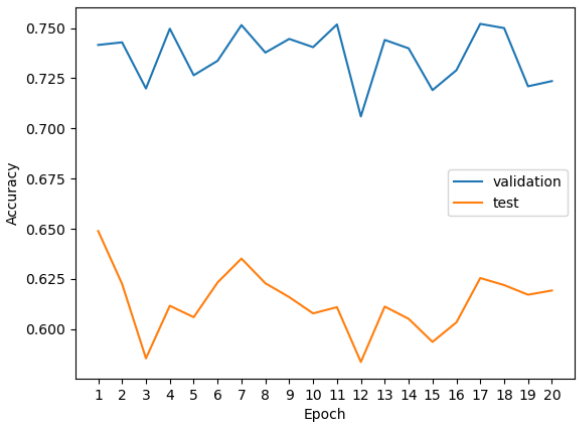

Figura 2: Resultado obtido para a implementação Perceptron

Neste gráfico é possível observar a *Accuracy* quer do conjunto de dados de validação quer do conjunto de dados que foi utilizado para o teste do modelo. Podemos verificar que a *Accuracy* do *train batch* é mais elevada que a do conjunto de teste, uma vez que o modelo foi treinado com estes dados em específico, estando portanto mais ajustado aos referidos. Por outro lado podemos ver que a métrica utilizada na avaliação do Perceptron não melhorou devido à linearidade da função da ativação.

Questão 1.1. b) Regressão Logística

De seguida, foi necessário implementar, com o mesmo objetivo, uma *logistic regression*, utilizando *stochastic gradient descent* como algoritmo de treino. É realizada a multiplicação do vetor w pelo vetor x, obtendo-se assim $\hat{y}$. É criado então um vetor *one-hot*, cujo valor positivo está na posição y. Prossegue-se então para o cálculo da seguinte probabilidade condicional (transformação *softmax*):

$$P(y \mid x) = \frac{\exp\left(w_y^T \phi(x)\right)}{Z_x}, \qquad \text{where } Z_x = \sum_{y' \in y} \exp\left(w_{y'}^T \phi(x)\right) \tag{1}$$

Este valor é então utilizado para atualizar o vetor de *weights* através da seguinte equação:

$$\boldsymbol{W}^{(k+1)} = \boldsymbol{W}^{(k)} + \eta_k \left(e_y \phi(x)^\top - \sum_{y'} P_{\boldsymbol{W}}\left(y' \mid x\right) e_{y'} \phi(x)^\top \right) \tag{2}$$

A regressão logística foi implementada através do seguinte código:

```python
class LogisticRegression(LinearModel):
    def update_weight(self, x_i, y_i, learning_rate=0.001):
        """
        x_i (n_features): a single training example
        y_i: the gold label for that example
        learning_rate (float): keep it at the default value for your plots
        """

        label_scores = self.W.dot(x_i)[:, None]
        y_one_hot = np.zeros((np.size(self.W, 0), 1))
        y_one_hot[y_i] = 1
        label_probabilities = np.exp(label_scores) / np.sum(np.exp(label_scores))
        self.W += learning_rate * (y_one_hot - label_probabilities) * x_i[None, :]
```

Figura 3: Implementação da regressão logística

Resultando então no seguinte resultado:

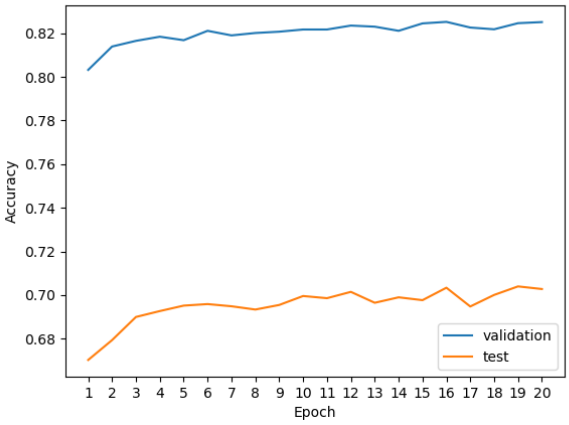

Figura 4: Resultado da regressão logística

Questão 1.2. Multi-layer Perceptron

Finalmente prosseguiu-se então para a implementação do *multi-layer Perceptron*.

1.2. a) Um único neurónio *Perceptron* apenas consegue aprender funções separadas linearmente, ou seja, apenas consegue aprender a classificar dados separados por uma linha reta. De modo a superar esta limitação, deve ser utilizado um *multi-layer Perceptron*, dado que este consegue combinar várias decisões lineares em diferentes modos com o objetivo de aprender funções não lineares. Caso o *multi-layer Perceptron* utilizasse uma função de ativação linear, comportar-se-ia como um único neurónio com uma função de ativação linear, incapaz de aprender funções mais complexas.
Exemplo:

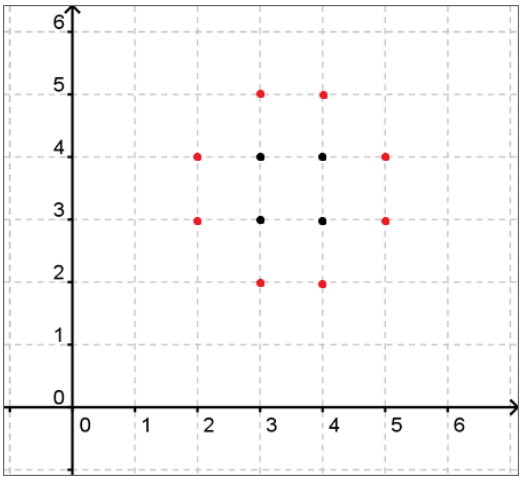

Figura 5: Função a aprender

Neste caso, é necessária a utilização de um *multi-layer Perceptron* para aprender a função representada acima, dado que os dados não podem ser separados por uma reta. Seria necessária a utilização de funções de ativação não lineares de modo a criar uma linha de separação.

1.2. b) Para a implementação da MLP, foi necessário o desenvolvimento de vários métodos na classe em questão (*MLP)*. Foi necessário criar e inicializar dois vetores de pesos e *biases*, um que atuará entre a entrada e a primeira *hidden layer* e um segundo entre a *hidden layer* e o *output* realizando-se de seguida a concatenação destes no método *__init__*. Prosseguiu-se também à implementação da função de ativação *ReLU* (*rectified linear unit*) e da respetiva derivada (*dReLU*). O método definido como *predict* no código dado foi substituido por quatro outros: *forward, backward, predict_label* e *update_parameters*.

Os métodos *forward* e *backward* tratam do cálculo dos forward e backwards pass da rede, respetivamente. Para a função *forward* implementou-se uma série de instruções que consistem em um conjunto de multiplicações matriciais utilizadas para produzir um output da MLP a partir do conjunto de dados fornecido. Inicialmente, os dados de entrada passam pela camada de entrada e multiplicados pelos pesos e *biases* da primeira camada. A saída resultante é passada por uma função de ativação, que produz a saída da primeira camada. A saída da primeira camada é então passada pela segunda camada (oculta), sendo desta vez multiplicada pelos pesos e *biases* dessa camada. A saída da camada oculta passa por outra função de ativação, que produz a saída da segunda camada. Por fim, a saída da segunda camada é passada pela camada de saída e multiplicada pelos pesos e *biases* dessa camada, produzindo a saída final do MLP. O *output* final foi normalizado antes do próximo passo na função *train_epoch*.

Já na função *backward* é calculado o erro entre a saída da MLP e a saída desejada. Este erro é então propagado para trás através da rede, e os pesos e *biases* de cada camada são atualizados de forma a reduzir o erro.

Na função *train_epoch* foi realizada uma conversão do valor do Y recebido para um vetor *one-hot* para auxiliar no cálculo dentro de outras funções.

Obteve-se então o seguinte gráfico:

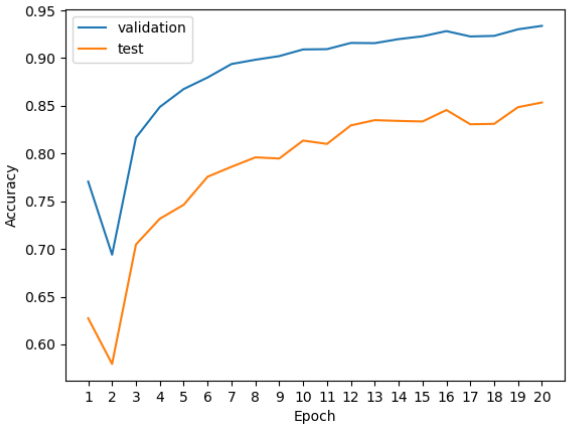

Figura 6: Resultado da MLP

Final test accuracy	Validation Accuracy	Total Loss
85,35%	93,38%	5525,3834

3 Questão 2

Na segunda questão, foi proposta a implementação do mesmo sistema desenvolvido anteriormente, mas desta vez utilizando um framework com derivação automática, algo que facilitou substancialmente o desenvolvimento.

Questão 2.1

```
class LogisticRegression(nn.Module):

    def __init__(self, n_classes, n_features, **kwargs):

        super(LogisticRegression, self).__init__()
        self.layer = nn.Linear(n_features, n_classes)
        #self.activation = nn.Sigmoid()

    def forward(self, x, **kwargs):
        x = self.layer(x)
        x = self.activation(x)
        return x


def train_batch(X, y, model, optimizer, criterion, **kwargs):

    optimizer.zero_grad()
    y_hat = model(X)
    loss = criterion(y_hat, y)
    loss.backward()
    optimizer.step()
    return loss.item()
```

Figura 7: Implementação da regressão logística

Implementou-se então uma regressão logística, na qual foi necessário desenvolver os métodos *train_batch*, *__init__* e *forward*. Na imagem 7 está representada a nossa abordagem.

Após realizar testes para os três *learning rates* pedidos, foi possível concluir que o melhor resultado se obtem para o valor de *0.001*, dado que apresenta a loss final mais baixa, e a accuracy final mais alta. Apresentam-se de seguida os gráficos obtidos para este *learning rate*:

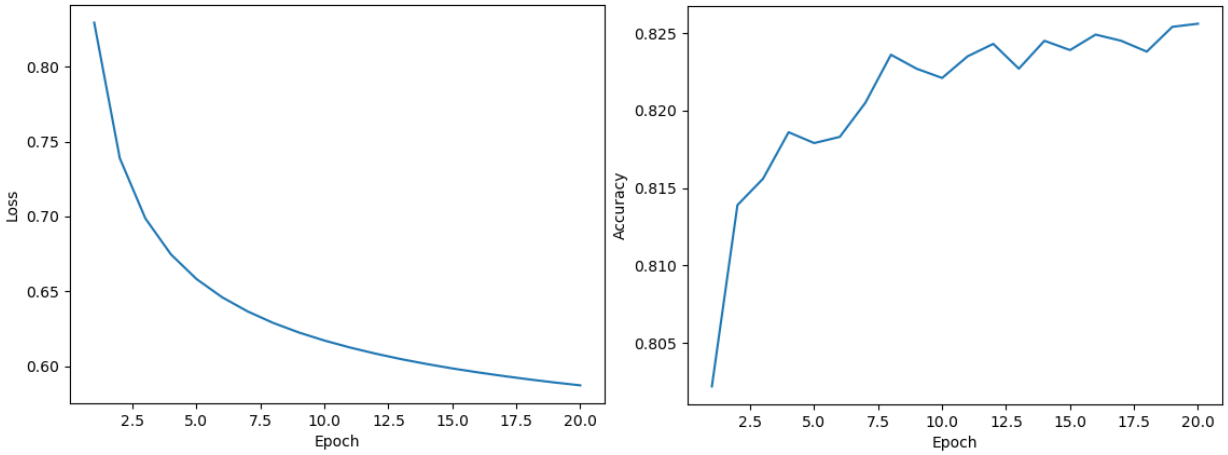

Figura 8: Loss e accuracy para *learning rate* = 0.001

Final test accuracy	Validation Accuracy	Training Loss
64,30%	78,95%	1,7307

Questão 2.2

Realizando-se os testes para os parâmetros pedidos no enunciado, obtiveram-se os seguintes valores de *accuracy* e *loss*:

Parâmetro Alterado	Final Test Accuracy
Learning Rate = 0.001	74,49%
Learning Rate = 0.01	85,88%
Learning Rate = 0.1	87,58%
Hidden Size = 200	88,19%
Dropout Probability = 0.5	84,10%
Activation Function = tanh	82,57%

Verifica-se então a melhor *accuracy* para *Hidden Size 200* e *Learning Rate 0.1*. Os *plots* correspondentes a esta configuração apresentam-se a seguir:

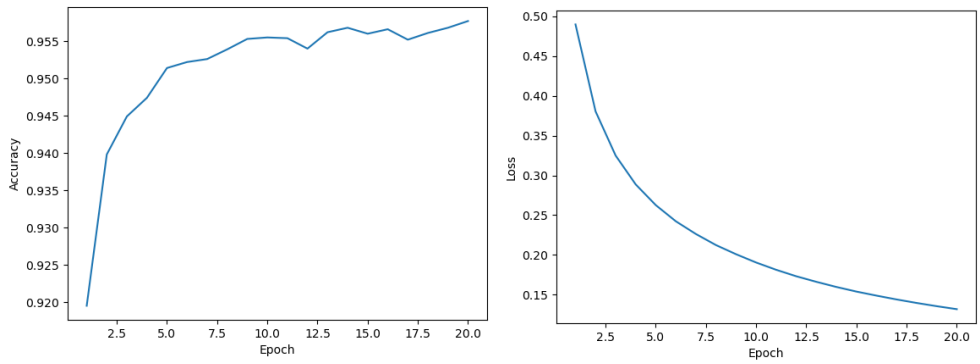

Figura 9: Loss e accuracy para *Hidden Size* = 200

Questão 2.3

Alterando o modelo para utilizar 2 *layers*, obtiveram-se os seguintes resultados:

Parâmetro Alterado	Final Test Accuracy
Learning Rate = 0.001	74,33%
Learning Rate = 0.01	87,11%
Learning Rate = 0.1	87,36%
Hidden Size = 200	89,37%
Dropout Probability = 0.5	84,12%
Activation Function = tanh	82,77%

Conclui-se então que o melhor resultado se obtém para *Hidden Size* = 200 e 0.1 *Learning Rate*, como no caso em que temos apenas um layer. Para esta situação a *Loss* e a *Accuracy* apresentam os seguintes gráficos:

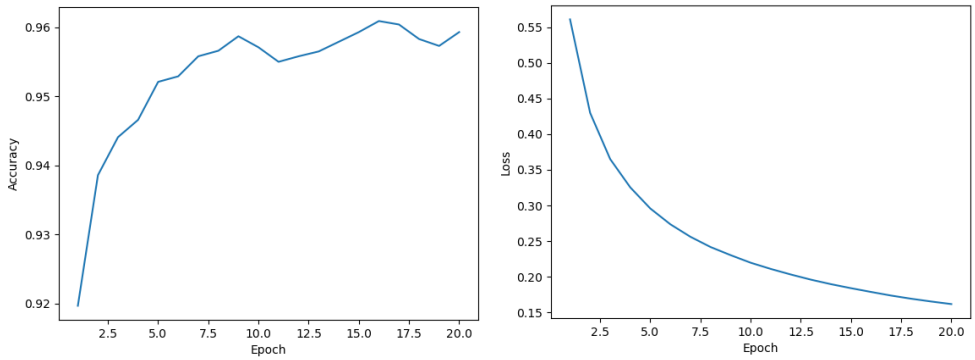

Figura 10: Loss e accuracy para *Hidden Size* = 200, utilizando 2 hidden layers

Repetiram-se os testes para um sistema com três hidden layers, obtendo-se os seguintes valores:

Parâmetro Alterado	Final Test Accuracy
Learning Rate = 0.001	70,55%
Learning Rate = 0.01	86,35%
Learning Rate = 0.1	87,23%
Hidden Size = 200	89,04%
Dropout Probability = 0.5	82,70%
Activation Function = tanh	82,66%

É possível observar que os melhores resultados são, de novo, para o caso com 200 *hidden size* e com 0.1 *Learning Rate*. Os gráficos correspondentes a este caso apresentam-se a seguir:

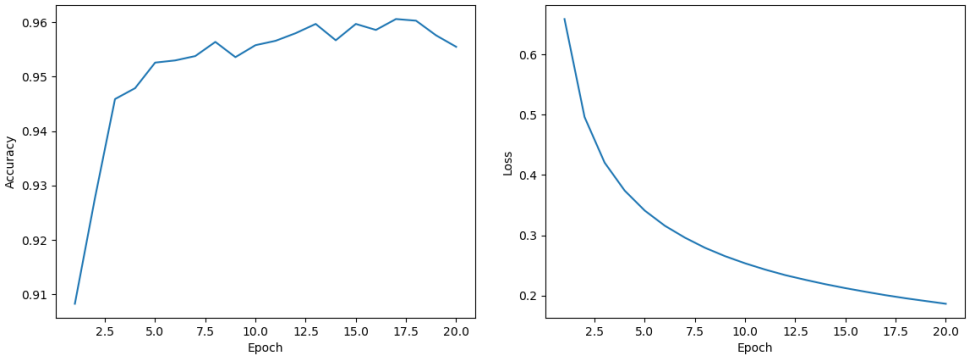

Figura 11: Loss e accuracy para *Hidden Size* = 200, utilizando 3 hidden layers

É possível assim concluir que a melhor configuração possível é utilizando duas *hidden layers*, e com o *hidden size* definido a 200 e 0.1 *Learning Rate*.

4 Questão 3

Questão 3.1

Para mostrarmos que h é uma transformação linear de $\phi(x)$, a matrix A_θ necessita de ser escrita como função de h so a forma enunciada: h= $A_\theta \times \phi(x)$. Para tal expressamos h em função de W e x, sendo que $\phi(x)$ é a transformada de h.

$$h = g(Wx) = (Wx)^2 \implies \phi(x) = [x, x^2] \tag{3}$$

De seguida foi feita a combinação linear dos elementos deste mesmo vetor criando assim a matriz A_θ

$$h = [x, x^2] \times [W, W^2]^T \implies A_\theta = [W, W^2] \tag{4}$$

Este resultado é útil porque permite tratar a rede neural com ativações quadráticas como um modelo linear. Isto é possível porque a saída $\hat{y}$ é dada por $\hat{y} = v^T$ h, e h é uma transformação linear de $\phi(x)$. Portanto, $\hat{y}$ também é uma transformação linear de $\phi(x)$, o que significa que a rede neural pode ser tratada como um modelo linear.

Questão 3.2

Uma vez que o número de perceptrons é diferente do número de x, ou seja, o número de classes, então é necessário extender o tamanho das matrizes, ou seja, adicionar uma coluna de zeros a A_θ. Uma vez que h é uma combinação linear de $\phi(x)$, podemos afirmar o seguinte:

$$h = A_\theta \times \phi(x) = [W^2, W, 0][x^2, 2x^2, 1] = W^2x^2 + Wx^2 + 0 \tag{5}$$

substituindo h por $\hat{y}$:

$$\hat{y} = \hat{y}^T \times h = \hat{y}^T[W^2x^2 + Wx^2 + 0] \tag{6}$$

Assim podemos confirmar que $\hat{y}$ também é uma combinação linear de $\phi(x)$ portanto:

$$\hat{y} = c_\theta \times \phi(x) \tag{7}$$

Sendo c_θ é um vetor de coeficientes dado por $c_\theta = v^T[W^2, W, 0]$.

No entanto, isso não significa necessariamente que este seja um modelo linear em termos dos parâmetros originais θ. Embora $\hat{y}$ seja uma função linear de $\phi(x)$, não é necessariamente uma função linear de θ. Isso ocorre porque os coeficientes c_θ dependem dos valores de W e v, que são funções de θ.

Questão 3.3

Para mostrar que existe uma escolha de parâmetros $\theta = (W, v)$ tal que $c_\theta = c$, podemos recorrer à seguinte demonstração:

Tomando W como uma matriz D x D e v como um vetor D x 1, podemos definir:

$$c_\theta = \begin{pmatrix} W & v \end{pmatrix} \tag{8}$$

$$\theta = \begin{pmatrix} W & v \end{pmatrix} \tag{9}$$

De onde é fácil concluir que, para cada c_θ, é possível encontrar uma parametrização equivalente com θ.

Neste caso, o modelo apresenta-se linear em c_θ caso também o seja em θ.

Se K ¡ D, pode não ser possível encontrar uma parametrização tal que $c_\theta = \theta$. Isto deve-se ao facto de que o número de parâmetros em θ, ou seja, o número de elementos em W e v, é *D(D+1)/2*, que é um valor inferior ao número de elementos em c. Como tal, não existem parâmetros suficientes em $c\theta$.

Questão 3.4

Neste caso é possível encontrar o valor de c_θ, já que a *loss function* é função quadrática de c_θ, ou seja, tem um mínimo absoluto que pode ser encontrado igualando a derivada da *loss function* a zero e resolvendo em ordem a c_θ.

Este caso é especial já que a *loss function* é convexa, algo que geralmente não ocorre. Nestes casos mais comuns, é impossível encontrar o minimo absoluto, sendo assim impossível encontrar o valor de c_θ.

5 Conclusão

Em suma, foi possível, ao longo do desenvolvimento das várias tarefas propostas, aprofundar o conhecimento dos algoritmos abordados (*Perceptron*, *regressão linear* e *multilayer Perceptron*), bem como desenvolver conhecimentos teóricos sobre o funcionamento destes. Surgiram algumas questões durante o desenvolvimento, que foram prontamente respondidas através da consulta dos *slides* das aulas teóricas.

6 Contribuição dos elementos do grupo

Cada um dos elementos do grupo apresentou uma prestação equalitária, estando ambos os membros presentes tanto no desnvolvimento do código como na composição do relatório. As alíneas da questão 3 foram divididas pelos membros do grupo, ficando cada membro encarregue de duas alíneas.