Universidade do Minho
Mestrado em Engenharia Informática
Otimização em Machine Learning

Micro Projeto T4 - Shallow Logistic Classifiers

Lucas Mello - PG40158
Gonçalo Almeida - A84610
Nuno Pereira - PG42846
Artur Ribeiro - A82516
Pedro Ribeiro - PG42848
Gonçalo Costeira - A79799

Mestrado em Engenharia Informática
2021

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

No âmbito da unidade curricular de Otimização em Machine Learning, no qual está enquadrado este Micro-Projeto, foi elaborado um estudo sobre a introdução de apenas uma camada oculta numa rede neuronal associada a uma função de ativação não linear com o objetivo de perceber a hierarquização dos dados e a capacidade de melhorar a predição com dados mais complexos.

Deste modo, o uso de apenas uma ou duas camadas ocultas numa rede neuronal dá-se o nome de *Shallow Neural Network*. Este algoritmo permite-nos perceber o que acontece numa *Deep Neural Network* (DNN), normalmente consituida por várias camadas ocultas, em que neste caso só existe uma ou duas destas camadas onde toda a aprendizagem do modelo acontece, diminuindo a complexidade do problema.

Num algoritmo *Shallow Logistic Classifier* (SLC) os valores das *features* dos dados a serem classificados, recebidos pela *input layer*, são passados aos neurónios da camada oculta. Estes neurónios geram uma resposta de acordo com a função de ativação e sobre o somatório dos pesos atribuídos ao modelo. As respostas geradas pelos neurónios da camada oculta são passados para a última camada, *output layer*, produzindo assim a predição de classificação esperada.

Neste estudo iremos aplicar uma SNN a diferentes bases de dados, explicadas em detalhe no decorrer deste documento, tal como toda a implementação e todas as decisões tomadas pelo do grupo.

1.1 Motivação

A diferença mais destacável entre *Shallow Neural Networks* e *Deep Neural Networks* é a capacidade que as SNN têm de aprender com uma quantidade bastante reduzida de dados quando comparado com DNN, que necessitam de bastantes recursos para obter bons resultados.

Deste modo, existem algumas vantagens consideráveis para o uso de SNN, como o seu baixo custo computacional e de armazenamento, a facilidade e versatilidade da sua aplicação a qualquer problema, um nível elevado de interpretabilidade, a capacidade de processar valores em falta e *features* não relacionadas e a capacidade de lidar com problemas não lineares utilizando poucos dados.

Contudo, a aplicação deste algoritmo também apresenta algumas desvantagens. Como

se trata de uma rede neuronal mais simples e apenas com uma ou duas camadas ocultas, os resultados obtidos podem não ser os esperados, tendo por vezes baixas acurácias quando lidamos com um grande número de tipos variados de dados. Alguns modelos deste género também apresentam problemas de *over* e *underfitting*, alcançando maus resultados quando aplicados a grandes amostras de dados.[**?**]

Em suma, uma SNN é geralmente usada quando temos dados estruturados, de pequena escala, não obtendo bons resultados quando temos um grande volume de dados com tipos variados. Quando queremos lidar com um maior volume de dados não estruturados como processamento de imagens ou vídeos, DNN seria uma solução mais viável.

1.2 Objetivos

Este micro-projeto tem como principal objetivo o estudo da aplicação de uma *Shallow Logistic Layer* para perceber a hierarquização de dados e para melhorar a predição da classificação destes mesmos dados. Deste modo, os objetivos principais deste projeto dividem-se da seguinte forma:

- Implementação do *Shallow Layer Logistic Classifier (SLC)*;
- Implementação da técnica de aprendizagem;
- Experimentação e avaliação da performance do *Shallow Logistic Classifier* em bases de dados com complexidades crescentes;
- Extensão do algoritmo para *Shallow Layer Softmax Classifier*.

Para que fosse possível realizar os objetivos propostos, a equipa docente disponibilizou dois documentos pertinentes como referência, estando estes contidos na bibliografia deste relatório. Por fim, o grupo propõe-se a explicar todo o processo efetuado durante este estudo no presente documento, demonstrando com clareza os resultados obtidos e as conclusões a retirar sobre esses mesmos resultados.

Capítulo 2

Shallow Logistic Classifier

As redes neuronais foram introduzidas pela primeira vez na década de 1960, como um sistema de processamento de informações, e agora são um fator muito importante na aprendizagem estatística. Para exemplo, na aprendizagem supervisionada, estruturas de *Deep Learning* podem realizar tarefas de classificação de imagens com um nível similar ou em muitos casos superior aos humanos. A chave para uma rede neuronal são os pesos que conectam as diferentes camadas, e que transformam as entradas em saídas desejadas. Embora o grande sucesso deva ser creditado a muitas técnicas aplicadas às redes, como estruturas profundas, camadas convolucionais, etc., uma parte fundamental é o método do gradiente.

Neste microprojeto focamos em analisar o *Shallow Logistic Classifier*. Neste classificador o método do gradiente é aplicado a apenas uma camada oculta, e os J neurônios na camada oculta são ativados pela função logística *sigmoid*.

2.1 Descrição da Base de Dados

Foi utilizado um conjunto de bases de dados para este trabalho, contento a *line600.txt*, *rectangle600.txt*, *square_circle.txt* e a *XOR.txt*.
Na primeira linha da base de dados *line600.txt*, temos um *header* que nos indica que a base de dados tem 600 linhas, em que cada linha representa uma matriz de 8 por 8, portanto uma imagem 8 por 8. Para esta base de dados o objetivo é a classificação correta do que são linhas verticais e das linhas horizontais. A última coluna serve como identificação, 0 para linha horizontal e 1 para vertical. Na figura 2.1 está uma representação de apenas 60 imagens da base de dados *line600.txt*.

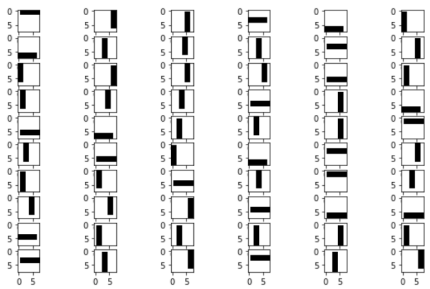

Figura 2.1: Base de dados line600

A seguir temos a base de dados *rectangle600.txt* que é semelhante à *line600.txt*. O *header* desta base de dados é constituído da mesma forma que a base de dados anterior, na linha 1 temos descrito o número total de linhas e o tamanho da matriz, que tem os valores 600 e 8*8 respetivamente. O objetivo para esta base de dados é a identificação correta da posição dos retângulos, se estão na vertical ou na horizontal. A última coluna dos serve como identificação, 0 para o retângulo vertical e 1 para horizontal. Na figura 2.2 está uma representação de apenas 60 imagens da base de dados *rectangle600.txt*.

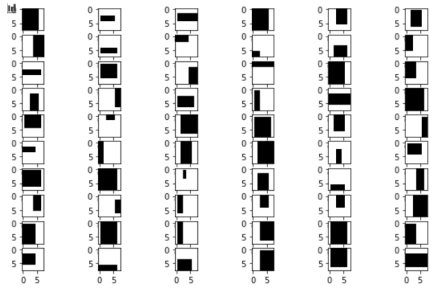

Figura 2.2: Base de dados rectangle600

A base de dados *XOR.txt* é um ficheiro de 5 linhas a contar com o *header*, em que as duas primeiras colunas correspondem ao *input* e a última coluna corresponde ao *output*. O *header* desta base de dados que representa a primeira linha, em que a primeira coluna é o número de *samples*, a segunda coluna indica o número de colunas que correspondem aos atributos e a terceira coluna indica o número de colunas que corresponde ao *output*. O objetivo para esta base de dados é prever corretamente a operação *xor* entre as *features*.
A base de dados *square_circle.txt*, tem também um *header* na primeira linha, em que a primeira coluna corresponde ao número de linhas da base de dados e as duas últimas colunas correspondem à dimensão da matriz que vai formar uma imagem. Os valores das colunas do *header* são 1500, 12 e 12 respetivamente. A última coluna de cada linha corresponde à identificação da imagem, se corresponde a um círculo ou a um quadrado, o valor 1 é para o quadrado e -1 é para o círculo. Na figura 2.3 está uma representação de apenas 60 imagens da base de dados *square_circle.txt*.

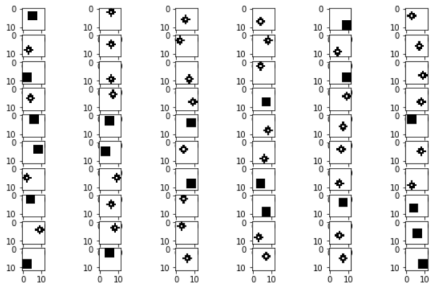

Figura 2.3: Base de dados square circle

2.2 Função *Predict*

Na arquitetura de um *Shallow Logistic Classifier*, os vetores com os valores da base de dados (*input layers*) são passados para uma camada de neurônios com um determinado peso

associado, esta camada é conhecida como *hidden layer*, cada neurônio gera uma resposta de acordo com uma função de ativação logistica (utilizaremos a função *sigmoid*). O *output* de cada neurônio presente na *hidden layer*, é então passado junto de outro peso associado a uma camada final (*output layer*), que consiste somente de um neurônio, cuja a ativação produz o valor de *output* previsto para classificação. A estrutura de um *Shallow Logistic Classifier* é ilustrado na figura 2.4.

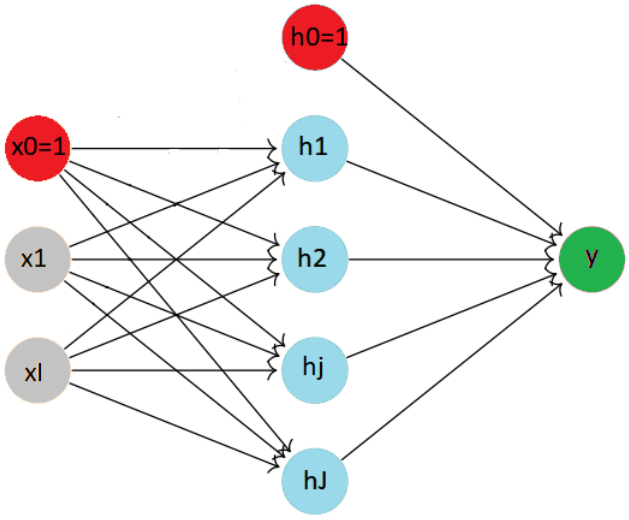

Figura 2.4: Estrutura de um Shallow Logistic Classifier

2.2.1 Neurônio

O neurônio é a unidade atômica de uma rede neuronal. Dada uma entrada de um vetor x da camada de *input*, cujo espaço de atributos é $A = \mathbb{R}^I$ que notamos por $x = (x_1,\cdots,x_I)^T$, e um vetor de bias b notado por $b = (b_1, ..., b_J) \in \mathbb{R}^J$, são conectados a cada neurônio presente na *hidden layer* através de um peso w notado pela matriz $W = \begin{bmatrix} w_{11} & \dots & w_{I1} \\ \vdots & & \vdots \\ w_{1J} & \dots & w_{IJ} \end{bmatrix} \in \mathbb{R}^{JxI}$. Um neurônio h é notado por $h = (h_1, ...{}_{hJ}) \in \mathbb{R}^J$ e pode ser pensado como uma combinação de 2 partes:

- A primeira parte calcula o *output* s utilizando os *inputs* x, b e pesos W

- Na segunda parte é calculado a função de ativação do *output* s e retornando o *output* final h, a fórmula para o cálculo de s e h é apresentada na figura 2.5.

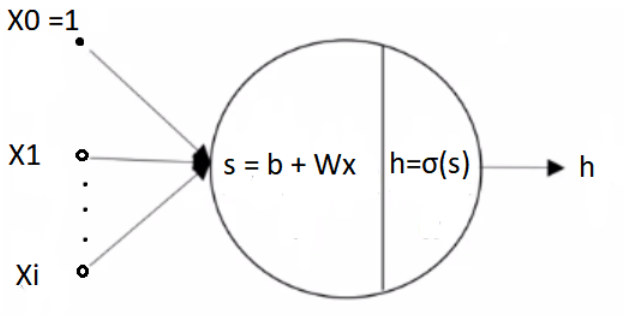

Figura 2.5: Neurônio

2.2.2 Hidden Layer

Uma *hidden layer* é composta por vários neurônios, cada neurônio computa os 2 cálculos apresentados anteriormente. Introduzimos a função para o cálculo de cada neurônio presente na *hidden layer*.

$$h_1 = \sigma(s_1), s_1 = \sum_{i=0}^{I} w_{1i}x_i = b_1 + \sum_{i=1}^{I} w_{1i}x_i$$

$$h_2 = \sigma(s_2), s_2 = \sum_{i=0}^{I} w_{2i}x_i = b_2 + \sum_{i=1}^{I} w_{2i}x_i$$

$$h_j = \sigma(s_j), s_j = \sum_{i=0}^{I} w_{ji}x_i = b_j + \sum_{i=1}^{I} w_{ji}x_i$$

A função de ativação (*sigmoid*) aplicada no *output* de cada neurônio h é definida pela função.

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

Com isso obtemos a seguinte estrutura vetorial:

$$h = \begin{bmatrix} h1 \\ h2 \\ . \\ . \\ hJ \end{bmatrix} = \begin{bmatrix} \sigma(s_1) \\ \sigma(s_2) \\ . \\ . \\ \sigma(s_J) \end{bmatrix} = \sigma \begin{bmatrix} s_1 \\ s_2 \\ . \\ . \\ s_J \end{bmatrix} = \sigma(s),\ s = b + Wx$$

Realizado este processo é obtido o resultado para cada neurônio h presente na *hidden layer*, o próximo passo é calcular o valor da predição propriamente dito, para isto é feito um processo similar ao anterior, os *inputs* agora são definidos pelos h_J *outputs* da camada oculta, além um novo *bias* $c \in \mathbb{R}$, que são conectados à camada de *output* (composta apenas por um neurônio) através de novos pesos v notados por $v = (v_1, ..., v_J) \in \mathbb{R}^J$, então é aplicada

a função de ativação mais uma vez e é gerado o resultado final $\hat{y} \in [0, 1]$. É importante ressaltar que por ser uma classificação logística o *output* é uma probabilidade $p \in [0, 1]$, mas será feito um abuso de notação identificando o *output* como y. A função que produz o *output* final $\hat{y}$ é definida por.

$$z = c + \sum_{j=1}^{J} v_j h_j \Longrightarrow \hat{y} = \sigma(z)$$

2.3 Função Custo

Uma boa maneira de avaliar o desempenho de um algoritmo de *Machine Learning* ou um problema de *Deep Learning* é através de uma função custo, função esta que é responsável por quantificar o erro entre o valor da previsão e os valores esperados. O objetivo será quase sempre minimizar o valor da função custo. Visto que a nossa função de previsão no algoritmo *Shallow Logistic Classifier* é não linear, devido à transformação *sigmoid*, para a função custo é aplicada a fórmula da *Binary Cross Entropy*.

A função *Binary Cross Entropy*, também conhecida por *Log Loss*, mede o desempenho de um modelo de classificação retornando um valor de probabilidade entre 0 e 1 e é dada pela seguinte expressão matemática:

$$E = -\frac{\sum_{m=1}^{N} (y^m * log(\hat{y}^m) + (1 - y^m) * log(1 - \hat{y}^m))}{N}$$

É possível observar os benefícios de escolher o logaritmo através dos gráficos das funções de custo para y=1 e y=0 da figura 2.6. Estas funções monótonas (sempre a subir ou a descer) facilitam o cálculo do gradiente e minimizam o custo.

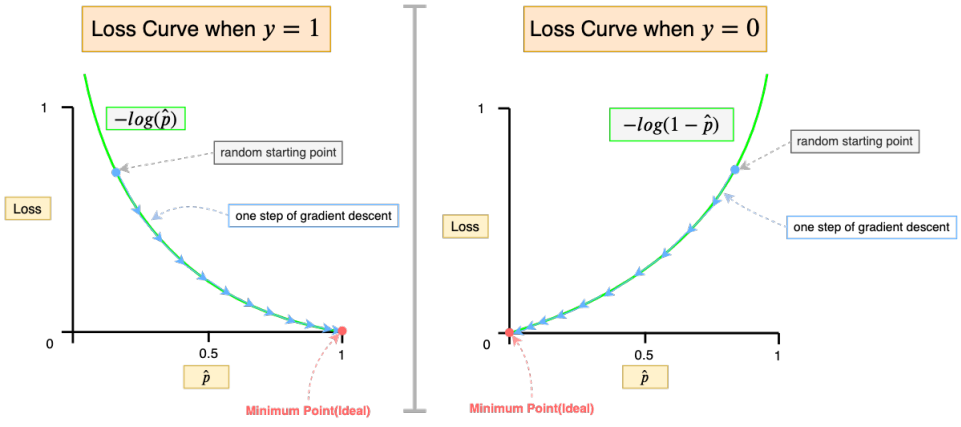

Figura 2.6: Gráficos das funções custo

É importante reter que esta função custo penaliza mais previsões erradas do que recompensa previsões corretas.

2.4 Função *Update*

Os parâmetros (W, b, v, c) da rede são inicializados com valores aleatórios, e para que o classificador faça predições corretas é necessário atualizar estes parâmetros iterativamente de maneira a encontrar os melhores valores possiveis. Para realizar este processo é utilizado o método do gradiente estocástico. As fórmulas que calculam o gradiente da função custo E (definida anterioremente) em ordem a cada parâmetro é definida por:

- $\nabla_w E = (\hat{y}^m - y^m) \begin{bmatrix} h^m{}_1(1-h^m{}_1) & & 0 \\ & \ddots & \\ 0 & & h^m{}_J(1-h^m{}_J) \end{bmatrix} \begin{bmatrix} v_1 x^m{}_1 & \cdots & v_1 x^m{}_I) \\ v_2 x^m{}_1 & \cdots & v_2 x^m{}_I) \\ & \ddots & \\ v_J x^m{}_1 & \cdots & v_J x^m{}_I) \end{bmatrix}$
- $\nabla_b E = (\hat{y}^m - y^m) \begin{bmatrix} h^m{}_1(1-h^m{}_1) & & 0 \\ & \ddots & \\ 0 & & h^m{}_J(1-h^m{}_J) \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_J \end{bmatrix}$
- $\nabla_v E = (\hat{y}^m - y^m) h^m$
- $\nabla_c E = (\hat{y}^m - y^m)$

O cálculo do *update* é realizado de maneira iterativa representada por t iterações. Para não calcular o gradiente dos $n_1, ..., N$ eventos da base de dados e ter um custo alto de processamento, é feito o cálculo do gradiente estocástico, para tal é definido m um único elemento aleatório do banco de dados. O cálculo de *update* de parâmetros é caracterizado pela função iterativa de *update*.

- $W(t+1) = W(t) - \eta \nabla_W E^m(W(t), b(t), v(t), c(t))$
- $b(t+1) = b(t) - \eta \nabla_b E^m(W(t), b(t), v(t), c(t))$
- $v(t+1) = v(t) - \eta \nabla_v E^m(W(t), b(t), v(t), c(t))$
- $c(t+1) = c(t) - \eta \nabla_c E^m(W(t), b(t), v(t), c(t))$

Onde a taxa de aprendizagem é dada por η, e o processo termina quando a função custo E é menor que um determinado treshold ε, ou quando atinge um limite de t iterações.

Capítulo 3

Implementação da Arquitetura

A arquitetura descrita foi implementada com a linguagem de programação Python de forma a analisar os resultados obtidos com as diferentes bases de dados fornecidas. Para além dos métodos de leitura dos ficheiros correspondentes às bases de dados, de vizualização dos dados e dos erros, do cálculo da matriz de confusão, e da função *sigmoid*, o foco principal desta implementação incide no desenvolvimento das funções de custo, de *update*, de predição e de execução do *Shallow Logistic Classifier*.

3.1 Função *Predict*

Como cálculo auxiliar definimos o método *get_h* para calcular o vetor h. Para tal, calculamos o vetor s através da soma do vetor b com o produto escalar da matriz W e do vetor x^m. Este método é demonstrado no apêndice A.1.

Para a fase de predição, após obter o vetor h com o método *get_h*, é calculado o valor de z e é chamado o método *sigmoid* sobre o mesmo para determinar a predição $\hat{y}^m$.

A função de predição corresponte ao método *predict* no *script* desenvolvido, demonstrado no apêndice A.2.

3.2 Função Custo

Para a função de custo é aplicada a fórmula da *Cross Entropy* mas garantimos que, para cada instância da base de dados m, o valor da predição $\hat{y}^m$ pertence ao intervalo $[1^{-12}, 1 - 1^{-12}]$ para evitar problemas numéricos com o logaritmo.

A função de custo corresponde ao método *cost* no *script* desenvolvido, demonstrado no apêndice A.3.

3.3 Função *Update*

Como cálculo auxiliar definimos o método *gradient* para realizar o cálculo dos gradientes de W, b, v e c que tira partido do *package* numpy para a definição das matrizes auxiliares. Há que notar que os cálculos realizados poderiam ser otimizados com a utilização de ferramentas

como o método *tensordot* do mesmo *package*, contudo, sendo este um projeto com foco na aprendizagem, decidimos utilizar abordagens de mais fácil compreensão. Este método é demonstrado no apêndice A.4.

Para a fase de *update* são chamados os métodos *get_h* e *predict* para obter o vetor h e a predição $\hat{y}^m$ respetivamente. Após o cálculo dos gradientes com o método *gradient* são atualizadas ambas as camadas do classificador.

A função de *update* corresponte ao método *update* no *script* desenvolvido, demonstrado no apêndice A.5.

3.4 Função de Execução

Esta função corresponde à fase de treino do classificador e agrega todos os métodos anteriormente descritos. É inicializado um ciclo cujo caso de paragem corresponde a obter um valor de erro não superior a um valor *epsi* ou atingir o número máximo de iterações, sendo ambos os valores pré-definidos. Para cada iteração é escolhida uma instância do conjunto de dados de forma aleatória e é chamado o método *update* para atualizar as camadas do classificador. Contudo, o erro só é calculado a cada *subloop* iterações por motivos de tempo de execução, sendo o valor de *subloop* também pré-definido.

A função de execução corresponte ao método *run_slc* no *script* desenvolvido, demonstrado no apêndice A.6.

Capítulo 4

Benchmarking

Para os diferentes conjuntos de dados foram realizados vários testes para determinar o desempenho da implementação de um *Shallow Logistic Classifier*, verificando se cada *dataset* é linearmente separável ou quantos neurónios, no mínimo, são necessários na camada oculta para separar corretamente os dados.

Antes de analisar os resultados obtidos, é importante ter em consideração que foi aplicada uma operação de *shuffle* sobre os *datasets* e, de seguida, foram separados em conjuntos de treino e de teste com as percentagens de dados de 80% e 20% correspondentemente. Adicionalmente, as classes foram transformadas para os valores 0 e 1, devido à existência de conjuntos de dados com classes -1 e 1, como o *square_ circle*, e isto não pode acontecer devido à utilização da função *sigmoid*.

Para cada conjunto de dados são apresentados os parâmetros utilizados para o treino do modelo, bem como as matrizes de confusão para os conjuntos de treino e teste. Estas matrizes são indicadoras que quão bem o classificador separa os dados visto que apresentam a quantidade de valores corretamente e incorrectamente classificados para cada uma das classes objetivo.

Há que notar que foram realizados múltiplos testes para cada *dataset*, visto que se trata de um problema iterativo aleatório, mas apenas serão apresentados os melhores resultados obtidos.

4.1 Conjunto de Dados *'XOR'*

O *dataset 'XOR'* é um conjunto que pode ser separado com duas retas, contudo, com a nossa implementação conseguimos separar corretamente o conjunto de conjunto de treino mas não o conjunto de teste, mesmo com vários testes com diferentes números de nós na camada oculta e várias fases de treino. Com as seguintes tabelas de confusão é possível verificar que o classificador separou os dados de treino com sucesso mas não os dados de teste que, com as percentagens de separação, corresponde apenas a uma instância.

	0	1
0	1	0
1	0	2

Tabela 4.1: Matriz de confusão para os dados de treino

	0	1
0	0	1
1	0	0

Tabela 4.2: Matriz de confusão para os dados de teste

4.2 Conjunto de Dados *'line600'*

Para o *dataset 'line600'* apenas foi possível separar corretamente todos os dados com 3 nós na camada oculta (J = 3), o que significa que este conjunto de dados não é linearmente separável.

Foram realizados 3 treinos, isto é, foi chamada a função de execução 3 vezes e foram obtidos com os seguintes resultados para cada treino:

	Iterações	Custo	η
Treino 1	2000	0.009343	0.50
Treino 2	2000	0.004935	0.40
Treino 3	2000	0.003701	0.28

Tabela 4.3: Resultados de cada treino

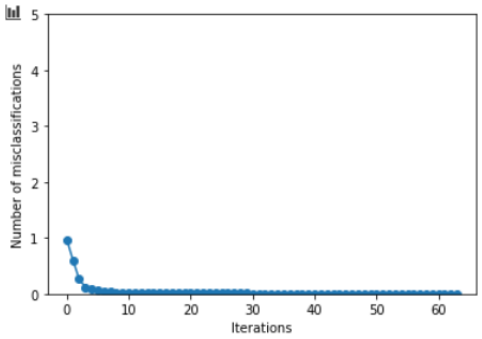

Figura 4.1: Erros obtidos durante o treino

Com as seguintes matrizes de confusão podemos confirmar que, como não existem valores fora da diagonal, tanto o conjunto de treino como o conjunto de teste foram separados com sucesso.

	0	1
0	231	0
1	0	249

Tabela 4.4: Matriz de confusão para os dados de treino

	0	1
0	62	0
1	0	58

Tabela 4.5: Matriz de confusão para os dados de teste

4.3 Conjunto de Dados *'rectangle600'*

Para o *dataset 'rectangle600'* apenas foi possível separar corretamente todos os dados com 9 nós na camada oculta (J = 9), o que significa que este é um conjunto de dados mais difícil de separar linearmente comparativamente ao anterior. Contudo, foram obtidos resultados para números de nós inferiores, nomeadamente entre os valores 4 e 8, sobre os quais o classificador conseguiu classificar corretamente o conjunto de treino mas o mesmo não aconteceu para todos os valores do conjunto de teste, o que leva a suspeitar que existam alguns *outliers* neste conjunto que causem este problema.

Foram realizados 4 treinos e foram obtidos com os seguintes resultados para cada treino:

	Iterações	Custo	η
Treino 1	3000	0.0158060	0.500
Treino 2	3000	0.006467	0.200
Treino 3	3000	0.005587	0.060
Treino 4	3000	0.005430	0.012

Tabela 4.6: Resultados de cada treino

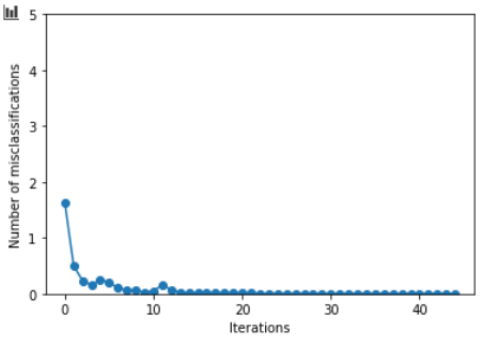

Figura 4.2: Erros obtidos durante o treino

Com as seguintes matrizes de confusão podemos confirmar que, como não existem valores fora da diagonal, tanto o conjunto de treino como o conjunto de teste foram separados com sucesso.

	0	1
0	275	0
1	0	205

Tabela 4.7: Matriz de confusão para os dados de treino

	0	1
0	67	0
1	0	53

Tabela 4.8: Matriz de confusão para os dados de teste

4.4 Conjunto de Dados *'square_ circle'*

Para o *dataset 'square_ circle'* foi possível separar corretamente todos os dados com apenas 1 nó na camada oculta (J = 1), ou seja, este conjunto é linearmente separável.

Sendo este um conjunto mais simples comparativamente aos dois anteriores, foi possível separar os dados com apenas 1 treino do classificador que apresentou os seguintes resultados:

	Iterações	Custo	η
Treino 1	3000	0.007983	1.0

Tabela 4.9: Resultados de cada treino

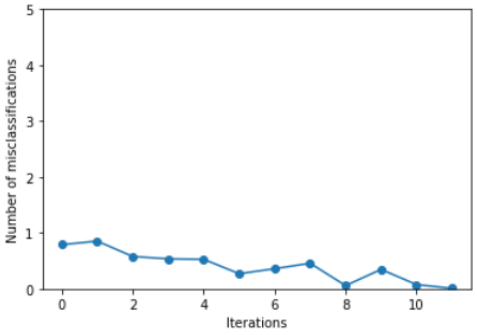

Figura 4.3: Erros obtidos durante o treino

Com as seguintes matrizes de confusão podemos confirmar que, como não existem valores fora da diagonal, tanto o conjunto de treino como o conjunto de teste foram separados com sucesso.

	0	1
0	624	0
1	0	576

Tabela 4.10: Matriz de confusão para os dados de treino

	0	1
0	152	0
1	0	148

Tabela 4.11: Matriz de confusão para os dados de teste

Capítulo 5

Conclusões

Uma vez terminada a exposição do trabalho efetuado podemos tirar algumas conclusões sobre o mesmo.

Como ponto inicial começamos por ressalvar a utilidade das reuniões realizadas com o coordenador deste projeto visto que facilitaram a compreensão da arquitetura do *Shallow Logistic Classifier* e serviram para exclarecer algumas dúvidas que surgiram ao longo da implementação do modelo.

A fase inicial de compreensão do classificador em causa consumiu uma quantidade de tempo considerável mas mostrou-se fundamental para concretização deste trabalho.

Incidindo concretamente sobre o classificador implementado, surgiram alguns problemas, nomeadamente a validação do valor da predição, que provocaram resultados não ideais mas conseguimos eventualmente determinar e ultrapassar estes obstáculos. Quanto aos resultados obtidos, para o *dataset XOR* os resultados não foram ideais contudo, com os resultados obtidos nos restantes conjuntos, consideramos que a maioria dos objetivos deste projeto foram atingidos com sucesso.

Apêndice A

Código da Implementação

A.1 Cálculo do Vetor h

```
1
2       def get_h(x, W, b):
3           # x, w, b -> b + w * x -> s
4           s = b + np.dot(W, x)
5           # s -> s.map(sigmoid) -> h
6           func = lambda a: sigmoid(a)
7           h = np.array([func(s_j) for s_j in s])
8           #print('a')
9           return h
10
```

Listing A.1: Método para o cálculo do vetor h

A.2 Função *Predict*

```
1
2       def predict(x, J, W, b, v, c):
3           h = get_h(x, W, b)
4           # c, v, h -> c + v.T * h -> z
5           sum = 0
6           for j in range(J):
7               sum += v[j] * h[j]
8           z = c + sum
9           # prediction
10          y = sigmoid(z)
11          return y
12
```

Listing A.2: Método para a função *predict*

A.3 Função Custo

```
        def cost(X, Y, N, W, b, v, c):
            epsi = 1.e-12
            sum = 0
            for n in range(N):
                prediction = predict(X[n], J, W, b, v, c)
                # verify if the value of the prediction is in [1e-12, 1-1e-12]
                if prediction < epsi:
                    prediction = epsi
                if prediction > 1 - epsi:
                    prediction = 1 - epsi
                sum += Y[n] * np.log(prediction) + (1 - Y[n]) * np.log(1 -
    prediction)
            E = - sum / N
            return E

```

Listing A.3: Método para a função custo

A.4 Cálculo dos Gradientes

```
        def gradient(x, y, v, h, prediction):
            func = lambda a: a * (1 - a)
            diagonal = np.array([func(h_m) for h_m in h])
            matrix1 = np.diag(diagonal) # J x J
            matrix2 = np.tile(x, (J, 1))
            matrix2 = np.transpose(matrix2) * v
            matrix2 = np.transpose(matrix2) # J x I
            delta = prediction - y
            Gw = delta * np.matmul(matrix1, matrix2) # w's gradient
            Gb = delta * np.dot(matrix1, v)          # b's gradient
            Gv = delta * h                           # v's gradient
            Gc = delta                               # c's gradient
            return Gw, Gb, Gv, Gc

```

Listing A.4: Método para o cálculo dos gradientes

A.5 Função *Update*

```
        def update(x, y, J, eta, W, b, v, c):
            # calculate the gradients
            h = get_h(x, W, b)
            prediction = predict(x, J, W, b, v, c)
            Gw, Gb, Gv, Gc = gradient(x, y, v, h, prediction)
            # update the first layer
            W = W - (eta * Gw) # w (t + 1)
            b = b - (eta * Gb) # b (t + 1)
            # update the second layer
            v = v - (eta * Gv) # v (t + 1)
            c = c - (eta * Gc) # c (t + 1)
            return W, b, v, c
```

Listing A.5: Método para a função *update*

A.6 Função de Execução

```

        def run_slc(X, Y, N, J, subloop, eta, max_iteration, w, b, v, c,
    errors):
            #epsi = 10e-3
            epsi = 0
            iteration = 0
            while (errors[-1] > epsi):
                for j in range(subloop):
                    # choose random data from dataset
                    m = randint(0, N - 1)
                    x = X[m]
                    y = Y[m]
                    # update w, b, v, c from (t) to (t + 1)
                    w, b, v, c = update(x, y, J, eta, w, b, v, c)
                    # calculate error
                error = cost(X, Y, N, w, b, v, c)
                errors.append(error)
                print('number of iterations = %d, cost = %f, eta = %e\r' %(
    iteration, errors[-1], eta), end='')
                iteration += subloop
                if (iteration > max_iteration):
                    break
            return w, b, v, c, errors

```

Listing A.6: Método para a função de execução