Aprendizagem Computacional

Trabalho Prático 2
OCR - Optical Character Recognition

João Silva nº2012131780
Noé Godinho nº 2011159459

13 de Outubro de 2015

Índice

1 Introdução

Este trabalho foca-se no reconhecimento dos algarismos de *0* a *9*, que são introduzidos pelo utilizador através de um rato ou um dispositivo táctil.

Pretende-se que este reconhecimento seja realizado por uma aplicação desenvolvida em *Matlab*, que faz uso de redes neuronais na sua arquitectura interna, disponíveis na *Neural Networks Toolbox*, do *Matlab*.

A aplicação desenvolvida visa o estudo de duas arquiteturas distintas no reconhecimento dos caracteres:

- Na primeira arquitetura a aplicação será constituída por uma *memória associativa* e um *classificador*.

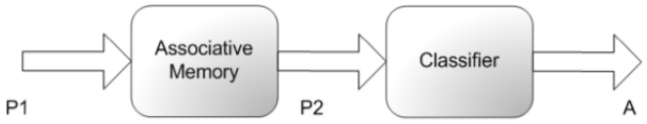

Figura 1: Arquitetura da aplicação com *memória associativa* + *classificador*

- Na segunda arquitetura a aplicação apenas recorre ao *classificador*.

Figura 2: Arquitetura da aplicação apenas com *classificador*

Através da análise das figuras podemos determinar um comportamento padrão para a aplicação:

- Numa fase inicial, os caracteres a identificar poderão, ou não, ser fornecidos à *memória associativa*, que se encarregará de os filtrar e corrigir, ou seja, se os caracteres fornecidos não forem perfeitos, a *memória associativa* aproxima-os dos respetivos caracteres perfeitos.

- Seguidamente, os dados, corrigidos ou não, serão fornecidos ao *classificador*, que se encarregará de proceder à identificação dos mesmos.

Neste documento iremos proceder a uma apresentação em maior detalhe destas duas arquiteturas e das suas implementações, tal como a aplicação desenvolvida em *Matlab*, referindo também a maneira de a utilizar. Pretendemos também fazer uma análise crítica da performance da aplicação, nomeadamente da sua capacidade de classificar corretamente novos caracteres fornecidos.

2 Aplicação

O objetivo desta aplicação é identificar e classificar corretamente caracteres desenhados pelo utilizador com o rato numa interface fornecida.

Tal como já foi referido anteriormente, foram implementadas duas arquiteturas:

- *Memória associativa + classificador*
- *Só classificador*

A cada dígito a ser classificado, é atribuído um elemento do conjunto de dados de entrada da aplicação de uma *classe*, sendo representada dos dígitos *1* ao *10*. Cada elemento corresponde ao dígito desenhado pelo utilizador correspondente, exceto o 10^o elemento, que corresponde ao número *0*.

2.1 Memória associativa + classificador

Esta arquitetura consiste em duas partes.

O objetivo da *memória associativa* é aproximar os caracteres desenhados pelo utilizador, que são imperfeitos, com os respetivos caracteres perfeitos existentes, ou seja, *suavizar* a diferença entre os dois tipos de dígitos. Isto resulta, em teoria, numa melhor classificação por parte do *classificador*.

A seguir aplica-se o *classificador* sobre os dados obtidos da *memória associativa*. Os dígitos são classificados ao usar uma *rede neuronal* previamente treinada, que processa as entradas e produz uma classificação para cada uma, produzindo a identificação dos caracteres e mostrando-as ao utilizador.

2.2 Classificador

No caso de não haver *memória associativa*, ou seja, *só classificador*, os dados são classificados sem haver tentativa de aproximar dos números perfeitos, isto é, tal como são recebidos.

Em relação ao funcionamento, é igual ao explicado na alínea anterior.

2.3 Implementação

Para este trabalho, foi fornecido algum código fonte:

- *mpaper.m*: Esta função permite ao utilizador desenhar os dígitos a classificar e chamar a função *ocr_fun*.

- *ocr_fun.m*: Esta função chama a função *myclassify* e mostra o resultado da classificação.

- *grafica.m*: Esta função grafica um, dois ou três dígitos desenhados, definidos pelo utilizador.

- *showim.m*: Esta função mostra os dígitos desenhados pelo utilizador.

Uma vez percebido o código, foi necessário implementar as funções *myclassify*, *neural_network* e *associative_memory*, de maneira a criar e treinar a *rede neuronal*, aplicar a *memória associativa*, caso o utilizador o deseje, e criar o *classificador*.

- *associative_memory.m*: Caso o utilizador deseje aplicar *memória associativa* nos dados, de maneira a aproximar os dígitos desenhados pelo utilizador dos dígitos perfeitos.
  Quando o utilizador escolhe a *memória associativa* na função *myclassify*, é dada a escolher a opção de usar a regra de *Hebb*, ou seja, a *transposta*, e a *pseudo-inversa*. A seguir é carregado o ficheiro dos dígitos perfeitos, feita a multiplicação dessa matriz com a *transposta* ou *pseudo-inversa* dos dados de treino e depois é feita a multiplicação dessa matriz resultante com a dos dígitos desenhados pelo utilizador.

- *neural_network.m*: Esta função implementa a criação e treino da *rede neuronal*.
  É dada a opção ao utilizador de escolher entre 200 e 500 casos de teste e de escolher o tipo de função de activação:

  - *Binária*: usa a função *hardlim* e o método de aprendizagem *learnp*.
  - *Linear*: usa a função *purelin* e o método de aprendizagem *learngd*.
  - *Sigmoidal*: usa a função *logsig* e o método de aprendizagem *learngd*.

  A seguir é criada a rede neuronal e inicializada e treinada com os parâmetros fornecidos no enunciado do trabalho.

- *myclassify.m*: A partir desta função, é dada a opção ao utilizador de escolher se quer memória associativa e chamar a função da mesma e chamar a função que cria e treina a *rede neuronal*.
  Uma vez feito isso, irá ser validada com a função *sim* do *Matlab* e adaptar os dados da classificação num formato necessário para retornar à função *ocr_fun*, para ser possível mostrar ao utilizador o resultado da classificação.

2.4 Execução

Uma vez explicada a divisão e funcionamento do código, é necessário saber como executá-lo, para poderem ser realizados testes.

O utilizador só precisa de executar a função *mpaper*, que irá fazer um *prompt* de uma interface que lhe permite desenhar os dígitos desejados. Com o botão esquerdo do rato, pode desenhar os dígitos, com o botão direito apagar um dígito de um quadrado e com o botão do meio realizar a classificação.

Uma vez desenhados os dígitos e feito o *click* no botão do meio do rato para *classificar*, irá aparecer no terminal do *Matlab* a opção de escolher *memória associativa + classificador* ou *só classificador*. Caso escolha o 1º, terá de escolher entre a regra de *Hebb* ou a *Pseudo-inversa*. A seguir, irá ser dada a escolha do número de casos de teste, 200 ou 500 e a função de activação, *binária*, *linear* e *sigmoidal*.

Por último, só é necessário esperar que a rede seja treinada e validada e mostre os resultados da classificação.

3 Treino e Testes da Aplicação

3.1 Treino

No desenvolvimento da aplicação, optámos por treinar a *rede neuronal* sempre que pretendêssemos realizar um teste, tanto para o caso da *memória associativa* e *classificador*, como para apenas com este último. Para ambos os casos usámos dados de treino com 200 e 500 elementos, encontrando-se, respetivamente, nos ficheiros *PFinal200.mat* e *PFinal500.mat*, fornecidos com a aplicação e com este relatório.

No caso da *memória associativa*, uma vez que a implementação da nossa aplicação consiste na determinação dos pesos das suas ligações e posterior aplicação desses pesos aos dados de entrada para obter a saída correspondente, o treino desta estrutura resume-se à determinação destes pesos.

Por fim, no caso de se usar apenas o *classificador*, dado que a rede neuronal foi implementada com recurso à *Neural Network Toolbox* do *Matlab*, o seu treino foi realizado com recurso à função *train*, sendo apenas necessário fornecer a rede a treinar, e os respetivos dados de entrada e saída do treino.

3.2 Testes

Para cada uma das arquiteturas usadas realizámos testes individuais para cada função de ativação do *classificador*, considerando ainda *classificadores* treinados com os dois dados de treino, com 200 e 500 elementos.

De forma a podermos realizar uma comparação entre as várias configurações das arquiteturas implementadas na aplicação, em todos os testes apresentados utilizámos o mesmo conjunto de *dados de entrada*, composto por 50 caracteres, desenhados manualmente com recurso à função *mpaper*, contida no código-fonte original.

Em anexo a este documento encontram-se imagens relativas aos testes realizados, onde se pode verificar a saída obtida para cada um deles.

A tabela apresentada abaixo contém os resultados obtidos nos diferentes testes realizados.

Memória Associativa		Classificador			Resultados		
Tipo	Nº Casos Treino	Função Ativação	Nº Casos Treino	Tempo Treino	Corretos	Incorretos	Não Detetados
Nenhuma	-	hardlim	500	10s	31	1	18
Nenhuma	-	purelin	500	8m37s	28	22	0
Nenhuma	-	logsig	500	9m12s	43	7	0
Nenhuma	-	hardlim	200	1s	21	4	25
Nenhuma	-	purelin	200	3m34s	16	34	0
Nenhuma	-	logsig	200	3m32s	36	14	0
Transpose	500	hardlim	500	9s	5	45	0
Transpose	500	purelin	500	8m41s	5	45	0
Transpose	500	logsig	500	9m7s	5	43	2
Transpose	200	hardlim	200	1s	0	0	50
Transpose	200	purelin	200	3m39s	5	45	0
Transpose	200	logsig	200	3m33s	0	0	50
Pinv	500	hardlim	500	7s	12	1	37
Pinv	500	purelin	500	9m10s	31	19	0
Pinv	500	logsig	500	8m58s	33	17	0
Pinv	200	hardlim	200	1s	3	3	44
Pinv	200	purelin	200	3m33s	9	41	0
Pinv	200	logsig	200	3m37s	18	32	0

Figura 3: Resultados dos testes realizados

4 Conclusões

Como é possível verificar, os resultados obtidos não foram os esperados.

Era esperado que a arquitetura que a apresentar melhores resultados fosse a *memória associativa* + *classificador*, sendo que *suaviza* os dados de entrada, aproximando-os dos dígitos perfeitos. No entanto, observa-se que o *classificador* por si só mostra melhores resultados.

O que destaca mais é a regra de *Hebb* ou *transposta*, que apresenta resultados péssimos, o que se pode concluir que este tipo de *memória associativa* não é o correto para este trabalho.

Como a *memória associativa* simula um conjunto de neurónios, resulta do aumento da probabilidade de ativação de neurónios próximos ao ativo. Isto resulta na possível classificação de uma entrada em várias *classes*, o que causa resultados inconclusivos.

Outro ponto a destacar é o facto de existirem poucos casos de teste. Disponibilizamos 200 e 500 casos de teste, sendo possível observar na tabela a diferença de dígitos corretos quando o número de casos é de 500, em vez de 200. Isto é, se houvessem mais casos de teste, a possibilidade da rede ser melhor treinada é maior.

Em seguida, também existe um *bias* na criação dos casos de treino, porque

esses foram realizados maioritariamente pelos dois membros do grupo. Também, os dígitos a serem testados foram desenhados pelos membros do grupo.

Por último, analisando as funções de ativação, observa-se que a função *sigmoidal* é a que apresentou melhores resultados. No entanto, quando não é aplicada *memória associativa*, a função *binária*, apresenta ligeiramente melhores resultados que a *linear*, no entanto, continua a não detetar muitos dígitos.

5 Anexos

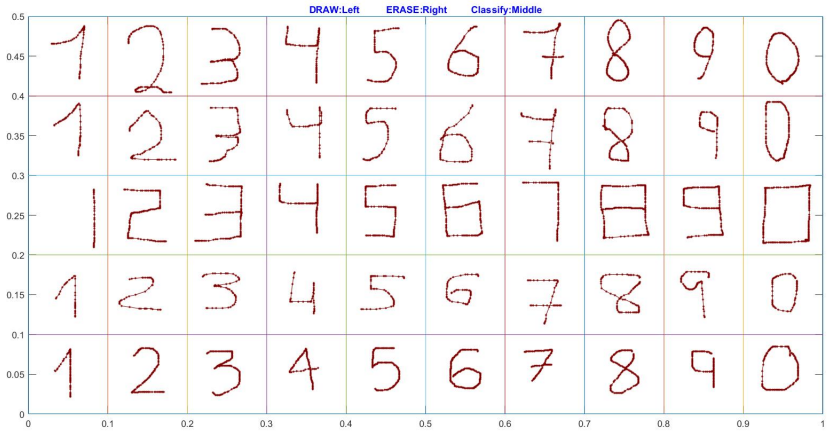

Figura 4: Dados de entrada usados para os testes

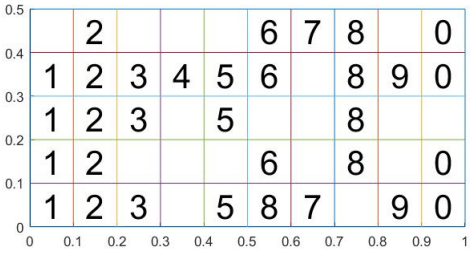

Figura 5: Treino com 500 elementos, utilizando a função *hardlim* como função de ativação, e sem utilização de *memória associativa*

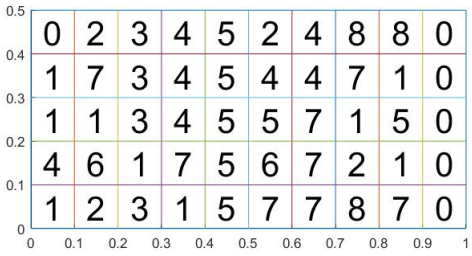

Figura 6: Treino com 500 elementos, utilizando a função *purelin* como função de ativação, e sem utilização de *memória associativa*

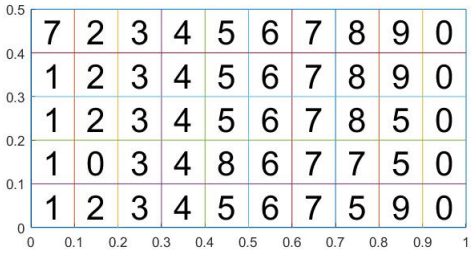

Figura 7: Treino com 500 elementos, utilizando a função *logsig* como função de ativação, e sem utilização de *memória associativa*

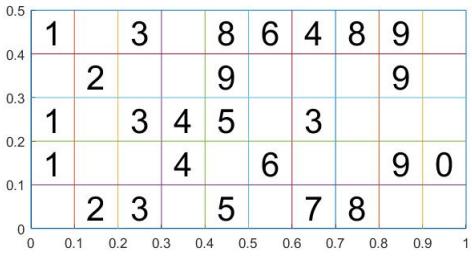

Figura 8: Treino com 200 elementos, utilizando a função *hardlim* como função de ativação, e sem utilização de *memória associativa*

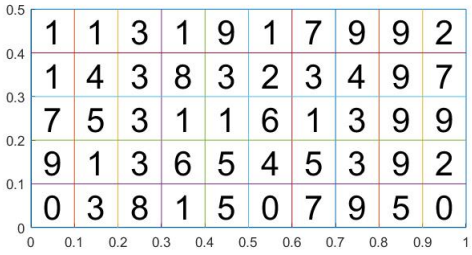

Figura 9: Treino com 200 elementos, utilizando a função *purelin* como função de ativação, e sem utilização de *memória associativa*

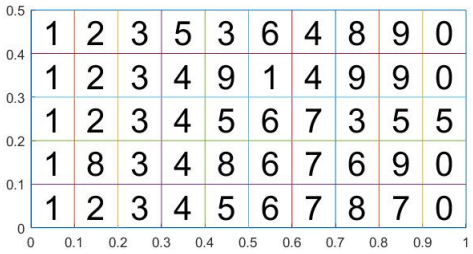

Figura 10: Treino com 200 elementos, utilizando a função *logsig* como função de ativação, e sem utilização de *memória associativa*

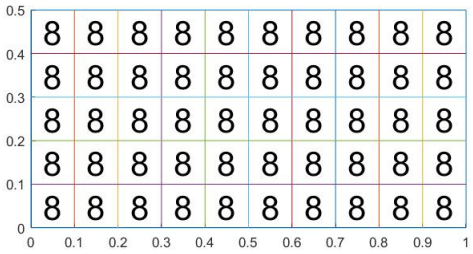

Figura 11: Treino com 500 elementos, utilizando a função *hardlim* como função de ativação, e com utilização de *memória associativa*, fazendo uso da fórmula $target \times input^T$

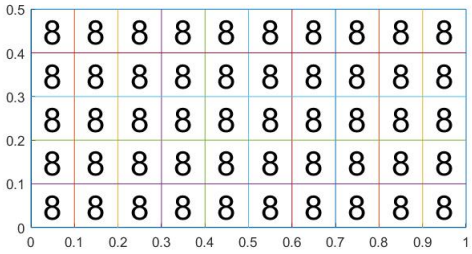

Figura 12: Treino com 500 elementos, utilizando a função *purelin* como função de ativação, e com utilização de *memória associativa*, fazendo uso da fórmula $target \times input^T$

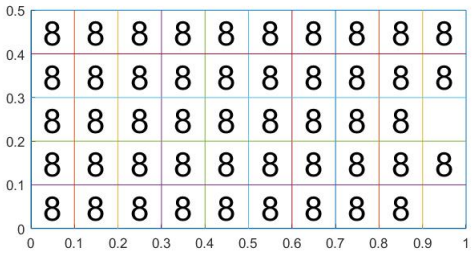

Figura 13: Treino com 500 elementos, utilizando a função *logsig* como função de ativação, e com utilização de *memória associativa*, fazendo uso da fórmula $target \times input^T$

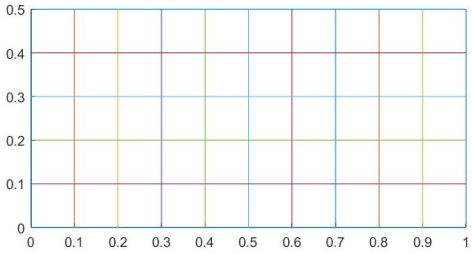

Figura 14: Treino com 200 elementos, utilizando a função *hardlim* como função de ativação, e com utilização de *memória associativa*, fazendo uso da fórmula $target \times input^T$

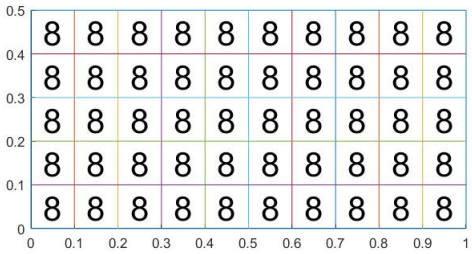

Figura 15: Treino com 200 elementos, utilizando a função *purelin* como função de ativação, e com utilização de *memória associativa*, fazendo uso da fórmula $target \times input^T$

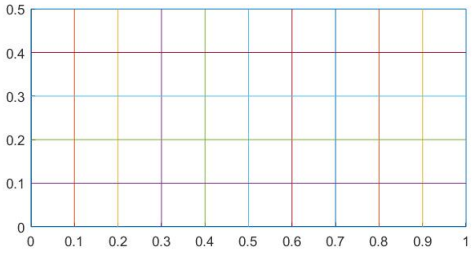

Figura 16: Treino com 200 elementos, utilizando a função *logsig* como função de ativação, e com utilização de *memória associativa*, fazendo uso da fórmula $target \times input^T$

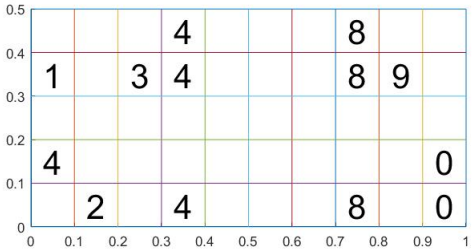

Figura 17: Treino com 500 elementos, utilizando a função *hardlim* como função de ativação, e com utilização de *memória associativa*, fazendo uso da fórmula $target \times pinv(input)$

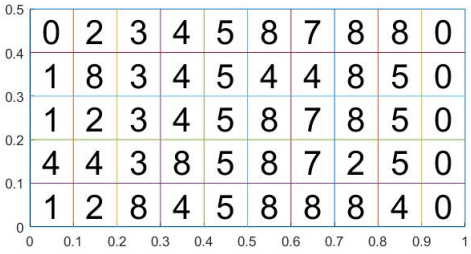

Figura 18: Treino com 500 elementos, utilizando a função *purelin* como função de ativação, e com utilização de *memória associativa*, fazendo uso da fórmula $target \times pinv(input)$

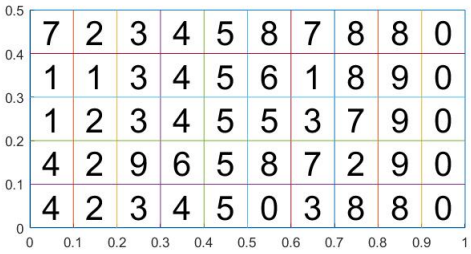

Figura 19: Treino com 500 elementos, utilizando a função *logsig* como função de ativação, e com utilização de *memória associativa*, fazendo uso da fórmula $target \times pinv(input)$

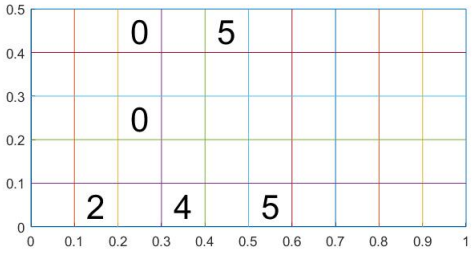

Figura 20: Treino com 200 elementos, utilizando a função *hardlim* como função de ativação, e com utilização de *memória associativa*, fazendo uso da fórmula $target \times pinv(input)$

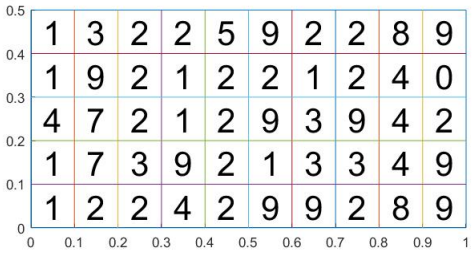

Figura 21: Treino com 200 elementos, utilizando a função *purelin* como função de ativação, e com utilização de *memória associativa*, fazendo uso da fórmula $target \times pinv(input)$

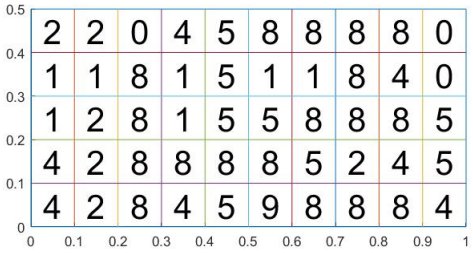

Figura 22: Treino com 200 elementos, utilizando a função *logsig* como função de ativação, e com utilização de *memória associativa*, fazendo uso da fórmula $target \times pinv(input)$