Aprendizagem Computacional - Trabalho Prático 2

João Tiago Márcia do Nascimento Fernandes - 2011162899
Joaquim Pedro Bento Gonçalves Pratas Leitão - 2011150072

11 de Outubro de 2014

Índice

1 Introdução

Este trabalho foca-se no reconhecimento de caracteres da numeração árabe, ou seja, os caracteres 0 a 9.

Pretende-se que este reconhecimento seja realizado por uma aplicação desenvolvida em *Matlab*, que faz uso de redes neuronais na sua arquitetura interna, disponíveis na *Nerual Networks Toolbox* do próprio *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*
- Na segunda arquitetura a aplicação apenas recorre ao *classificador*

As duas arquiteturas apresentadas estão presentes nas figuras que se seguem:

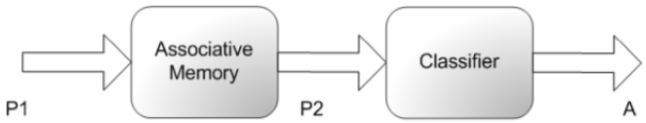

Figura 1: Arquitetura da aplicação com *memória associativa + classificador*

Figura 2: Arquitetura da aplicação apenas com o *classificador*

Através da análise destas 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 está encarregue da sua "filtragem"ou "correção": Se os caracteres fornecidos não forem perfeitos, a memória associativa aproxima-os dos respetivos caracteres perfeitos.
- De seguida os dados, corrigidos ou não, serão fornecidos ao *classificador*, que se encarregará de proceder à identificação dos mesmos.

No presente documento iremos proceder à apresentação em maior detalhe destas duas arquiteturas e das suas implementações, bem como da aplicação *Matlab* desenvolvida, e de como poderá ser utilizada. Pretendemos também fazer uma análise crítica da performance da aplicação, nomeadamente da sua capacidade de classificar corretamente novos caracteres fornecidos.

2 Aplicação Desenvolvida

A aplicação desenvolvida visa identificar e classificar corretamente caracteres desenhados pelo utilizador, implementando para isso as duas arquiteturas apresentadas anteriormente: *Memória Associativa + Classificador* e *Classificador*.

Esta classificação é realizada atribuindo a cada elemento do conjunto de dados de entrada da aplicação uma *classe*, representada por um número inteiro entre *1* e *10*. Quando um dado elemento é classificado com a *classe 1* então a aplicação indica que o elemento em questão corresponde à escrita manual do número 1, mantendo-se o raciocínio para as restantes *classes*, com a exceção da *classe 10*, que representa o número *0*.

Ambas as arquiteturas e os respetivos modos de funcionamento serão apresentados de seguida.

2.1 Memória Associativa + Classificador

Na arquitetura *Memória Associativa + Classificador*, os caracteres desenhados pelo utilizador, que constituem o *input* da nossa aplicação, são fornecidos à *memória associativa*, onde são *"purificados"*.

Com esta operação pretende-se aproximar os caracteres desenhados pelo utilizador, que são naturalmente imperfeitos, dos respetivos caracteres perfeitamente desenhados. Ao purificarmos os dados estaríamos a aumentar a sua precisão, o que, teoricamente, resultará numa melhor classificação por parte do *classificador*.

Por seu turno, o *classificador* recebe como entradas os caracteres purificados e, com recurso a uma rede neuronal previamente treinada, processa cada uma das suas entradas, produzindo uma classificação para cada uma delas, que corresponde à identificação do caracter em questão, classificação essa que será apresentada ao utilizador.

2.2 Classificador

Ao contrário da arquitetura anterior, na arquitetura *Memória Associativa + Classificador*, os caracteres desenhados pelo utilizador não sofrem qualquer tipo de purificação, sendo fornecidos ao *classificador* tal como foram desenhados pelo utilizador.

Tal como o descrito na arquitetura anterior, o classificador recorre a uma rede neuronal previamente treinada para realizar a classificação das entradas. Uma vez classificadas as entradas, o resultado da sua classificação é apresentado ao utilizador.

2.3 Implementação em Matlab

Para o desenvolvimento da aplicação foi-nos fornecido código-fonte, que cria e disponibiliza ao utilizador uma grelha onde este irá desenhar os caracteres a identificar, encarregando-se de toda a lógica interna da aplicação com a exceção

da implementação do sistema de classificação dos caracteres e de toda a lógica a ele inerente (implementação das diferentes arquiteturas da aplicação, realização dos treinos das redes neuronais implementadas, etc)

2.3.1 associativeMemory.m

Neste ficheiro encontra-se uma pequena implementação da memória associativa, presente na função *associativeMemory*. Esta implementção visa *"purificar"* os caracteres desenhados pelo utilizador, aproximando-os do respetivo caracter perfeitamente desenhado. Nesta implementação apenas são calculados os pesos da rede neuronal que constituí a *memória associativa*.

O cálculo dos pesos é feito de duas formas diferentes, com dois conjuntos de dado de treino de diferentes dimensões, escolhidos pelo utilizador. Para o efeito, foram utilizados dados de treino com 100 e 500 casos de teste.

Assim, numa situação, os pesos são calculados recorrendo à fórmula: $target \times input^T$. Na segunda situação, os pesos são calculados de acordo com: $target \times pinv(input)$, onde *input* corresponde aos dados de treino fornecidos à *memória associativa*, *target* corresponde aos respetivos valores esperados à saída da *memória associativa* e *pinv* é uma função específica do *Matlab*.

Uma vez calculados os valores desses pesos a função *associativeMemory* retorna, devolvendo os valores calculados. A *"purificação"* dos dados criados pelo utilizador só será realizada na função *myclassify* (presente no ficheiro *myclassify*), após terem sido obtidos os pesos da *memória associativa*.

2.3.2 createNetwork.m

No ficheiro *createNetwork.m* encontramos a função *createNetwork*, responsável pela criação e treino de uma rede neuronal que representará o *classificador*.

Esta rede é criada de acordo com algumas características pré-definidas, e um conjunto de características escolhidas pelo utilizador, como é o caso da função de ativação. Uma vez criada a rede neuronal, esta é treinada com um conjunto de dados previamente obtidos, e que se encontram nos ficheiros *PTreino500.mat* e *Tfinal500.mat*.

2.3.3 myclassify.m

No ficheiro *myclassify.m* encontramos a função *myclassify*, onde se encontra uma boa parte da lógica principal da aplicação.

Esta função é chamada pela função *ocr_func*, que por sua vez é chamada pela função *mpaper*, e recebe como argumento os caracteres desenhados pelo utilizador. Ambas as funções *ocr_func* e *mpaper* são funções do código-fonte fornecido originalmente.

Caso o utilizador tenha indicado a utilização de uma *memória associativa*, e qual o método para o cálculos dos seus pesos, a função verifica se a *memória associativa* pretendida já foi previamente criada. Caso tenha sido criada, os seus pesos são carregados em memória e os caracteres desenhados pelo utilizador são

"purificados". Caso nenhuma tenha sido criada, a *memória associativa* é criada, os seus pesos são determinados com base em valores de treino previamente computados, e de seguida os dados fornecidos pelo utilizador são *"purificados"*.

Após esta etapa a função solicita ao utilizador dados relativos a propriedades da rede neuronal implementada pelo *classificador*, como por exemplo a *função de ativação* a utilizar. De seguida é criada e treinada a rede neuronal implementada pelo *classificador*, fazendo posteriormente a classificação dos caracteres desenhados pelo utilizador.

Caso o utilizador tenha optado por utilizar uma *memória associativa*, os dados de treino do *classificador* são também *"purificados"* antes de serem fornecidos à rede neuronal criada como dados de treino.

Após a classificação dos caracteres, esta é retornada pela função, sendo posteriormente apresentada ao utilizador.

2.3.4 run.m

Este ficheiro contém um *script* que permite executar a aplicação. Neste *script* o utilizador poderá definir alguns parâmetros da rede, nomeadamente a sua arquitetura e a função de ativação a utilizar no *classificador*. O funcionamento da aplicação, tal como a estrutura deste ficheiro, serão abordados com maior detalhe na secção que se segue.

2.4 Execução

Para executar a aplicação o utilizador deverá executar o ficheiro *run.m*. Uma vez iniciado, será pedido ao utilizador que selecione uma de duas possíveis arquiteturas para a aplicação: Utilizando *Memória Associativa* em conjunto com o *Classificador*, e utilizando apenas o *Classificador*.

Caso o utilizador selecione a utilização da *Memória Associativa*, então terá de escolher o tipo de treino a realizar na mesma.

Existem dois tipos de treino distintos, utilizando cada um, dados de treino com diferentes dimensões, que o utilizador poderá escolher. Para o efeito, foram utilizados dados de treino com 100 e 500 casos de teste.

Assim, no primeiro tipo de treino, os pesos são calculados de acordo com a seguinte fórmula: $target \times input^T$ (*Regra da Transposta*), enquanto que no segundo caso os pesos são calculados de acordo com a fórmula: $target \times pinv(input)$ (*Regra de Pseudo-Inversa*), onde *input* corresponde aos dados de treino fornecidos à *memória associativa*, *target* corresponde aos respetivos valores esperados à saída da *memória associativa* e *pinv* é uma função específica do *Matlab*.

Após esta seleção a *Memória Associativa* é criada, aparecendo de seguida uma grelha, onde o utilizador desenhará os caracteres a serem classificados. Assim que os caracteres são desenhados, o utilizador tem de carregar no botão do meio do seu rato, de forma a transitar para a fase de execução seguinte.

Nesta fase, o utilizador terá que escolher algumas características da rede neural que o classificador da aplicação implementa. Caso já exista alguma rede previamente criada com as características especificadas pelo utilizador, essa rede

é carregada para memória e utilizada na execução. Caso ainda não exista, uma rede neuronal com as características desejadas é criada e treinada, sendo também guardada para posteriores execuções.

O processo de criação e treino desta rede neuronal é feito com recurso à *Neural Network Toolbox*, que através do *Matlab* disponibiliza uma interface para a criação e programação de uma rede neuronal, bem como para o seu treino.

Uma vez obtida a rede a utilizar, esta classifica os dados introduzidos pelo utilizador, que lhe serão apresentados numa grelha semelhante à onde inicialmente desenhou os caracteres.

3 Treino e Testes da Aplicação

Após o desenvolvimento inicial da aplicação foi necessário proceder ao treino das diferentes estruturas implementadas (*memória associativa* e *classificador*), testando de seguida a aplicação, a fim de aferir do seu correto, ou incorreto, funcionamento.

De seguida passamos a descrever o processo de treino adotado para a *memória associativa* e para o *classificador*.

3.1 Treino da Memória Associativa

Uma vez que a implementação *memória associativa* 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, de acordo com as expressões indicadas anteriormente.

Assim, treinámos duas *memórias associativas* diferentes, recorrendo a dados de treino com *100* e *500* elementos. Estes dados encontram-se, respetivamente, nos ficheiros *PTreino100.mat* e *Tfinal100.mat*, e nos ficheiros *PTreino500.mat* e *Tfinal500.mat*, fornecidos com a aplicação e com este relatório.

Uma vez treinada cada *memória associativa*, recorremos à função *save* do *Matlab*, guardando assim os valores dos pesos, o que nos permite utilizar estas duas variantes da *memória associativa* sem ser necessário fazer previamente o seu treino.

As duas *memórias associativas* treinadas encontram-se nos ficheiros *associative_weights_Transpose.mat* e *associative_weights_Pinv.mat*.

3.2 Treino do Classificador

À semelhança do descrito para a *memória associativa*, também no treino da rede neuronal implementadas para o *classificador* recorremos a dados de treino com *100* e *500* elementos.

Neste caso, dado que a rede neuronal foi implementada com recurso à *Neural Network Tollbox* 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.

3.3 Testes

Após o desenvolvimento e treino da aplicação realizámos alguns testes, variando as suas configurações: Consideramos as duas arquiteturas apresentadas, *memória associativa + classificador* e *classificador*.

Para cada uma destas arquiteturas realizámos testes individuais para cada função de ativação do *classificador*, considerando ainda *classificadores* treinados com os dois dados de treino: Dados com *100* e *500* elementos de treino.

A fim de 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 fornecido originalmente.

Em anexo a este documento encontram-se ainda algumas imagens dos testes realizados, onde se pode visualizar, para cada teste, o tempo de treino da rede neuronal do *classificador*, os dados de entrada utilizados e a saída obtida.

Na tabela que se segue apresentamos os resultados obtidos para os diferentes testes realizados.

Queremos desde já salientar os casos em que o tempo de treino da rede neuronal do *classificador* foi bastante reduzido (inferior a 1 minuto), bem como os casos em que este não detetou (correta ou incorretamente) nenhum dos exemplos de teste.

Estas situações permitem-nos desde já concluir que existem situações em que a função de ativação escolhida, devido à sua natureza pouco suave, não é a mais adequada para classificar este tipo de dados. Para além disso, podemos também verificar que a utilização da *Regra de Transposta* na *memória associativa* não se revelou eficaz, o que nos leva a concluir que esta regra não é aplicável ao trabalho realizado.

Memória Associativa		Classificador			Resultados		
Tipo	Nº Casos Treino	Função Ativação	Nº Casos Treino	Tempo Treino	Corretos	Incorretos	Não Detetados
Nenhuma	-	hardlim	100	1s	34%	10%	56%
Nenhuma	-	logsig	100	5min22s	52%	48%	0%
Nenhuma	-	purelin	100	2min05s	44%	56%	0%
Nenhuma	-	hardlim	500	15s	60%	4%	36%
Nenhuma	-	logsig	500	27min10s	84%	16%	0%
Nenhuma	-	purelin	500	26min07s	36%	64%	0%
Pinv	100	hardlim	100	1s	28%	16%	56%
Pinv	100	logsig	100	5min19s	46%	54%	0%
Pinv	100	purelin	100	5min14s	38%	62%	0%
Pinv	500	hardlim	500	41s	28%	20%	52%
Pinv	500	logsig	500	27min1s	48%	52%	0%
Pinv	500	purelin	500	25min48s	38%	62%	0%
Transpose	100	hardlim	100	3min20s	28%	16%	56%
Transpose	100	logsig	100	5min22s	0%	0%	100%
Transpose	100	purelin	100	0s	0%	0%	100%
Transpose	500	hardlim	500	18min16s	30%	18%	52%
Transpose	500	logsig	500	29min39s	0%	0%	100%
Transpose	500	purelin	500	1s	0%	0%	100%

Figura 3: Resultados dos testes realizados

4 Conclusões

Após uma análise crítica dos resultados obtidos existem alguns pontos que consideramos importante salientar.

O primeiro, e talvez o mais surpreendente, ponto refere-se às arquiteturas implementadas e em comparação neste trabalho: *memória associativa + classificador* vs *classificador*. Analisando os resultados obtidos podemos concluir que a arquitetura que apresenta melhores resultados é a que apenas contém o *classificador*.

De facto, considerámos a *memória associativa* como uma espécie de *"corretor"* dos dados de entrada do sistema, aproximando-os dos respetivos valores perfeitos, o que previámos que tornasse a sua classificação mais fácil. No entanto tal não se verificou.

Após analisar o resultado da aplicação de ambas as versões da memória associativa a algumas entradas fornecidas ao sistema, verificámos que este se afastava do valor perfeito respetivo. Mais ainda, este desvio é mais significativo quando os pesos da *memória associativa* são calculados de acordo com a *Regra da Transposta*, do que quando é utilizada a *Regra de Pseudo-Inversa*.

Uma possível explicação para este resultado prende-se com o facto da *memória associativa* simular um *"conjunto"* de neurónios, onde se um dado neurónio é ativado, então é bastante provável que os neurónios próximos desse também sejam ativados. Assim, existe uma maior possibilidade de uma dada entrada ser classificada em várias *classes* em simultâneo, e por isso inconclusivas. Desta forma podemos justificar as elevadas percentagens de dados não detetados sempre que utilizámos a *memória associativa*.

Dados os dois métodos de treino para a *memória associativa* discutidos neste documento, verificamos que estes são métodos de *treino em batch*, isto é, o treino é realizado num único passo, envolvendo todos os exemplos de treino.

Alternativamente, poderíamos considerar a utilização de métodos de *treino iterativo*, onde os pesos da *memória associativa* seriam calculados e ajustados analisando cada caso de treino individualmente, o que poderia resultar numa implementação da *memória associativa* mais bem conseguida, que efetivamente aproximaria os dados de entrada dos respetivos valores perfeitos.

Gostaríamos também de salientar que, embora o enunciado deste trabalho refira a implementação da *Regra de Hebb* para treinar os pesos da *memória associativa*, optámos por não incluir essa regra, uma vez que esta só pode ser aplicada a matrizes *ortonormadas*, o que nem sempre se verifica, dado que o utilizador é livre de desenhar qualquer conjunto de caracteres, que quando mapeados para uma matriz (com recurso à função *mpaper*) nem sempre produzem uma matriz *ortonormada*.

O terceiro ponto que gostaríamos de referir prende-se com a análise da *performance* da rede do *classificador* tendo em conta o número de dados com os quais o seu treino foi realizado. Uma vez que não utilizámos dados de treino excessivamente elevados (apenas recorremos a *100* e a *500* dados de treino) seria de esperar que as redes treinadas com um maior número de casos de treino apresentariam um maior número de deteções corretas.

Efetivamente, verificámos essa situação, uma vez que os *classificadores* treinados com *500* dados de treino apresentaram percentagens de deteções corretas maiores do que os *classificadores* treinados com *100* dados de treino. Uma vez que o número de casos de treino não foi excessivamente elevado, ao fornecermos mais dados ao *classificador* este vai cobrir um maior número de situações apresentando uma maior convergência, que resulta numa melhor performance. Quando o número de exemplos de teste é muito elevado, o sistema apresenta uma excelente *performance* para os exemplos de teste, mas uma *performance* inferior para novos exemplos, não contemplados nesses exemplos de teste (problema denominado *overfitting*).

Analisando também os resultados, tendo em conta as diferentes funções de ativação implementadas, podemos concluir que a função *sigmoidal, logsig* apresentou os melhores resultados, uma vez que com essa função obtiveram-se as maiores percentagens de classificações corretas, nomeadamente a maior percentagem de classificações corretas registada em todos os testes realizados: 84%.

Para finalizar, tendo em conta os resultados apresentados, podemos concluir que o sistema, quando não faz uso da memória associativa apresenta percentagens de classificações corretas mais elevadas, sendo capaz de identificar alguns

dígitos desenhados de diferentes maneiras. Mais ainda, quando a função de ativação do *classificador* é a função *sigmoidal*, este apresenta melhores resultados. Isto permite-nos concluir que esta função é a que mais se adequa ao tipo de classificação e ao tipo de dados presentes neste trabalho.

Tendo em conta a tarefa realizada, a aplicação desenvolvida consegue atingir alguns dos objetivos propostos: apresenta alguma robustez e capacidade de generalização, sendo capaz de identificar corretamente dígitos perfeitos bem como alguns dígitos não perfeitos. No entanto, esta está longe de fazer uma classificação perfeita dos caracteres, nomeadamente os desenhados de forma diferente dos incluídos nos dados de treino.

Não obstante, uma vez que o mesmo caracter pode ser desenhado de várias maneiras diferentes, consideramos que, quando não fazemos uso da *memória associativa*, os resultados obtidos são bastante razoáveis.

5 Anexos

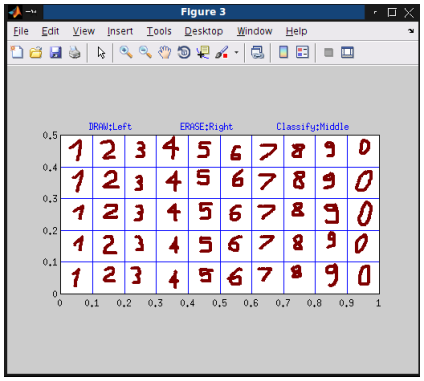

Figura 4: Dados de entrada utilizados para os testes realizados à aplicação

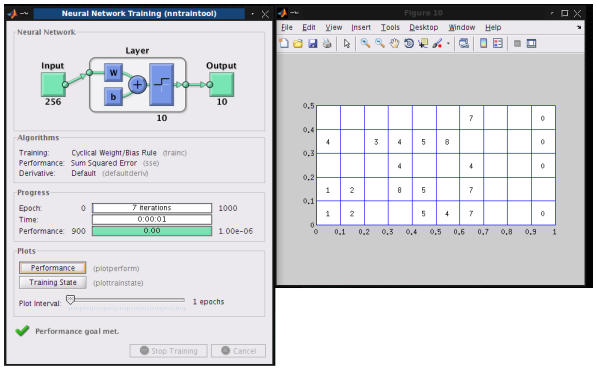

Figura 5: Tempo de treino da rede neuronal do classificador recorrendo a dados de treino com 100 elementos, utilizando a função *hardlim* como função de ativação e sem utilização de *memória associativa*. São também visíveis os resultados da classificação do input da figura 3

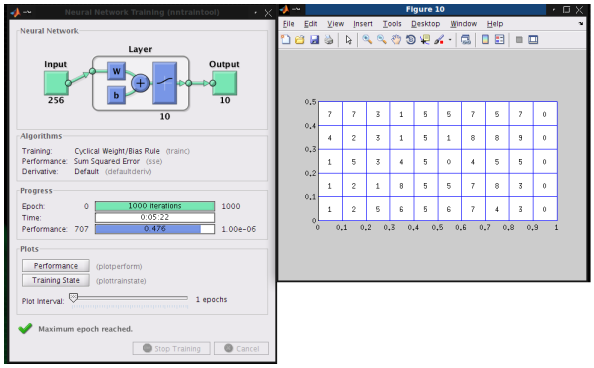

Figura 6: Tempo de treino da rede neuronal do classificador recorrendo a dados de treino com 100 elementos, utilizando a função *logsig* como função de ativação e sem utilização de *memória associativa*. São também visíveis os resultados da classificação do input da figura 3

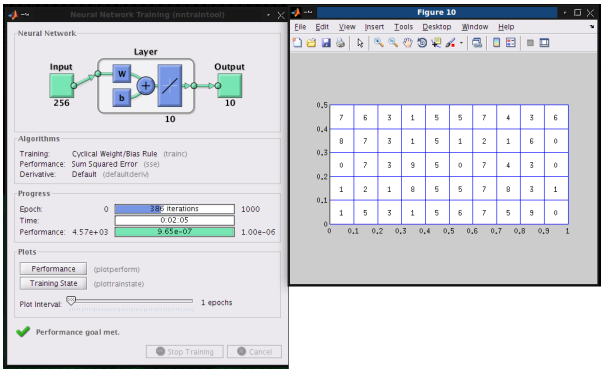

Figura 7: Tempo de treino da rede neuronal do classificador recorrendo a dados de treino com 100 elementos, utilizando a função *purelin* como função de ativação e sem utilização de *memória associativa*. São também visíveis os resultados da classificação do input da figura 3

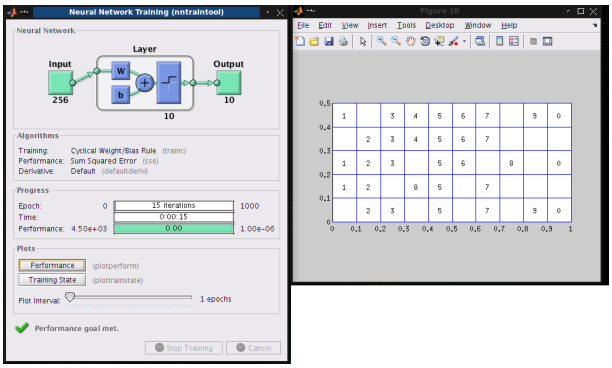

Figura 8: Tempo de treino da rede neuronal do classificador recorrendo a dados de treino com 500 elementos, utilizando a função *hardlim* como função de ativação e sem utilização de *memória associativa*. São também visíveis os resultados da classificação do input da figura 3

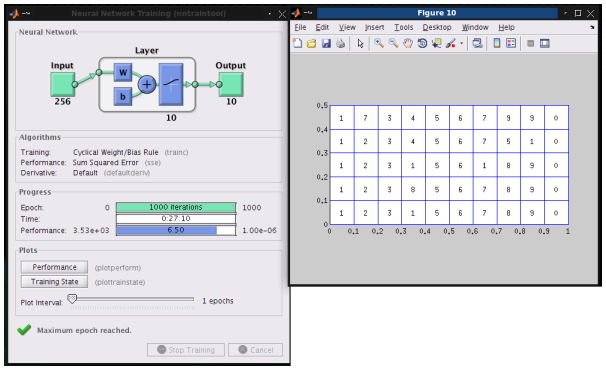

Figura 9: Tempo de treino da rede neuronal do classificador recorrendo a dados de treino com 500 elementos, utilizando a função *logsig* como função de ativação e sem utilização de *memória associativa*. São também visíveis os resultados da classificação do input da figura 3

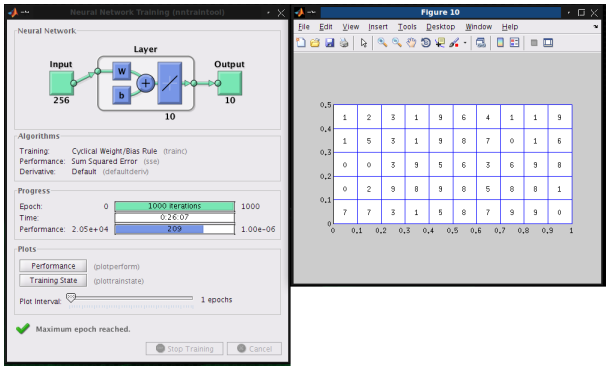

Figura 10: Tempo de treino da rede neuronal do classificador recorrendo a dados de treino com 500 elementos, utilizando a função *purelin* como função de ativação e sem utilização de *memória associativa*. São também visíveis os resultados da classificação do input da figura 3

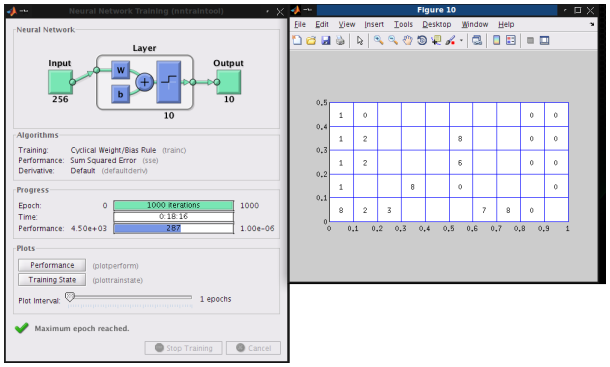

Figura 11: Tempo de treino da rede neuronal do classificador recorrendo a dados de treino com 500 elementos, utilizando a função *hardlim* como função de ativação e com utilização de *memória associativa*, fazendo uso da fórmula $target \times input^T$. São também visíveis os resultados da classificação do input da figura 3

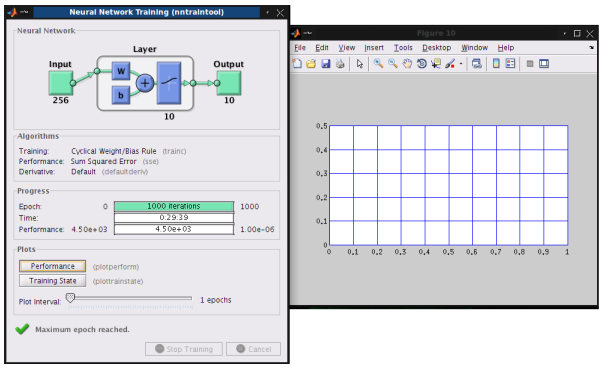

Figura 12: Tempo de treino da rede neuronal do classificador recorrendo a dados de treino com 500 elementos, utilizando a função *logsig* como função de ativação e com utilização de *memória associativa*, fazendo uso da fórmula $target \times input^T$. São também visíveis os resultados da classificação do input da figura 3

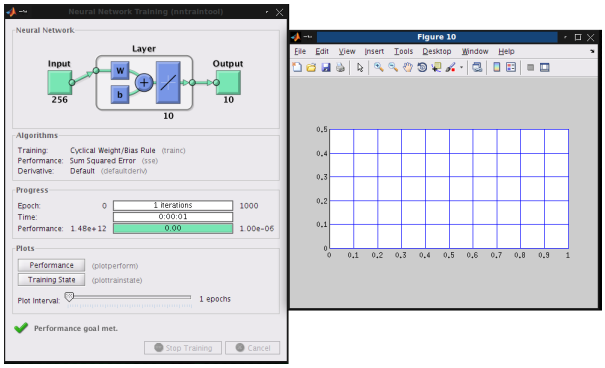

Figura 13: Tempo de treino da rede neuronal do classificador recorrendo a dados de treino com 500 elementos, utilizando a função *purelin* como função de ativação e com utilização de *memória associativa*, fazendo uso da fórmula $target \times input^T$. São também visíveis os resultados da classificação do input da figura 3

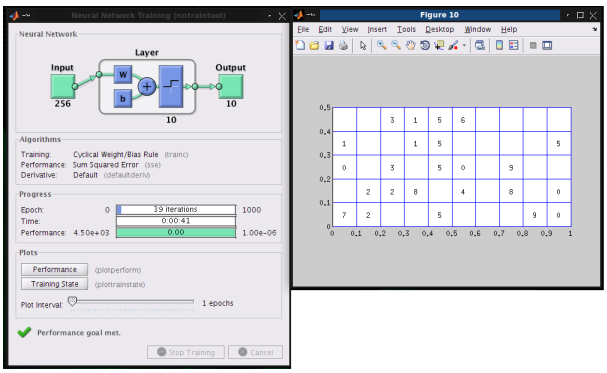

Figura 14: Tempo de treino da rede neuronal do classificador recorrendo a dados de treino com 500 elementos, utilizando a função *hardlim* como função de ativação e com utilização de *memória associativa*, fazendo uso da fórmula $target \times pinv(input)$. São também visíveis os resultados da classificação do input da figura 3

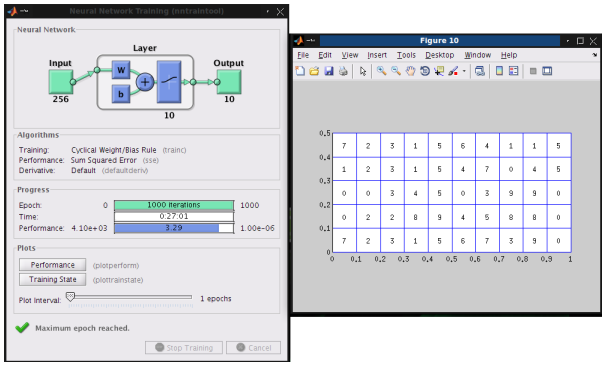

Figura 15: Tempo de treino da rede neuronal do classificador recorrendo a dados de treino com 500 elementos, utilizando a função *logsig* como função de ativação e com utilização de *memória associativa*, fazendo uso da fórmula $target \times pinv(input)$. São também visíveis os resultados da classificação do input da figura 3

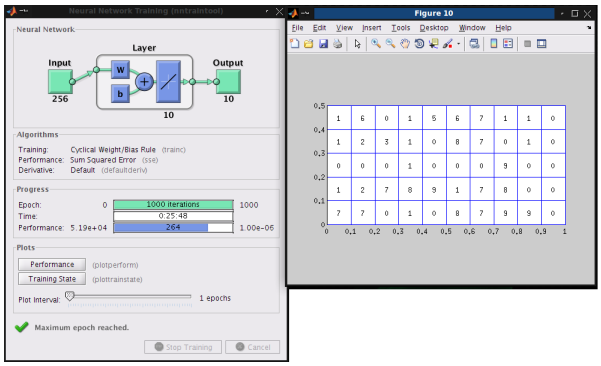

Figura 16: Tempo de treino da rede neuronal do classificador recorrendo a dados de treino com 500 elementos, utilizando a função *purelin* como função de ativação e com utilização de *memória associativa*, fazendo uso da fórmula $target \times pinv(input)$. São também visíveis os resultados da classificação do input da figura 3

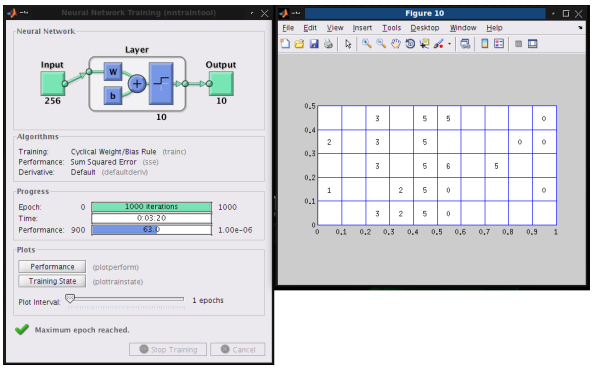

Figura 17: Tempo de treino da rede neuronal do classificador recorrendo a dados de treino com 100 elementos, utilizando a função *hardlim* como função de ativação e com utilização de *memória associativa*, fazendo uso da fórmula $target \times input^T$. São também visíveis os resultados da classificação do input da figura 3

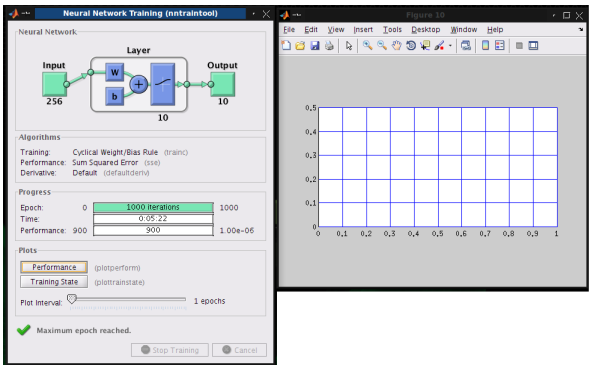

Figura 18: Tempo de treino da rede neuronal do classificador recorrendo a dados de treino com 100 elementos, utilizando a função *logsig* como função de ativação e com utilização de *memória associativa*, fazendo uso da fórmula $target \times input^T$. São também visíveis os resultados da classificação do input da figura 3

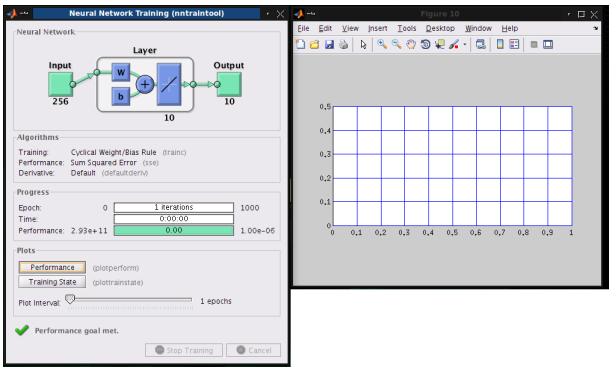

Figura 19: Tempo de treino da rede neuronal do classificador recorrendo a dados de treino com 100 elementos, utilizando a função *purelin* como função de ativação e com utilização de *memória associativa*, fazendo uso da fórmula $target \times input^T$. São também visíveis os resultados da classificação do input da figura 3

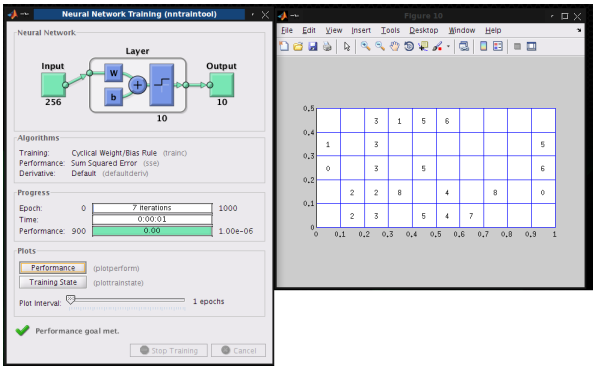

Figura 20: Tempo de treino da rede neuronal do classificador recorrendo a dados de treino com 100 elementos, utilizando a função *hardlim* como função de ativação e com utilização de *memória associativa*, fazendo uso da fórmula $target \times pinv(input)$. São também visíveis os resultados da classificação do input da figura 3

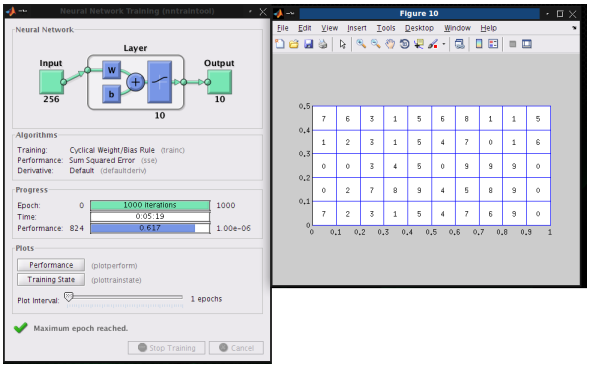

Figura 21: Tempo de treino da rede neuronal do classificador recorrendo a dados de treino com 100 elementos, utilizando a função *logsig* como função de ativação e com utilização de *memória associativa*, fazendo uso da fórmula $target \times pinv(input)$. São também visíveis os resultados da classificação do input da figura 3

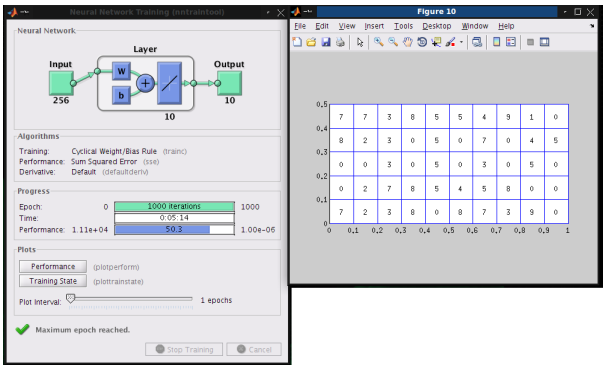

Figura 22: Tempo de treino da rede neuronal do classificador recorrendo a dados de treino com 100 elementos, utilizando a função *purelin* como função de ativação e com utilização de *memória associativa*, fazendo uso da fórmula $target \times pinv(input)$. São também visíveis os resultados da classificação do input da figura 3