Universidade do Minho
Departamento de Informática
Mestrado Integrado em Engenharia Informática

Artificial Neural Networks

Ferramentas de conceção e desenvolvimento

Miguel Miranda, A74726

Braga, 11 de Março de 2018

Conteúdo

Resumo

Intrinsecamente ligadas às temáticas de *Machile learning e Deep Learning*, redes neuronais artificiais mantêm atualmente um destaque de topo enquanto soluções para diversos desafios do presente, associados com inteligência artificial, extração de conhecimento e aprendizagem.

Nesse sentido, torna-se assim relevante compreender os principais fundamentos que dão base a estes sistemas, como forma de identificar as características que permitem a sua manipulação e adaptação para diferentes contextos de aprendizagem. O presente relatório procura realizar uma análise teórica dos principais aspetos relacionados com RNAs e, numa perspetiva mais prática, apresenta ainda exemplos de plataformas sobre as quais é possível conceber e desenvolver estas estruturas.

1. Introdução

Redes neuronais artificias apresentam-se como um sistema conexionista, fortemente inspirado nas características do sistema nervoso central do ser humano. Apesar das RNAs serem um modelo simplificado, a sua arquitetura extremamente interconectada de unidades de processamento permite que estas sejam capazes de generalizar e adquirir conhecimento, através de um processo de aprendizagem.

Tirando partidos das suas características, vários algoritmos de *machine learning* recaem sobre esta estrutura de aprendizagem, devido às suas capacidades de classificação e previsão, em qualquer um dos paradigmas de aprendizagem.

No sentido de introduzir os conceitos em torno das RNAs, o presente relatório apresenta inicialmente um descrição teórica dos principais conceitos associados com a modelação de uma rede neuronal artificial. Depois de analisados os principais tópicos é feita uma análise de algumas soluções e plataformas que permitem criar estas estruturas e, implementar sobre elas algoritmos de aprendizagem e extração de conhecimento. A introdução teórica inicial permite posteriormente justificar e compreender a necessidade de passar alguns argumentos e parâmetros às funções de manipulação de RNAs.

Como estrutura do presente documento: o capítulo 2 apresenta uma descrição breve daquilo que atualmente se entende por uma rede neuronal artificial, focando com relevo as características das suas unidades de processamento (neurónios), as arquiteturas existentes e o algoritmo de aprendizagem de *Back-Propagation*; o capítulo 3 apresenta algumas das ferramentas atualmente utilizadas para desenvolver RNA, sendo dado especial ênfase à criação e gestão de uma RNA na linguagem R e Julia, através de uma descrição detalhada dos procedimentos necessários. No final do deste capítulo é ainda realizada uma comparação geral a nível das duas implementações analisadas.

Por fim, o capitulo 4 apresenta as conclusões relativamente ao tema abordado e ferramentas exploradas.

2. Redes Neuronais Artificiais

2.1 Descrição geral

Apresentando-se como uma abordagem de *machine learning*, os algoritmos de redes neuronais artificias (RNAs) baseiam-se num sistema conexionista, inspirado no funcionamento da estrutura cerebral humana.

Estas técnicas recorrem a uma topologia em rede para receber informações que, após processadas, influenciam o estado interno de cada neurónio da rede. O conhecimento é assim adquirido através do ambiente por um processo iterativo de aprendizagem. Este conhecimento é armazenado nas conexões da rede, através de uma adaptação dos pesos dos neurónios -ou nodos- ao longo de um processo de aprendizagem.

Independente do contexto, uma rede neuronal apresenta uma arquitetura genérica (Figura 2.2), normalmente composta pelas seguintes camadas [1, 2]:

- **Camada de input**, na qual é recebida a informação a analisar. Apesar dos dados serem normalmente obtidos através de um dataset pré preparado, esta camada pode ser vista como a comunicação da rede com o ambiente, como forma de percecionar e obter dados.

  É assim uma camada sem antecessor, servindo como uma interface para alimenta a rede com nova informação.

- **Camadas intermédias**, que desempenham a tarefa de processamento dos dados de entrada, com base num determinado paradigma de aprendizagem.

- **Camada saída**, onde se devolvem os resultados do processo de aprendizagem da rede.

  O número de nodos de saída está diretamente relacionado com o número de parâmetros que são previsto pela rede num determinado contexto.

Alguns dos benefícios das redes neuronais, comparativamente a outros processos de *machine learning*, estão associados ao seu puder de processamento. Esta capacidade deve-se essencialmente à sua estrutura, que permite executar um ambiente extremamente paralelo. Além deste fator, apresentam ainda outras características de relevo, nomeadamente [2] [1] [3]:

- **Generalização**: No final do processo de aprendizagem, a rede é capaz de descrever um universo completo, através de uma aprendizagem prévia com algumas das suas partes. Além disto, devido à sua robustez, as RNAs são ainda capazes de processar ruído ou informação incompleta, conseguindo generalizar a informação que analisam na fase de treino para novos casos futuros.

- **Não linearidade**: Além de problemas simples, estes algoritmos apresentam-se capazes de analisar características não lineares, através da interconexão de diferentes neurónios e uso de diferentes funções de ativação;
- **Relação entre input-output**: Quando aplicadas num contexto de aprendizagem supervisionada, com um conjunto de input de treino e um dataset de output esperado, as redes neuronais são capazes de gerar resultados satisfatórios de classificação;
- **Adaptação**: Uma das principais características das RNAs reside na sua capacidade de adaptação, útil em ambientes com dados desconhecidos ou em falta.

  Os pesos atribuídos a cada neurónio são capazes de ser adaptar, conforme o domínio sobre a qual a rede se encontra a ser treinada. Alem disto, a sua topologia pode também alterar-se de acordo com as mudanças do ambiente, criando ou reduzindo o número de ligações entre nodos.

  Desta forma, no final do processo de aprendizagem, uma RNA é capaz de generalizar novos casos à sua capacidade de raciocínio.
- **Tolerância a falhas**: Devido à sua topologia, as RNAs permitem um elevada tolerância a falhas. A falha de um neurónio pode ser atenuada pela marcação do peso dos seus axónios como zero, simulando assim o desaparecimento das ligações para aquele neurónio.

  A falha de um neurónio leva ao reajustamento dos restantes, levando a uma degradação gradual e suave da rede, em caso de desativação de algumas conexões entre neurónios.

2.2 Neurónio Artificial

Cada camada de uma RNA é composta por um conjunto de neurónios, identificados pelos nodos da topologia da rede. Um neurónio pode ser visto como uma unidade de processamento da rede, recebendo ligações de outro neurónios e unindo-se a outros nodos da rede semelhantes a si.

As ligações entre neurónios são designadas por axónios, tal como na estrutura biológica do cérebro e, têm a si associadas um peso. Este peso pode, numa interpretação simples, ser visto como a importância da informação que aquele axónio envia a um neurónio. O peso de cada axónio é regulado ao longo do processo de aprendizagem, levando a uma generalização da topologia da rede aos dados, até se obter um resultado coerente com os esperados no contexto. A transmissão de informação entre os nodos é conseguida através de sinapses, novamente seguindo a nomenclatura biológica.

Ao longo do processo de evolução da rede, os neurónios vão atualizando o seu estado, sendo este normalmente representado por um valor numérico entre *[0, 1]*. De uma forma geral, o estado interno de um neurónio é caracterizado por:

- Um **conjunto de conexões**, cada uma com o seu próprio peso. Ao longo da evolução da rede, o sinal propagado por um determinado axónio é multiplicado pelo peso do mesmo, resultando assim na excitação -ou inibição- do nodo;

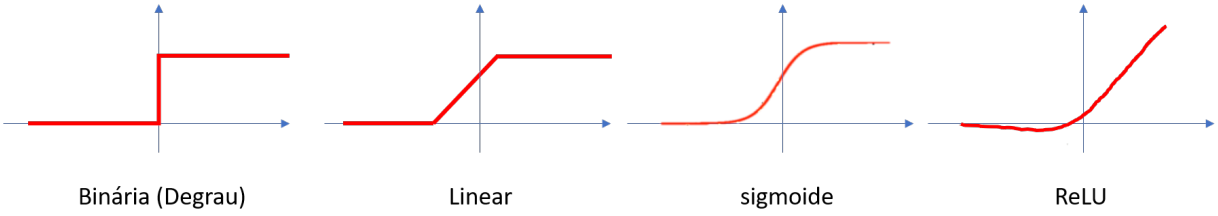


Figura 2.1: Exemplos de diferentes tipos de funções utilizadas como função de ativação dos neurónios.

- Um **integrador**: Apresenta-se como uma função de redução, que simplifica a influencia das N conexões de entrada de um neurónio para um único valor.

  Geralmente, o integrador realiza apenas o somatório $(\sum)$ dos valores recebidos pelas N ligações de entrada, multiplicados pelo peso da respetiva ligação;

- A **função de ativação**: Determina o valor de saída do neurónio, de forma a restringir o valor de saída do neurónio para um determinado valor finito. Este valor é normalmente calculado através do mapeamento do valor do integrador para uma determinada função não linear. De referir que a escolha da função de ativação tem uma influência direta e bastante significativa na capacidade de aprendizagem da rede.

  Algumas das principais funções são a função binária (degrau), linear ou funções do tipo *sigmoide*. [4] Em pesquisas recentes, foi ainda apresentada a função designada por *ReLU*. (Figura 2.1) Esta função, explorada pelos investigadores da *Google Brain Team*, apresenta algumas vantagens face a implementações com a função *sigmoide*.

2.3 Arquiteturas da Rede

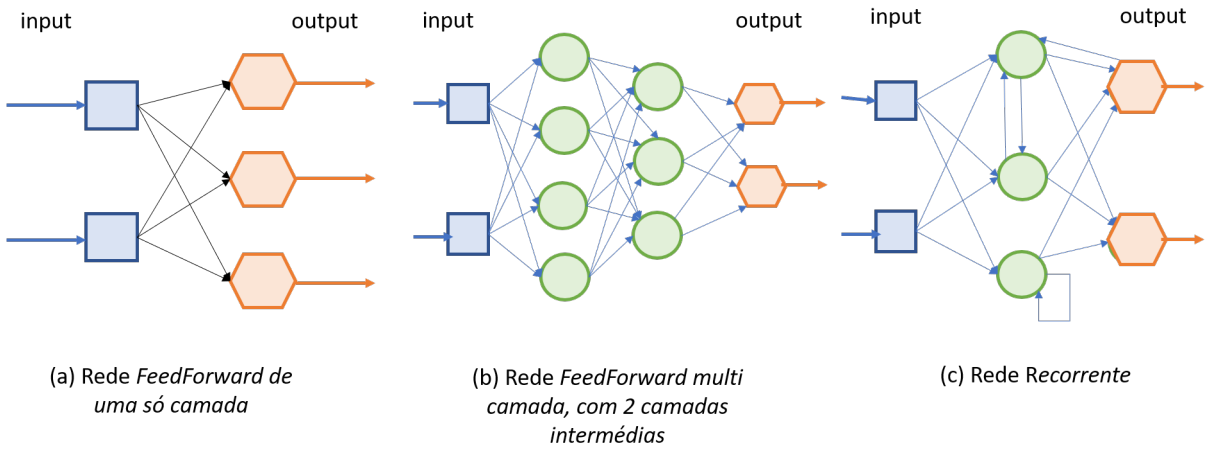
A arquitetura de uma rede neuronal artificial especifica a quantidade de neurónios (nodos) que a mesma possui e, de que formas estes se relacionam e interligam entre si. A arquitetura pode ser visualizada na forma de grafos, no qual as ligações entre os nodos são orientadas.

De uma forma geral existem três tipos distintos de topologias, sendo que quanto mais complexa a arquitetura de uma RNA, maior a sua capacidade para lidar com problemas mais complexos.

- ***Single layer feedforward***: As ligações da rede seguem apenas uma direção. Nesta arquitetura não existem camadas intermédias, existindo apenas as camadas de *input* e *output*.

  A partir de um nodo de entrada existem apenas ligações divergentes (1 para N) e os nodos de saída apresentam ligações convergentes (N para 1). Esta topologia representa as arquiteturas inicias das RNAs. (Figura 2.2-a)

- ***Multi-layer feedforward***: Seguindo os mesmos conceitos da arquitetura anterior, as RNAs *multi-layer* permitem a definição de uma ou mais camadas



(a) Rede *FeedForward de uma só camada*

(b) Rede *FeedForward multi camada, com 2 camadas intermédias*

(c) Rede Recorrente

Figura 2.2: Exemplos de algumas topologias de arquiteturas de RNAs

intermédias. Embora esta relação não seja linear, a adição de camadas intermédias a uma rede permite aumentar a sua capacidade de aprendizagem, podendo assim lidar com funções e relações entre os dados mais complexas.

Estas redes são assim úteis em contextos onde existe um grande número de dados de input, sobre os quais se espera observar alguma relação ou padrões gerais. Contudo, o aumento do número de camadas de uma rede neuronal, leva a um aumento exponencial do seu tempo de aprendizagem. (Figura 2.2-b)

Neste tipo de arquiteturas, o algoritmo de *Back-Propagation* é um dos principais métodos de aprendizagem. Este método calcula a contribuição do erro para cada um dos nodos da rede, ajustando o peso de cada nodo com base no erro calculado (Secção 2.4).

- **Redes recorrentes**: Numa sucessão à arquitetura anterior, as RNAs baseadas em redes recorrentes permitam a existência de conexões de um nodo para si próprio. A ligação de um neurónio para si próprio permite a introdução de conceitos de "memória" no próprio nodo.

  Além deste fator, um neurónio pode ainda ligar-se a outros nodos de camadas anteriores ou da própria camada, permitindo assim a existência de ciclos na estrutura. (Figura 2.2-c)

2.4 *Back-Propagation*

Associado ao método de aprendizagem do Gradiente Descendente, o algoritmo de *Back-Propagation* é um dos algoritmos mais utilizados no paradigma de aprendizagem supervisionada de redes neuronais artificiais.

Regras de aprendizagem do tipo Gradiente Descendente baseiam-se na diminuição de um erro, sendo este estimado entre um valor pretendido e um valor obtido

pela previsão da rede. [3] Para cada estado do processo de evolução da rede, o objetivo prende-se com a minimização de uma função de custo, definida em termos do valor de erro.

Este processo de avaliação é descrito pela Equação 2.1, onde η é a taxa de aprendizagem e $\Delta\epsilon$ a diferenças entre o valor esperado e o valor obtido para uma dada entrada (gradiente da função de custo).

$$\Delta w = \eta \Delta \epsilon \tag{2.1}$$

No caso especifico do algoritmo de *Back-Propagation*, a sua implementação segue os seguintes passos:

- Antes de se iniciar o treino, são atribuídos valores aleatórios aos axónios da arquitetura, marcando assim o peso inicial de cada ligação;
- Iniciado o treino, os valores (vetores entrada) fornecidos aos nodos de input vão-se propagando para a frente, até chegarem aos nodos de saída. Chegando à camada final, é calculado o erro entre o valor obtido pela rede e o valor esperado;
- O referido erro é propagado para as camadas anteriores, desde os nodos de saída até aos aos nodos da camada de *input*. Esta retro propagação leva ao ajustamento dos pesos de cada nodo segundo a regra de *Widrow-Hoff* Equação 2.1;
- Este processo é repetido para cada iteração. A fase de treino termina quando forem cumpridos os critérios de paragem definidos.

3. Implementação RNAs

3.1 Ferramentas desenvolvimento

Tendo sido focado no capitulo anterior os principais conceitos associados com a estrutura, evolução e avaliação do desempenho de uma rede neuronal artificial, neste capítulo serão abordados algumas das ferramentas e plataformas existentes para desenvolver RNAs.

De uma forma geral, para uma inicialização à temática das RNAs, existem soluções bastante práticas, como por exemplo:

- ***SimBrain*** [5]: apresenta-se como uma ferramenta com uma forte componente visual e educacional. As suas principais características encontram-se no facto de ser uma plataforma capaz de ser executada em qualquer sistema operativo e, por disponibilizar a capacidade de visualização da evolução da topologia da rede, ao longo do processo de aprendizagem.

- ***JustNN*** [6]: Também no segmento de ferramentas extremamente intuitivas de usar pelo utilizador final, existe a ferramenta *JustNN*. Permite a definição da arquitetura de uma RNA, assim como testar o seu desempenho após um processo de aprendizagem com um conjunto de dados.

Numa vertente mais aberta ao programador, existe ainda um conjunto diversos de livrarias e *frameworks* para praticamente todos os ambientes e linguagens de programação, como Java, C++ ou Python. Deste universo de soluções, e focadas no contexto atual de desenvolvimento de RNAs, destacam-se as seguintes:

- ***OpenNN*** [7] e ***TensorFlow*** [8]: Apresentam-se como duas soluções capazes de suportar as atuais implementações da RNAs em ambientes extremamente paralelos, como nas unidades de processamento da GPU. A conceção de redes neuronais nestes contextos, permite reduzir o tempo de execução (treino) da rede e aumentar o volume de dados em análise;

  OpenNN é uma plataforma desenvolvida em *C++* e com capacidade para execução paralela em GPU, através da do ambiente CUDA. *TensorFlow* é uma ferramenta voltada para *Deep Machine Learning* e desenvolvida pela *Google Brain Team*, tendo ganho uma forte adesão desde o seu recente surgimento. Esta ferramenta pode ser integrada com as linguagem *Python, C++, Java ou Go*;

- ***R-Studio*** [9]: Através de *packages* como *MXNet* ou *NeuralNet*, é possível construir e gerir redes neuronais artificias sobre a linguagem de programação *R*. Por omissão, os métodos deste *package* baseia-se na técnica de aprendizagem de *Back-Propagation* (Secção 2.4).

- ***Julia*** [10, 11]: Devido às suas características de compilação e execução, têm sido criados na linguagem *julia* um conjunto diverso de *packages*, que permitem explorar RNAs sobre as vantagens de desempenho que a linguagem oferece.

  Os packages *Mocha, MXNet, TensorFlow, ANN e Knet* são alguns dos exemplos que permitem a definição e criação de RNAs neste ambiente. A titulo de exemplo, o package *MXNet* permite a execução da evolução da rede neuronal a nível do contexto da GPU.

- ***Python***: Pela sua expressividade simples e desempenho de execução, *Python* apresenta-se como uma linguagem de programação de alto nível recorrentemente utilizada em contextos de análise de dados.

  Relativamente à exploração de RNAs nesta linguagem, existem packages como *Neupy (Theano), Blocks, Lasagne, Keras, Deepy ou NoLearn*, que permitem a conceção e gestão flexível de uma RNA e do seu processo de treino e teste.

  Alem destas livrarias, ferramentas complexas como *TensorFlow* podem ser integradas e utilizadas dentro do ambiente *Python*.

Embora sejam referidas de forma breve várias soluções disponibilizadas no mercado, será neste artigo realizada apenas um exploração detalhada da implementação de RNAs na linguagem *R* e *Julia*, segundo os packages *NeuralNet* e *MXNet*, respetivamente. A descrição das instruções e cuidados necessários para criar e gerir o processo de aprendizagem segundo estes dois *packages*, assume o tratamento prévio dos conjuntos de dados e um cenário de aprendizagem supervisionado. Deste modo, existe um dataset para a fase de treino e um outro dataset para teste e analise de sensibilidade da RNA treinada.

3.2 Conceção RNAs em *R*

Sendo a linguagem R voltada para a análise de dados, a utilização de alguns *packages* torna-a ideal para a definição, treino e teste de redes neuronais artificiais. Um dos *packages* mais utilizados neste sentido é designado por *NeuralNet*.

Este package utiliza o algoritmo de *Back-Propagation* [12], permitindo ainda escolher a função de ativação dos neurónios e a função de erro, para avaliação do desempenho da rede.

3.2.1 Definição e treino da rede

Nesta abordagem, a fase inicial de conceção de uma RNA começa pela criação da sua topologia, conforme uma determinada arquitetura. Para tal, o conjunto de nodos de *input* e *output* são definidos através de uma fórmula. Dado que na maioria dos casos, os dados de treino da rede neuronal provêm de um dataset, o lado direito desta fórmula agrupa o conjunto de atributos desse dataset que são utilizados para alimentar a rede. Assim, o número de atributos à direita do caractere ~ indica o número de nodos de entrada -*input*-.

De forma semelhante, o lado esquerdo da fórmula agrupa o conjunto de atributos a prever. O número de variáveis no lado esquerdo do caractere ~ indica assim o número de neurónios de saída e qual o atributo por eles previsto.

Na linguagem R, a definição de uma variável para guardar esta fórmula com os nodos de entrada e saída segue o seguinte formato:

```
formula <- Out-Node1 + ... + Out-Node_N ~ In-Node_1 + ... +
↪  In-Node_N
```

Tendo definido os nodos das camadas das periferias da rede, através da fórmula anterior, o processo de treino da rede é conseguido através da função *neuralnet*, existente no package com o mesmo nome. Para tal, é assumido que previamente foi criada uma variável referente ao dataset de treino.

```
rna <- neuralnet(formula, data, hidden=c(3,4,3), ..., paramN)
```

Nesta função, e como forma de definir alguns aspetos da topologia e aprendizagem das RNA, destacam-se os seguintes argumentos:

- ***formula***: Descrição simbólica das camadas de *input* e *output* da rede, através da fórmula anterior;
- ***data***: Dataset utilizado para o processo de treino da rede, cujos atributos devem ser especificados na fórmula;
- ***hidden***: Vetor de inteiros onde se define a topologia interna ("escondida") da rede. O tamanho do vetor indica o número de camadas intermédias e o valor de cada índice **i** indica número de nodos dessa camada **i**;
- ***threshold***: Indica o limite mínimo sobre o qual a rede deve convergir;
- ***stepmax***: Limita o número de iterações máximo que a rede pode executar. Acima deste valor, considera-se que a rede não converge em tempo aceitável, tendo que ser exploradas novas topologias ou processos de aprendizagem.
- ***lifesign***: String que especifica quanta informação a rede apresenta no terminal, ao longo da sua evolução.
- ***learningrate***: valor numérico a atribuir ao parâmetro Learning rate do algoritmo Back-Propagation;
- ***algorithm***: String que permite definir o algoritmo a utilizar no processo de aprendizagem. Independente da escolha, o processo será sempre uma variante com base nos conceitos de *Back-Propagation*.
- ***err.fct***: Permite definir a função utilizada para medir o erro de desempenho da rede. Por omissão, se for utilizada a *string "sse"* mede o erro através do somatório dos erros ao quadrado (*sum of squared erros*) e, com a *string "ce"* utiliza a função de entropia cruzada (*cross-entropy*).

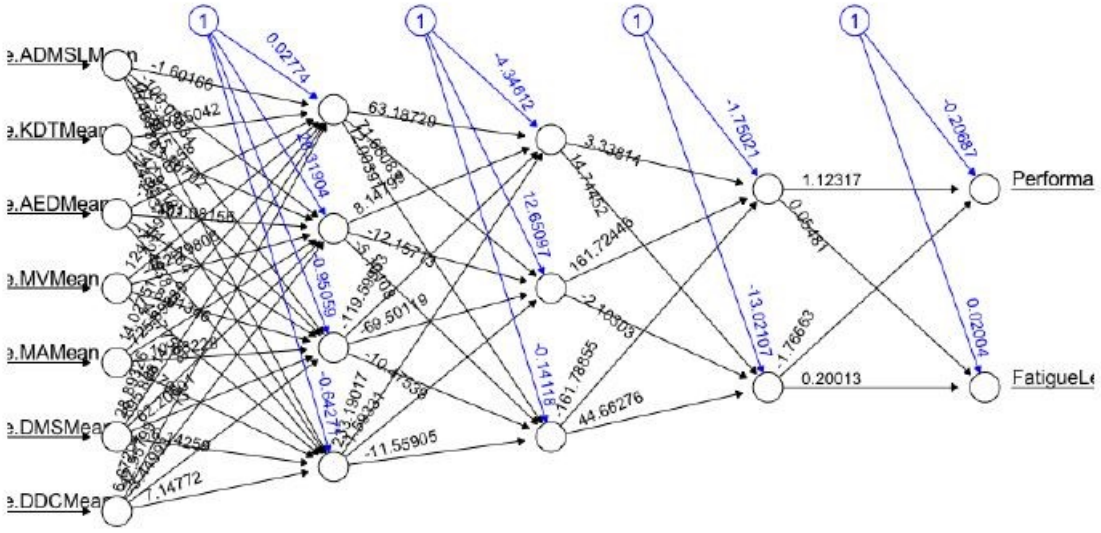


Figura 3.1: Exemplo do plot de uma rede com 7 nodos de input, 3 camadas intermédias e 2 nodos de output.

- ***act.fct***: Permite definir a função utilizada para mapear o resultado do integrador para um valor suave, que será o valor de saída do neurónio. Por omissão podem ser passadas neste argumento as *strings "logistic"ou "tanh"* para serem utilizadas, respetivamente, as funções logística ou tangente hiperbólica. Ambas estas funções enquadram-se nas funções de ativação *sigmoide*.

A visualização da estrutura da rede pode ser feita utilizando a função *plot* existente em R (Figura 3.1). No sentido de controlar a forma de visualização da rede, como as cores dos nodos, visibilidade dos pesos das ligações ou nomes dos nodos, a função *plotnet* do *package neuralNetTools* apresenta-se como uma solução de visualização da topologia da rede mais completa e detalhada a nível de opções.

3.2.2 Avaliação desempenho rede

Uma vez construida e treinada a rede, para um determinado conjunto de dados, é imprescindível realizar uma análise no erro de previsão do mesmo. Em contexto de aprendizagem supervisionada, esta avaliação do erro entre os valores esperados e obtido é realizado através de um dataset de teste, com casos não utilizados na fase de treino.

Para isso é utilizada a função *compute*, também disponível no *package NeuralNet*, que recebe no primeiro argumento a variável com a rede neuronal treinada e, no segundo argumento, recebe o dataset de teste.

```
# testar a previsão da rede rna, com todas as as linhas do
↪  dataset 'teste'
predicted.nn.values <- compute(rna, teste)

# aplicar a função round a todos os valores do atributo
↪  net.result
# passagem do argumento 'digits=2' para a função round, p/
↪  arredondar a 2 casas decimais
predicted.nn.values <-
```

```
    as.data.frame(sapply(predicted.nn.values$net.result,
    ↪  round, digits=2))

# calculo do mean square error
MSE.NN <-
    sum(teste$Avaliacao - predicted.nn.values$net.result)^2 /
    ↪  nrow(teste)
# Ou calculo RMSE
RMSE.NN <-
    rmse( c(teste$Avaliacao),
    ↪  c(predicted.nn.values$net.result) )
```

Os valores previstos pela RNA são guardados numa variável do tipo *List*, semelhante a um *array* tradicional. Comparando esta estrutura devolvida com a coluna correspondente aos valores esperados para as instâncias do dataset de teste, pode ser avaliada a sensibilidade da rede.

Desta forma, a qualidade do desempenho da rede pode ser medida através de qualquer métrica, sendo esta fase independente do *package* de redes neuronais utilizado. Das métricas mais utilizadas destaca-se o calculo do *Mean Squared Error* (MSE) ou, de forma semelhante, o *Root Mean Squared Error* (RMSE).

3.3 Conceção RNAs em *Julia*

De forma semelhante ao R, a linguagem *Julia* apresenta-se como uma linguagem de programação de alto nível, voltada para a computação de dados numéricos e científicos.

Apesar de ser uma linguagem bastante recente, a adesão à mesma tem sido cresceste, consequência da sua excelente performance, capacidade de paralelismo e computação distribuída. Devido às suas características e às linguagens em que foi escrita, *Julia* apresenta tempos de execução próximos de programas em C nativo, justificando assim a sua utilização em diversos contextos.

Apesar de existir uma grande variedade de *packages* [11] e APIs para integrar soluções de outros ambientes nesta linguagem, nesta secção será descrita a criação e evolução de uma RNA utilizando o package *MXNet*.

3.3.1 Definição e treino da rede

Independente do tipo de dados que sejam fornecidos para treinar a rede (imagens, texto, valores numéricos, ...) é necessário envolver o dataset num formato padrãom sendo para isso utilizados *data providers*. Estas estruturas funcionam como *wrappers* dos dados, sendo capazes de os dividir em conjuntos de menores dimensões, utilizando para alimentar a rede de forma uniforme.

Nos argumento da função *ArrayDataProvider*, deve ser passado o conjunto de dados de treino, o conjunto de previsões esperadas e marcada a opção de *shuffle* como *TRUE*, para que as instâncias dos dados sejam baralhadas, evitando assim a criação de eventuais tendências no processo de aprendizagem. Deste modo, é criado um *data provider* tanto para os dados de treino com para os dados de teste.

Adaptado o conjunto de dados, são criadas duas variáveis simbólicas associadas a um determinado nome.

```
treino = ...
teste  = ...

trainprovider = mx.ArrayDataProvider(:data => treino,
 ↪  batch_size=batchsize, shuffle=true, :label => teste)
evalprovider = mx.ArrayDataProvider(:data => treino,
 ↪  batch_size=batchsize, shuffle=true, :label => teste)

data = mx.Variable(:data)
label = mx.Variable(:label)
```

Uma vez tratada a forma dos dados, é definida a topologia da rede. Para isso, é utilizada uma cadeia de comandos iniciada pela macro *mx.chain*, sendo cada conjunto de instruções associadas a uma das camadas da rede.

```
net = @mx.chain   mx.Variable(:data) =>

                  mx.FullyConnected(num_hidden=2) =>
                  mx.Activation(act_type=:tanh) =>

                  mx.FullyConnected(num_hidden=2) =>
                  mx.Activation(act_type=:tanh) =>

                  mx.FullyConnected(num_hidden=1) =>
                  mx.LinearRegressionOutPut(label)
```

No exemplo de código anterior:

- A instrução *mx.Variable* define a camada de input da rede, existindo tantos nodos quantos atributos na variavel com o dataset de treino;
- A instrução *mx.FullyConnected* permite definir uma camada intermédia, sendo no argumento especificado o número de nodos dessa camada. Tal como o nome indica, todos os nodos desta camada estão conectados com os nodos da camada anterior;
- A instrução *mx.Activation* permite definir o função utilizada na fase de ativação do neurónio. Por omissão, existem predefinidas as funções: tangente hiperbólica *tanh*, *sigmoide* (*sigmoid*) ou a *rectified linear unit (ReLU)* (Figura 2.1).
- A adição de novos pares de instruções *mx.FullyConnected* e *mx.Activation* permite a criação de mais camadas intermédias com uma determinada função de ativação;
- Por fim, na camada de saída é apenas especificado o número de nodos da mesma, conforme o contexto em que se aplica a rede. A última instrução define a função a aplicar ao output final.

  No exemplo, foi utilizada a função linear *(mx.LinearRegressionOutPut)*, embora existam também as funções Logística (*mx.LogisticRegressionOutput*) e de erro absoluto *(MAERegressionOutput)*. A função Logística é adequada para contextos de classificação binária ou tarefas de previsão probabilística.

Definida a arquitetura da rede, é especificado o tipo de rede e o seu contexto de execução. Uma das mais valias deste *package MXNet* recai exatamente no facto de permitir que a RNA possa ser executada tanto a nivel do CPU como da GPU, sem que para isso o código de conceção da mesma tenha que ser alterado.

No exemplo abaixo, a rede segue a arquitetura *FeedForward* multi-camada (Secção 2.3) e é executada a nível do CPU.

```
model = mx.FeedForward(net, context=mx.cpu())
```

Antes de realizar o treino da rede, devem ainda ser definidos o algoritmo de otimização, a métrica de avaliação de desempenho e a forma de inicialização dos pesos das ligações.

```
optimizer = mx.SGD(lr=0.01, momentum=0.9,
↪  weight_decay=0.00001)

eval_metric = mx.MSE()

init_weights = mx.XavierInitializer(distribution = mx.xv_avg,
↪  magnitude = 1)
```

Nas instruções anteriores:

- A variável *optimizer* define a técnica de aprendizagem utilizada, neste caso o Método do Gradiente Descendente Estocástico (Stochastic gradient descent - SGD). Neste paradigma são indicados valores para a taxa de aprendizagem (*learning rate*), o *momentum* e a taxa de decaimento.

  Geralmente, até se obter sucesso na previsão da rede após um processo de aprendizagem, é necessário realizar a regulação destes parametros, repetindo o processo de treino da rede. O package oferece ainda outras técnicas de aprendizagem, que seguem o mesmo processo de aprendizagem baseado em gradientes. [13];

- A variável *eval_metric* indica a forma de avaliar a performance na fase de treino da RNA criada. Neste caso é utilizada a métrica *Mean Squared Error* sendo que a maioria das outras métricas disponibilizadas baseiam-se em medidas semelhantes.

  Neste parâmetro pode também ser utilizada a opção "accuracy"ou, tal como na linguagem R, ser definida uma função pelo programador para definir de que forma se deve avaliar o desempenho da rede;

  No caso de no processo de treino ser também passado um dataset de teste, para avaliar posteriormente o desempenho da RNA após treino, a mesma métrica será utilizada nessa avaliação.

- A variável *init_weights* define qual o método utilizado para gerar o valor inicial dos pesos das ligações. Para isso podem ser utilizadas funções como a distribuição normal, com uma dada média e desvio (*MXNet.mx.NormalInitializer*); a distribuição uniforme, com uma determinada escala *(MXNet.mx.UniformInitializer)* ou, neste caso, a distribuição de *Xavier* [14, 15].

Por fim, para realizar efetivamente o treino da rede, é utilizado o método *fit* ou o método *train* (*alias* para a função *fit*).

```
mx.fit(model, optimizer, trainprovider, n_epoch=50, eval_data
 ↪  = evalprovider, eval_metric, initializer = init_weights )
```

Dos vários argumentos desta função, destacam-se:

- ***model***: variável com a RNA a ser treinada;
- ***optimizer***: algoritmo de aprendizagem a utilizar;
- ***trainprovider***: dataset com as instâncias de treino;
- ***n_epoch***: número de passagens completas pelos dados. Por omissão = 10;
- ***eval_data***: argumento opcional. Usado para passar um dataset com instâncias de teste da rede, sendo assim utilizado em cenários de aprendizagem supervisionada;
- ***eval_metric***: métrica utilizada para avaliar a performance de treino da rede. Se *eval_data* não for vazio, a mesma métrica é usada na fase de teste;
- ***initializer***: técnica usada para inicializar os pesos das ligações.

3.3.2 Avaliação desempenho rede

De forma a determinar o desempenho de previsão da rede, perante um dataset de novas instâncias, é utilizada a função *predict*. Esta função recebe como argumentos a estrutura com a RNA treinada e o dataset com os dados a utilizar nesta fase de teste.

A função devolve o equivalente a um *array* na linguagem *Julia*. Por esse motivo, e tal como se verifica na avaliação de uma RNA dentro da linguagem R (Secção 3.2.2), é possível calcular qualquer métrica de erro, independente do *package*, topologia de rede ou forma de treino utilizada. Para tal basta comparar os valores devolvidos pela função *predict* com a coluna do dataset de teste que contem os valores esperados para cada instância.

```
//Fase de avaliação da RNA, com novas instâncias usadas
 ↪  exclusivamente para teste
predicted = predict(model, teste)

// Calculo do root mean squared error, com os valores
 ↪  predicted e esperados
rmse  = sqrt(mean((predicted - esperados)^2))
```

Novamente, outras métricas podem ser calculas, bastando para isso realizar as respetivas operações matemáticas sobre as diferenças entre as colunas com os valores previstos e esperados.

3.4 Resultados Comparativos

Dos módulos apresentados, tanto o package *neuralnet* para a linguagem R como o package *MXNet*, para a linguagem Julia, apresentam funcionalidades muito semelhantes. Ambos permitem a criação de uma RNA dando liberdade ao programador para definir a sua arquitetura, o processo de aprendizagem e métricas de avaliação interna da rede.

Relativamente à utilização do *package neuralnet*, assinala-se como desvantagem o facto deste módulo só permitir processos de aprendizagem baseados na técnica de *Back-Propagation*. Apesar de atualmente este algoritmo ser, de facto, um dos mais utilizados no processo de treino de uma rede, o uso de outras técnicas pode também ser relevante e interessante de explorar.

Outra limitação deste *package*, identifica-se no facto de não ser possível recorrer à GPU para executar o processo de treino ou teste de uma RNA. Dado que o paradigma atual de exploração de RNAs foca-se em tirar proveito das capacidades da GPU, este aspeto limita o alcance de processamento de RNAs nesta linguagem Além disto, a função de ativação fornecida é aplicada a todos os nodos da rede, não permitindo que a rede contenha nodos com outro tipo de função de ativação.

Em oposição, no *package MXNet* é possível realizar a evolução da RNA a nível do contexto da GPU. A função de ativação pode também ser definida por camada, permitindo assim que diferentes camadas em diferentes níveis da rede possam ter funções de ativação distintas.

Contudo, a implementação de RNAs em R apresenta uma estrutura mais simples e com uma documentação mais extensa e detalhada. Esta linguagem é também fácil de compreender, compilar e executar dentro do ambiente R-Studio, sendo assim um excelente ponto de partida para indivíduos sem bases de programação.

Implementações sobre Julia, enquadram-se em contextos mais científicos e com datasets de grandes dimensões para análise. Nestes cenários, justifica-se o ligeiro aumento de complexidade da implementaçºão de RNAs em julia, para que seja possível tirar partido da eficiência de processamento e capacidades da linguagem, como a execução a nível da GPU.

A nível de funções para a visualização ambas as linguagens disponibilizam um diverso conjunto de packages que dão resposta a estes pedidos. De igual forma, a nível da avaliação do desempenho da rede, os resultados previstos pela RNA ficam armazenados na forma de uma lista de valores. É assim possível, em qualquer uma das linguagem e recorrendo a operações sobre matrizes ou listas, comparar a diferença entre os valores esperados e os valores obtidos pela previsão da rede, segundo uma determinada métrica de avaliação.

4. Conclusão

No presente relatório foram descritos um conjunto de fundamentos base, necessários para compreender a temática de Redes Neuronais Artificiais. Além de uma breve análise teórica, são ainda apresentadas algumas soluções e plataformas que permitem o desenvolvimento destas estruturas, possíveis de ajustar e aplicar nos mais diversos contextos atuais.

A introdução teoria sobre RNAs torna-se necessária para perceber os principais algoritmos e paradigmas de aprendizagem utilizados atualmente. A partir desta contextualização, é possível compreender mais facilmente alguns dos argumentos que são necessários de fornecer às funções que constroem e treinam RNAs.

Independente da linguagem de programação, existem atualmente diversas livrarias externas que permitem a implementação e gestão de RNAs, em diversos contextos de execução. O elevado número de plataformas e módulos permite observar o relevo e importância que técnicas de machine learning sobre RNAs têm ganho atualmente. No geral, praticamente todas as soluções oferecem o mesmo tipo de funcionalidades, permitindo de forma flexível ao programador criar uma arquitetura, visualizar a rede, definir o paradigma de aprendizagem ou realizar o treino e teste da RNA.

Dos módulos apresentados, tanto o package *neuralnet* para a linguagem R como o package *MXNet*, para a linguagem Julia, apresentam características muito semelhantes. Contudo, o package MXNet merece destaque pelo facto de permitir gerir e alterar um maior número de parâmetros internos a nível da arquitetura da rede e paradigma de aprendizagem.

Das duas linguagens abordadas neste relatório, a implementação de RNAs na linguagem Julia apesar de ligeiramente mais complexa, foca-se naquilo que é atualmente o contexto de exploração de RNAs. Tal como na plataforma *TensorFlow*, o módulo *MXNet* integrado com Julia permite tirar partido das capacidades massivamente paralelas da GPU, realizando a execução do treino de uma RNA nas unidades de processamento gráficas. Esta linguagem, além de extremamente eficiente por si só, apresenta assim uma alteração extremamente simples do contexto de execução, sem que para isso seja necessário alterar as funções e a forma como se define uma RNA para executar a nível da GPU. Alem disto, este *package* está disponível para outras linguagens de programação e tem um maior leque de opções para gerir e definir parâmetros de aprendizagem da rede.

Ainda assim, a linguagem R permite uma abordagem simples, rápida e prática para implementar redes neuronais artificias, sendo um ambiente sólido e devidamente documentado para explorar a conceção e desenvolvimento destas estruturas.

Em suma, apesar da qualidade de um processo de treino de uma RNA estar intrinsecamente ligado com os parâmetros e técnicas utilizadas para a fase de aprendizagem, o uso de uma linguagem e *packages* apropriados permite simplificar significativamente o acesso a esta personalização e parâmetros internos de uma rede neuronal artificial.

Bibliografia

[1] Paulo Cortez e José Neves. Redes neuronais artificiais. *IEEE Computer Graphics and Applications*, 2000.

[2] Selva Staub, Emin Karaman, Seyit Kaya, Hatem Karapınar, and Elçin Güven. Artificial neural network and agility. *Procedia - Social and Behavioral Sciences*, pages 1477 – 1485, 2015. World Conference on Technology, Innovation and Entrepreneurship.

[3] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1998.

[4] Vinicius Goncalves Maltarollo, Kathia Maria Honorio, and Alberico Borges Ferreira da Silva. Applications of artificial neural networks in chemical problems. 2013.

[5] Simbrain. `http://simbrain.net/`. [Online; accessed 07-Março-2018].

[6] Justnn. `http://www.justnn.com/`. [Online; accessed 07-Março-2018].

[7] Opemnn. `http://www.opennn.net/`. [Online; accessed 07-Março-2018].

[8] Tensorflow. `https://www.tensorflow.org/`. [Online; accessed 07-Março-2018].

[9] R-studio - neuralnet. `https://cran.r-project.org/web/packages/neuralnet/neuralnet.pdf`. [Online; accessed 07-Março-2018].

[10] Julia. `https://julialang.org/`. [Online; accessed 07-Março-2018].

[11] Machine learning packages for julia. `https://github.com/svaksha/julia.jl/blob/master/AI.md#neuranetworks`. [Online; accessed 07-Março-2018].

[12] Frauke Günther and Stefan Fritsch. *neuralnet: Training of Neural Networks*. 2010.

[13] Mxnet optimizers. `http://dmlc.ml/MXNet.jl/latest/api/optimizer/`. [Online; accessed 09-Março-2018].

[14] Mxnet initializers. `http://dmlc.ml/MXNet.jl/latest/api/initializer/`. [Online; accessed 09-Março-2018].

[15] Xavier Glorot and Yoshua Bengio. *Understanding the difficulty of training deep feedforward neural networks*. 2010.