Redes Neuronais Artificiais

Conceitos e definições

Redes feedforward

Implementação em python

Analogia: aprendizagem natural

O cérebro é uma estrutura altamente complexa, **não linear** e **paralela**

Possui uma capacidade de organizar os seus neurónios, de modo a executarem tarefas complexas

Um neurónio é 5/6 vezes mais lento do que uma porta lógica

O cérebro ultrapassa a lentidão através de uma estrutura paralela

O córtex humano possui 10 biliões de neurónios e 60 triliões de sinapses !!

O que são redes neuronais

As redes neuronais são modelos de aprendizagem máquina que seguem uma analogia com o funcionamento do cérebro humano

Uma rede neuronal é um **processador paralelo**, composto por unidades simples de processamento (neurónios)

O conhecimento é armazenado nas conexões entre os neurónios

O conhecimento é adquirido a partir de um ambiente (dados), através de um processo de **aprendizagem** (algoritmo de treino) que ajusta os pesos das conexões

Benefícios / razões para o sucesso

Aprendizagem/generalização – permite adquirir novo conhecimento do ambiente

Processamento maciçamente paralelo - permitem que tarefas complexas sejam realizadas num curto espaço de tempo

Não linearidade - útil para muitos problemas reais

Adaptabilidade - podem adaptar a sua topologia de acordo com mudanças do ambiente

(…)

Benefícios / razões para o sucesso

Robustez e degradação suave - capazes de ignorar o ruído e atributos irrelevantes; lidam com informação incompleta de forma eficiente

Flexibilidade - grande domínio de aplicabilidade

Usabilidade - podem ser utilizadas como "caixas negras"; não é necessário um conhecimento explícito da função a ser aprendida

Neurónios artificiais

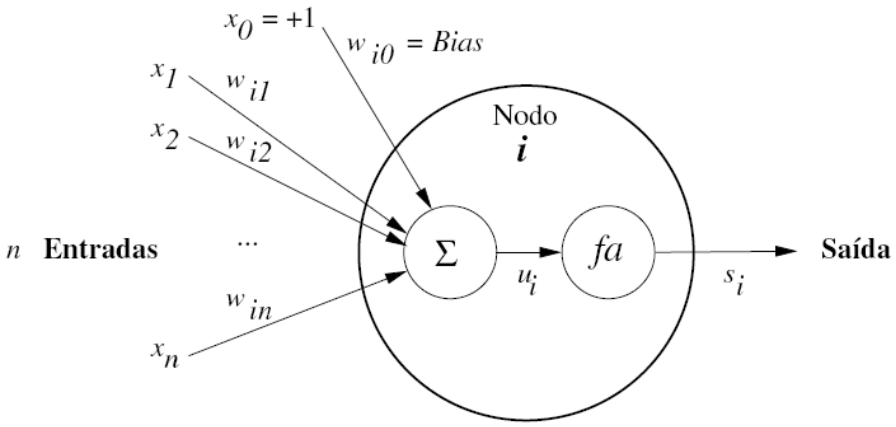
Recebem um conjunto de entradas (de dados ou de conexões)

Um peso (valor numérico) associado a cada conexão

Cada neurónio calcula a sua activação com base nos valores de entrada e nos pesos das ligações

O sinal calculado é passado para a saída depois de filtrado por uma função de activação

Estrutura de um neurónio



Funções de activação

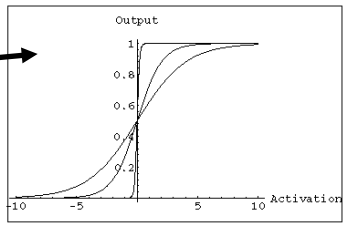
Sigmoid/ logistica

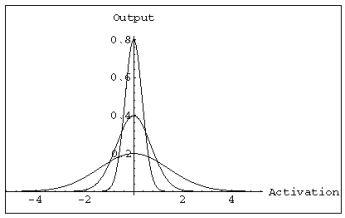
Linear

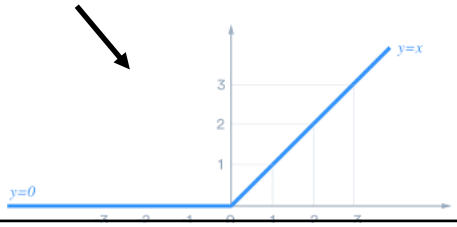
Tangente hiperbólica (Tanh)

Gaussiana

RelU (rectified linear)





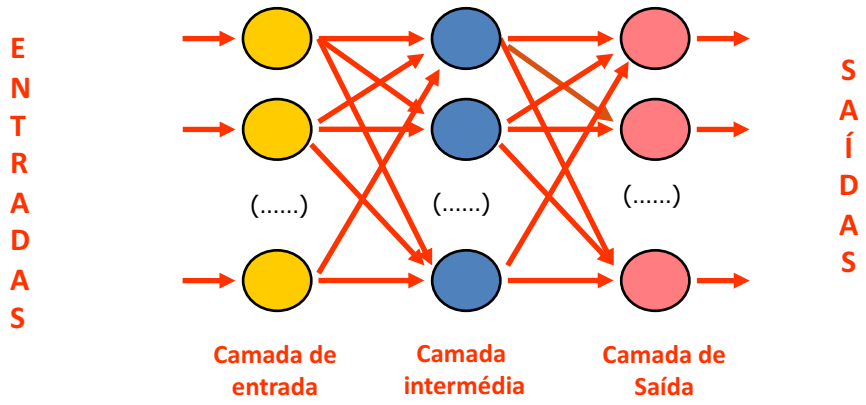


Topologias de rede

Arquitectura (ou topologia) - a forma como os nodos se interligam numa estrutura de rede

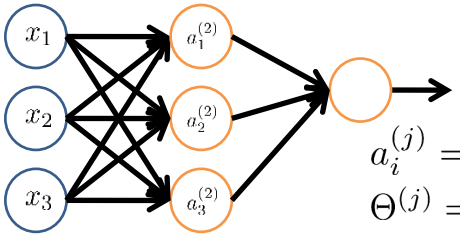
Existem inúmeros tipos de arquiteturas cada um com as suas potencialidades, caindo em duas categorias: supervisionadas e não supervisionadas

Topologia feedforward



Redes com várias camadas intermédias são chamadas de **Deep Neural Networks**
Com uma camada intermédia – **Multilayer perceptrons (MLPs)**

Representação da rede



$a_i^{(j)}$ = "ativação" do neurónio *i* na camada *j*

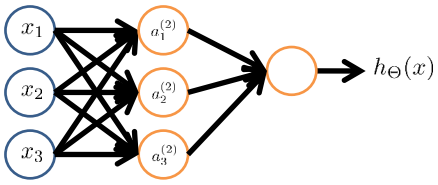
$\Theta^{(j)}$ = matriz de pesos entre neurónios das camadas *j* e *j+1* (linhas – destino, colunas - origem)

sigmoid =1

$$a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3)$$
$$a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3)$$
$$a_3^{(2)} = g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3)$$
$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})$$

=1

Computação da saída: vetorizada



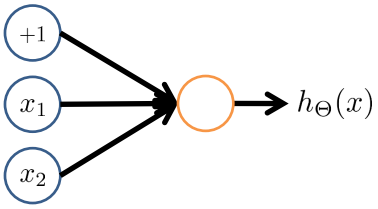
$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad z^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix}$$

$$z^{(2)} = \Theta^{(1)}x$$
$$a^{(2)} = g(z^{(2)}) \longrightarrow$$ Valores de ativação da camada intermédia

Adicionar $a_0^{(2)} = 1$

$$z^{(3)} = \Theta^{(2)}a^{(2)}$$
$$h_\Theta(x) = a^{(3)} = g(z^{(3)}) \longrightarrow$$ Valores de saída da rede

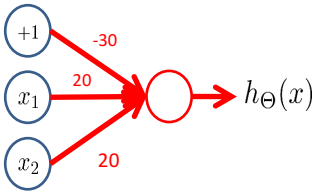
Exemplos



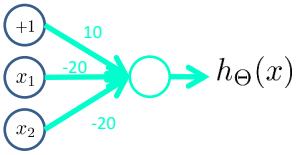
x_1	x_2	$h_\Theta(x)$
0	0	
0	1	
1	0	
1	1	

Calcular o valor de saída para os casos dos 2 pesos:
- 30, 20, 20
-10, 20, 20
10, -20, -20

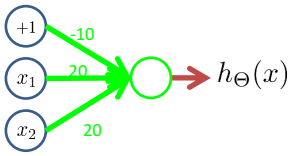
Exemplos



x_1 AND x_2



(NOT x_1) AND (NOT x_2)



x_1 OR x_2

Como fazer h(x1, x2) = x1 XNOR x2 com uma rede com uma camada intermédia ?

Note que: x1 XNOR x2 = (x1 AND x2) OR (NOT x1 AND NOT x2)

Treino: aprendizagem supervisionada

Dados: **exemplos de treino** constituídos por **entradas** e respectivas **saídas desejadas**;

Objectivo: fixar os valores dos pesos das conexões que minimizem a função de custo: no caso das RNs, generalização da função de custo da regressão logística

Vários algoritmos de treino baseados no gradiente descendente

- O mais usado o *backpropagation*
- Outros algoritmos: *Marquardt-Levenberg*, Rprop, Quickprop, etc.

Algoritmo *Back-propagation*

Baseia-se no vector gradiente da superfície do erro que define a direcção de descida máxima – método similar ao gradiente descendente

Parâmetro importante: taxa de aprendizagem - define a distância que se caminha

Uma sequência destes movimentos leva a um mínimo (espera-se que global)

O treino decorre um dado número de *épocas*: define o nº de vezes que cada caso é treinado pela rede, sendo os exemplos tipicamente divididos em *batches* (sub-conjuntos de exemplos)

Configuração inicial da rede (pesos das ligações) é gerada aleatoriamente

Critérios de paragem: nº fixo de épocas, tempo ou critérios de convergência baseados num sub-conjunto de exemplos de validação

Fases do algoritmo Backpropagation

Propagação frontal – calculado o valor de saída para o vector de entradas e o erro cometido;

Retropropagação – dado o erro cometido este é propagado para trás, ajustando-se os pesos das conexões no sentido da sua diminuição. Baseia-se no calculo do gradiente usando a regra de cálculo das derivadas parciais em cadeia (chain rule) para funções compostas

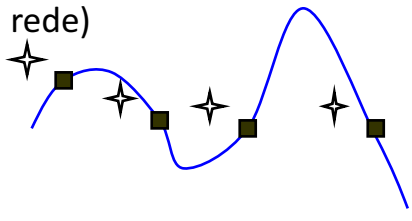
Generalização / overfitting

Treinar demais a rede pode impedir generalização por *overfitting* – a rede memoriza os casos de treino e não regras de generalização – pode-se parar o treino antecipadamente !

Pode usar-se regularização de forma semelhante a regressão linear/ logística (nas redes tem o nome de decay)

A probabilidade de overfitting aumenta se:

- Poucos casos de treino (qualidade da amostra)
- Muitas ligações (complexidade da rede)



Pré-processamento dos dados

Aplicam-se as técnicas discutidas no âmbito mais genérico dos modelos funcionais – técnica de regularização mais comum - *decay*

Normalização nas RNAs depende do domínio e contra-domínio das funções de activação

Conveniente verificar a distribuição estatística dos valores de cada atributo.

Interpretação das saídas: problemas de classificação

Se usarmos modelos funcionais para problemas de classificação, teremos que converter a saída do modelo (valor numérico) num valor do atributo de saída desejado (nominal), ou seja, na classe prevista.

Poderemos optar por cada uma das duas hipóteses anteriormente referidas: um neurónio dividindo o domínio ou 1-of-C / one-hot encoding

No último caso do 1-of-C, teremos M saídas numéricas (1 por classe), sendo normalmente escolhida a classe correspondente ao maior valor. Neste caso, poderemos também calcular facilmente probabilidades para cada classe

Escolha da topologia para uma RNA *Feedforward (MLP*

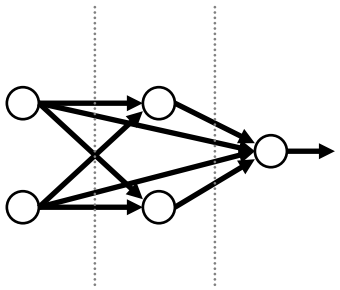
Quantos nodos de entrada e saída?

Quantas camadas e nós intermédios?

Como ligar os neurónios?

Ligações de atalho?

Modelo mais simples: Redes *Feedforward com camadas completamente interligadas*



Implementação em python - exercício

Definir uma classe MLP que implementa uma rede neuronal com uma camada intermédia, com os seguintes **atributos**:

- ***X, y*** – retirados do conjunto de dados (tal como nos casos da regressão)
- ***h*** – número de neurónios na camada intermédia
- ***W1***– matriz de pesos ligando a camada de entradas à camada intermédia (matriz de dimensões h por n+1 – onde n = número de inputs)
- ***W2*** – matriz de pesos ligando a camada intermédia à saída única (matriz de dimensões 1 por h+1

Métodos:

- ***predict*** – prever saída para um exemplo
- ***costFunction*** – calcular erro para um conjunto de exemplos (função de erro igual à definida na regressão linear)
- ***buildModel*** – treino da rede neuronal

Implementação em python - numpy

Construtor

```
class MLP:

  def __init__(self, dataset, hidden_nodes = 2):
      self.X, self.y = dataset.getXy()
      self.X = np.hstack ( (np.ones([self.X.shape[0],1]), self.X ) )
      self.h = hidden_nodes
      self.W1 = np.zeros([hidden_nodes, self.X.shape[1]])
      self.W2 = np.zeros([1, hidden_nodes+1])
```

Implementação em python - numpy

Previsão da saída para um exemplo

```
def  predict( self,  instance):
     ...
```

Implementação em python - numpy

Previsão da saída para um exemplo

```
def predict( self, instance):
    x = np.empty([self.X.shape[1]])

    x[0] = 1
    x[1:] = np.array(instance[:self.X.shape[1]-1])
    z2 = np.dot(self.W1, x)

    a2 = np.empty([z2.shape[0]+1])
    a2[0] = 1
    a2[1:] = sigmoid(z2)
    z3 = np.dot(self.W2, a2)

    return sigmoid ( z3 )
```

Implementação em python - numpy

Função de erro (cost function) – assume soma do quadrado dos erros

```
def costFunction(self):
    ...
```

Implementação em python - numpy

Função de erro (cost function) – assume soma do quadrado dos erros

```
def costFunction(self):
    m = self.X.shape[0]
    Z2 = np.dot(self.X, self.W1.T)
    A2 = np.hstack ( (np.ones([Z2.shape[0],1]), sigmoid(Z2) ))
    Z3 = np.dot(A2, self.W2.T)
    predictions = sigmoid(Z3)
    sqe = (predictions- self.y.reshape(m,1)) ** 2
    res = np.sum(sqe) / (2*m)
    return res
```

Implementação em python - numpy

Exemplo - XNOR

```
def  setWeights(self, w1, w2):
    self.W1 = w1
    self.W2 = w2
```

```
def test():
   ds= Dataset("xnor.data")
   nn = MLP(ds, 2)
   w1 = np.array([[-30,20,20],[10,-20,-20]])
   w2 = np.array([[-10,20,20]])
   nn.setWeights(w1, w2)
   print( nn.predict(np.array([0,0]) ) )
   print( nn.predict(np.array([0,1]) ) )
   print( nn.predict(np.array([1,0]) ) )
   print( nn.predict(np.array([1,1]) ) )
   print(nn.costFunction())

test()
```

Implementação em python - numpy

Treino do modelo – usa métodos de otimização do scipy

```
def costFunction(self, weights = None):
    if weights is not None:
        self.W1 = weights[:self.h * self.X.shape[1]].reshape([self.h, self.X.shape[1]])
        self.W2 = weights[self.h * self.X.shape[1]:].reshape([1, self.h+1])

(…)
```

```
def build_model(self):
    from scipy import optimize
    size = self.h * self.X.shape[1] + self.h+1
    initial_w = np.random.rand(size)
    result = optimize.minimize(lambda w: self.costFunction(w), initial_w, method='BFGS',
                    options={"maxiter":10000, "disp":False} )
    weights = result.x
    self.W1 = weights[:self.h * self.X.shape[1]].reshape([self.h, self.X.shape[1]])
    self.W2 = weights[self.h * self.X.shape[1]:].reshape([1, self.h+1])
```

Implementação em python - numpy

Exemplo - XNOR

```
def  test():
    ds=  Dataset("xnor.data")
    nn = MLP(ds,  5)
    nn.build_model()
    print(  nn.predict(np.array([0,0])  )  )
    print(  nn.predict(np.array([0,1])  )  )
    print(  nn.predict(np.array([1,0])  )  )
    print(  nn.predict(np.array([1,1])  )  )
    print(nn.costFunction())

test()
```

Implementação em python – scikit-learn

Classificação - XNOR

```
from sklearn.neural_network import MLPClassifier

X = [[0., 0.], [0., 1.], [1., 0.], [1., 1.]]
y = [1, 0, 0, 1]

mlp = MLPClassifier(solver='lbfgs', hidden_layer_sizes=(6,) )
mlp.fit(X, y)

preds = mlp.predict([[0., 0.], [0., 1.], [1., 0.], [1., 1.]])
print(preds)
```

Implementação em python – scikit-learn

Classificação - digits

```
from sklearn.model_selection import cross_val_score
from sklearn import datasets
from sklearn.preprocessing import StandardScaler

digits = datasets.load_digits()

mlp_dig = MLPClassifier(solver='lbfgs', hidden_layer_sizes=(50,))

scores = cross_val_score(mlp_dig, digits.data, digits.target, cv = 5)
print(scores.mean())
```

Implementação em python – scikit-learn

Classificação – *digits* com normalização

```
from sklearn.preprocessing import StandardScaler

mlp_dig = MLPClassifier(solver='lbfgs', hidden_layer_sizes=(50,))

scaler = StandardScaler()
scaler.fit(digits.data)
scaled_digits = scaler.transform(digits.data)

scores = cross_val_score(mlp_dig, scaled_digits, digits.target, cv = 5)
print(scores.mean())
```

Implementação em python – scikit-learn

Regressão

```
from sklearn.neural_network
import MLPRegressor

diabetes = datasets.load_diabetes()

mlp_diab = MLPRegressor(solver = "lbfgs", hidden_layer_sizes=(20,) )

scores = cross_val_score(mlp_diab, diabetes.data, diabetes.target, cv = 5)
print(scores.mean())
```