Trabalho #2 - Classificação binária de pontos no plano com RNA rasa

Nese trabalho iremos construir uma RNA rasa com uma única camada intermediária para classificar pontos de duas classes diferentes no plano. O objetivo desse trabalho é entender como funciona uma RNA e o seu treinamento usando o método do Gradiente Descendente.

Nesse trabalho você irá apreender como:

- Implementar uma RNA rasa para realizar classificação binária
- Calcular a função de custo logistica (entropia cruzada)
- Implementar a propagação para frente e a retro-propagação usando uma codificação simples sem vetorização nos exemplos

Esse trabalho é uma adaptação de Andrew Ng (deeplearning.ai)

Coloque os nomes e RAs dos alunos que fizeram esse trabalho

Nome e número dos alunos da equipe:

Aluno 1: 20.83992-8 Igor Amaral Correa

Aluno 2:

1 - Pacotes

Em primeiro lugar é necessário importar alguns pacotes do Python que serão usados nesse trabalho:

- numpy (www.numpy.org) pacote de cálculo científico com Python
- sklearn (http://scikit-learn.org/stable/) fornece ferramentes eficientes e simples para análise de dados, usada nesse trabalho para determinar os exemplos de treinamento
- matplotlib (http://matplotlib.org) biblioteca para gerar gráficos em Python
- planar_utils fornece funções úteis para esse trabalho

In [1]:

```
# Importação dos pacotes
import numpy as np
import matplotlib.pyplot as plt
import sklearn
import sklearn.datasets
import sklearn.linear_model
from planar_utils import plot_decision_boundary, sigmoid, load_planar_dataset, load_
extra_datasets

%matplotlib inline

np.random.seed(1) # define uma semente para geração de números aleatórios
```

2 - Conjunto de dados

A linha de programação abaixo carrega um conjunto de dados de pontos no plano com um formato de flor nas variáveis `X` e `Y`.

In [2]:

```
X, Y = load_planar_dataset()
```

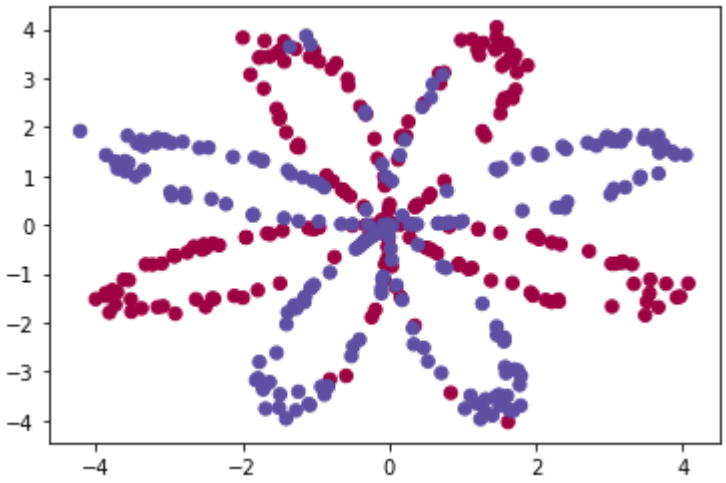
Visualização dos dados usando matplotlib. Os dados parecem ser uma "flor", com alguns pontos classificados como vermelhos (y = 0) e outros como azuis (y = 1). O objetivo é desenvolver uma RNA que classifica os pontos corretamente entre vermelhos e azuis.

In [3]:

```
# Visualização dos dados:
plt.scatter(X[0, :], X[1, :], c=Y.ravel(), s=40, cmap=plt.cm.Spectral)
```

Out[3]:

```
<matplotlib.collections.PathCollection at 0x177966ab488>
```



Exercício #1:

Os dados consistem em:

```
- um tensor numpy (matriz) X que contém as coordenadas dos pontos  (x1, x2) p
ara todos os exemplos
- um tensor numpy (vetor) Y quie contém as classes (vermelho:0, azul:1) para
 todos os exemplos
```

Vamos primeiramente analisar os dados.

Na célula abaixo crie um código que verifica quantos exemplos de treinamento existem e as dimensões (`shape`) das variavéis `X` and `Y`.

Dica: Como obter as dimensões de um tensor numpy? (help) (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.shape.html)

In [4]:

```
# PARA VOCÊ FAZER:

### COMEÇE AQUI ### (≈ 3 linhas)
shape_X = X.shape
shape_Y = Y.shape
m = shape_X[1]  # número de exemplos de treinamento
### TERMINE AQUI ###

print ('A dimensão de X é: ' + str(shape_X))
print ('A dimensão de Y é: ' + str(shape_Y))
print ('Existem m = %d exemplos de treinamento' % (m))
```

```
A dimensão de X é: (2, 400)
A dimensão de Y é: (1, 400)
Existem m = 400 exemplos de treinamento
```

Saída esperada:

A dimensão de X é: (2, 400)
A dimensão de Y é: (1, 400)
Existem m = 400 exemplos de treinamento

3 - Codificação da RNA

3.1 - Definição da estrutura da RNA

Exercício #2:

Na célula abaixo modifique a função `layer_sizes` para definir três variáveis:

- `n_x: número de entradas de cada exemplo`
- `n_h: número de neurônios da camada escondida`
- `n_y: número de saídas da RNA`

Dica: use as dimensões de X e Y para achar n_x e n_y. Além disso defina o número de neurônios da camada intermediária como sendo igual a 4.

In [5]:

```
# PARA VOCÊ FAZER: dimensões da RNA

def layer_sizes(X, Y):
    """
    Argumentos:
    X = conjunto de dados de entrada (dimensão: número de entradas, numero de exemplos))
    Y = classes dos dados (dimensão: número de saídas, numero de exemplos)

    Retorna:
    n_x = número de entradas
    n_h = número de neurônios da camada escondida
    n_y = número de saídas
    """
    ### COMECE AQUI ### (≈ 3 linHAS)
    n_x = X.shape[0]
    n_h = 4
    n_y = Y.shape[0]
    ### TERMINE AQUI ###

    return (n_x, n_h, n_y)
```

In [6]:

```
n_x, n_h, n_y = layer_sizes(X, Y)
print("Número de entradas: n_x = ", n_x)
print("Número de neurônios da camada escondida: n_h = ", n_h)
print("Número de saídas: n_y = ", n_y)
```

```
Número de entradas: n_x =  2
Número de neurônios da camada escondida: n_h =  4
Número de saídas: n_y =  1
```

Saída esperada:

Número de entradas: n_x = 2
Número de neurônios da camada escondida: n_h = 4
Número de saídas: n_y = 1

3.2 - Initialização dos parâmetros

Exercício #3:

Implemente a função `initialize_parameters()` na célula abaixo.

Instruções:

- Garanta que as dimensões dos seus parâmetros esteja correta. Veja as notas de aula.
- Os pesos das ligações são inicializados com números aleatórios.
- Os vieses dos neurônios são inicilizados com zeros.
- Use a função `np.random.randn` para gerar números aleatórios, que no caso desse trabalho terão distribuição normal.

In [7]:

```
# PARA VOCÊ FAZER: inicialização dos parâmetros da RNA

def initialize_parameters(n_x, n_h, n_y):
    """
    Argumentos:
    n_x = número de entradas
    n_h = número de neurônios da camada escondida
    n_y = número de saídas

    Retorna:
    W1 = matriz de pesos de dimensão (n_h, n_x)
    b1 = vetor de vieses de dimensão (n_h, 1)
    W2 = matriz de pesos de dimensão (n_y, n_h)
    b2 = vetor de vieses de dimensão (n_y, 1)
    """

    np.random.seed(2) # define semente para geração de números aleatórios de forma a uniformizar os resultados.

    ### COMECE AQUI ### (≈ 4 linhas)
    W1 = np.random.randn( n_h , n_x ) * 0.01
    b1 = np.zeros( (n_h , 1) ) * 0.01
    W2 = np.random.randn( n_y , n_h ) * 0.01
    b2 = np.zeros( (n_y , 1) ) * 0.01
    ### TERMINE AQUIE ###

    assert (W1.shape == (n_h, n_x))
    assert (b1.shape == (n_h, 1))
    assert (W2.shape == (n_y, n_h))
    assert (b2.shape == (n_y, 1))

    return (W1, b1, W2, b2)
```

In [8]:

```
W1, b1, W2, b2 = initialize_parameters(n_x, n_h, n_y)
print("W1 = ", W1)
print("b1 = ", b1)
print("W2 = ", W2)
print("b2 = ", b2)
```

```
W1 =  [[-0.00416758 -0.00056267]
 [-0.02136196  0.01640271]
 [-0.01793436 -0.00841747]
 [ 0.00502881 -0.01245288]]
b1 =  [[0.]
 [0.]
 [0.]
 [0.]]
W2 =  [[-0.01057952 -0.00909008  0.00551454  0.02292208]]
b2 =  [[0.]]
```

Saída esperada:

W1 = [[-0.00416758 -0.00056267] [-0.02136196 0.01640271] [-0.01793436 -0.00841747] [0.00502881 -0.01245288]]
b1 = [[0.] [0.] [0.] [0.]]
W2 = [[-0.01057952 -0.00909008 0.00551454 0.02292208]]
b2 = [[0.]]

3.3 - Propagação para frente

Exercício #4:

Implemente a função `forward_propagation()` que processa um único exemplo de treinamento.

Instruções:

- Como função de ativação da camada de saída use a função `sigmoid()`, ela está pronta no arquivo planar_utils.py que você já importou.
- Como função de ativação da camada intermediária use a função `np.tanh()`, que faz parte da bilioteca numpy.
- Codifique a propagação para frente, ou seja, calcule $z^{[1]}$, $a^{[1]}$, $z^{[2]}$ and $a^{[2]}$ para cada exemplo do conjunto de dados de treinamento.

Para auxiliar, as equações que implementam a propagação para frente vistas em aula estão repetidas abaixo. As equações não vetorizadas nos exemplos devem ser utilizadas no seu programa.

$$\begin{aligned}
\boldsymbol{z}^{[1](i)} &= \boldsymbol{W}^{[1]}\boldsymbol{x}^{(i)} + \boldsymbol{b}^{[1]} \\
\boldsymbol{a}^{[1](i)} &= g^{[1]}(\boldsymbol{z}^{[1](i)}) \\
z^{[2](i)} &= \boldsymbol{W}^{[2]}\boldsymbol{a}^{[1](i)} + b^{[2]} \\
a^{[2](i)} &= g^{[2]}(z^{[2](i)})
\end{aligned}$$

In [9]:

```
# PARA VOCÊ FAZER: propagação para frente para cada exemplo de treinamento

def forward_propagation(x, W1, b1, W2, b2):
    """
    Argumentos:
    x = dados de entrada de um exemplo com dimensão (n_x, 1)
    W1 = matriz de pesos de dimensão (n_h, n_x)
    b1 = vetor de vieses de dimensão (n_h, 1)
    W2 = matriz de pesos de dimensão (n_y, n_h)
    b2 = vetor de vieses de dimensão (n_y, 1)

    Retorna:
    z1 = estados dos neurônios da camada intermediária de dimensão (n_h, 1)
    a1 = ativações dos neurônios da camada intermediária de dimensão (n_h, 1)
    z2 = estado do neurônio da camada de saída de dimensão (n_y ,1)
    a2 = ativação do neurônio da camada de saída (saída da rede) de dimensão (n_y ,
1)
    """
    # Garante que dimensões dos dados de entrada são de fato um vetor de nx linhas e
uma coluna
    n_x = x.shape[0]
    x = np.reshape(x, (n_x, 1))
    #print(x.shape)

    # Implemente a propagação para frente para calcula A2, cuidado com as dimensões
 nas multiplicações
    ### COMECE AQUI ### (≈ 4 linhas)
    z1 = np.dot(W1 , x) + b1
    a1 = np.tanh( z1 )
    z2 = np.dot(W2 , a1) + b2
    a2 = sigmoid( z2 )
    ### TERMINE AQUI ###

    # Verifica dimensão de a2
    assert(a2.shape == (1, x.shape[1]))

    return (z1, a1, z2, a2)
```

In [10]:

```
z1, a1, z2, a2 = forward_propagation(X[:,0], W1, b1, W2, b2)

# Nota: usaremos a média somente para verificar os resultados.
print('z1 =', z1)
print('a1 =', a1)
print('z2 =', z2)
print('a2 =', a2)
```

```
z1 = [[-0.00703177]
 [ 0.03292871]
 [-0.05170274]
 [-0.03847601]]
a1 = [[-0.00703166]
 [ 0.03291681]
 [-0.05165672]
 [-0.03845703]]
z2 = [[-0.0013912]]
a2 = [[0.4996522]]
```

Saída esperada:

z1 = [[-0.00703177] [0.03292871] [-0.05170274] [-0.03847601]]
a1 = [[-0.00703166] [0.03291681] [-0.05165672] [-0.03845703]]
z2 = [[-0.0013912]]
a2 = [[0.4996522]]

3.4 - Função de erro

Dado que a saída da RNA, $a^{[2]}$, já foi calculada e está na variável `a2`, que contém a saída $a^{[2](i)}$ de um exemplo de treinamento, a função de erro logística, conforme visto na aula, é calculada da seguinte forma:

$$L = -\left(y^{(i)} \log\left(a^{[2](i)}\right) + (1-y^{(i)}) \log\left(1-a^{[2](i)}\right)\right)$$

Exercício #5:

Implemente a função `logistica()` para calcular L. Use a função numpy `np.log` da biblioteca numpy para calcular o logarítmo neperiano de um número real.

In [11]:

```
# PARA VOCÊ FAZER: cálculo da função de erro logística

def logistica(a2, y):
    """
    Calcula o custo entropia-cruzada

    Argumentos:
    a2 = saída da RNA (escalar)
    y = classe real do exemplo (escalar)

    Retorna:
    erro = função logística
    """

    # Calcule o custo entropia-cruzada
    ### COMECE AQUIE ### (≈ 1 linha)
    erro = - ( y * np.log(a2) + ( 1 - y ) * np.log(1-a2) )
    ### TERMINE AQUI ###

    erro = np.squeeze(erro) # para ter certeza de que as dimensões estão corretas

    return erro
```

In [12]:

```
print("Erro = " + str(logistica(a2, Y[0][0])))
```

```
Erro = 0.69245182100033
```

Saída esperada:

Erro = 0.69245182100033

3.5 - Retro-propagação

Usando os resultados da propagação para frente para um exemplo de treinamento, pode-se implementar a retro-propagação para esse exemplo.

Exercício #6:

Implemente a função `backward_propagation()` .

Instruções: A retro-propagação é a parte mais difícil de se calcular nas RNAs. Para auxiliar, as equações que implementam a retro-propagação vistas em aula estão repetidas abaixo. As equações não vetorizadas nos exemplos devem ser utilizadas no seu programa.

$$dz^{[2](i)} = a^{[2](i)} - y^{(i)}$$
$$d\boldsymbol{W}^{[2](i)} = dz^{[2](i)}\boldsymbol{a}^{[1](i)T}$$
$$db^{[2](i)} += dz^{[2](i)}$$
$$d\boldsymbol{z}^{[1](i)} = \boldsymbol{W}^{[2]T}dz^{[2](i)} * \frac{dg^{[1]}(\boldsymbol{z}^{[1](i)})}{dz}$$
$$d\boldsymbol{W}^{[1](i)} = d\boldsymbol{z}^{[1](i)}\boldsymbol{x}^{(i)T}$$
$$d\boldsymbol{b}^{[1](i)} = d\boldsymbol{z}^{[1](i)}$$

- Note que o símbolo "$*$" denota multiplicação elemento por elemento.
- Dicas:
  - Para calcular $d\boldsymbol{z}^{[1](i)}$ é necessário calcular $\frac{dg^{[1]}(\boldsymbol{z}^{[1](i)})}{dz}$.
  - Como $g^{[1]}(.)$ é a função de ativação `tanh` e $a^{[1](i)} = g^{[1]}(z^{[1](i)})$, então $\frac{dg^{[1]}(z^{[1](i)})}{dz} = 1 - (a^{[1](i)})^2$.
  - Portanto, pode-se calcular $\frac{dg^{[1]}(z^{[1](i)})}{dz}$ usando `(1 - np.power(a1, 2))` .
- Note que no caso dessa função de retro-propagação você não precisa acumular as somas dos gradientes, pois esse cálculo é feito para um único exemplo de treinamento. A somatória será realizada posteriormente.

In [13]:

```
# PARA VOCÊ FAZER: retro-propagação

def backward_propagation(x, y, z1, a1, z2, a2, W2):
    """
    Implemente a retro-propagação usando as equações acima.

    Argumentos:
    x = entrada de um exemplo com dimensão (2, 1)
    y = saída da classe real de um exemplo (escalar)
    z1 = estados dos neurônios da camada intermediária de dimensão (n_h, 1)
    a1 = ativações dos neurônios da camada intermediária e dimensão (n_h, 1)
    z2 = estado do neurônio da camada de saída de dimensão (n_y ,1)
    a2 = ativação do neurônio da camada de saída de dimensão (n_y ,1)
    W2 = matriz de pesos da camada de saída de dimensão (n_y, n_h)

    Retorna:
    dW1 = matriz de gradientes dos pesos de dimensão para um exemplo de treinamento
 (n_h, n_x)
    db1 = vetor de gradientes dos vieses de dimensão para um exemplo de treinamento
 (n_h, 1)
    dW2 = matriz de gradientes dos pesos de dimensão para um exemplo de treinamento
 (n_y, n_h)
    db2 = vetor de gradientes dos vieses de dimensão para um exemplo de treinamento
 (n_y, 1)
    """

    # Garante que dimensões de x estão corretas
    n_x = x.shape[0]
    x = np.reshape(x, (n_x, 1))


    # Retro-propagação: calcule dW1, db1, dW2, db2.
    ### COMECE AQUI ### (≈ 6 linhas correspondendo às 6 equações acima)
    dz2 = a2 - y
    dW2 = np.dot(dz2 , a1.T)
    db2 = dz2
    dz1 = np.dot( W2.T , dz2 ) * ( 1 - np.power(a1, 2) )
    dW1 = np.dot(dz1 , x.T)
    db1 = dz1
    ### TERMINE AQUI ###

    return dW1, db1, dW2, db2
```

In [14]:

```
dW1, db1, dW2, db2 = backward_propagation(X[:,0], Y[0][0], z1, a1, z2, a2, W2)
print ("dW1 = ", dW1)
print ("db1 = ", db1)
print ("dW2 = ", dW2)
print ("db2 = ", db2)
```

```
dW1 =  [[-0.00636647 -0.0189027 ]
 [-0.0054645  -0.01622467]
 [ 0.00330981  0.00982716]
 [ 0.01377416  0.0408969 ]]
db1 =  [[-0.00528582]
 [-0.00453696]
 [ 0.002748  ]
 [ 0.01143613]]
dW2 =  [[-0.00351338  0.01644696 -0.02581039 -0.01921514]]
db2 =  [[0.4996522]]
```

Saída esperada:

dW1 = [[-0.00636647 -0.0189027] [-0.0054645 -0.01622467] [0.00330981 0.00982716] [0.01377416 0.0408969]]
db1 = [[-0.00528582] [-0.00453696] [0.002748] [0.01143613]]
dW2 = [[-0.00351338 0.01644696 -0.02581039 -0.01921514]]
db2 = [[0.4996522]]

3.6 - Atualização dos parâmetros

Exercício #7:

Implemente a atualização dos parâmetros. Deve-se usar dJdW1, dJdb1, dJdW2 e dJdb2 para atualizar W1, b1, W2 e b2.

Equação geral do gradienet descendente: $\theta = \theta - \alpha \frac{\partial J}{\partial \theta}$ onde α é a taxa de aprendizagem e θ representa um parâmetro genérico.

In [15]:

```
# PARA VOCÊ FAZER: atualização dos parâmetros

def update_parameters(W1, b1, W2, b2, dJdW1, dJdb1, dJdW2, dJdb2, learning_rate = 1.
2):
    """
    Atualização dos parâmetros usando a regra do gradiente descendente

    Argumentos:
    W1 = matriz de pesos de dimensão (n_h, n_x)
    b1 = vetor de vieses de dimensão (n_h, 1)
    W2 = matriz de pesos de dimensão (n_y, n_h)
    b2 = vetor de vieses de dimensão (n_y, 1)
    dJdW1 = matriz de gradientes dos pesos de dimensão para todos exemplos de treina
mento (n_h, n_x)
    dJdb1 = vetor de gradientes dos vieses de dimensão para todos exemplos de treina
mento (n_h, 1)
    dJdW2 = matriz de gradientes dos pesos de dimensão para todos exemplos de treina
mento (n_y, n_h)
    dJdb2 = vetor de gradientes dos vieses de dimensão para todos exemplos de treina
mento (n_y, 1)

    Retorna parametros atualizados:
    W1 = matriz de pesos de dimensão (n_h, n_x)
    b1 = vetor de vieses de dimensão (n_h, 1)
    W2 = matriz de pesos de dimensão (n_y, n_h)
    b2 = vetor de vieses de dimensão (n_y, 1)
    """

    # Atualização dos parâmetros
    ### COMECE AQUI ### (≈ 4 linhas)
    W1 -= learning_rate * dJdW1
    b1 -= learning_rate * dJdb1
    W2 -= learning_rate * dJdW2
    b2 -= learning_rate * dJdb2
    ### TERMINE AQUI ###

    return W1, b1, W2, b2
```

In [16]:

```
# Nesse momento utilizamos os gradientes dos parâmetros para um único exemplo somente
# para podermos testar a função update parâmetros
dJdW1 = dW1
dJdb1 = db1
dJdW2 = dW2
dJdb2 = db2

W1_n, b1_n, W2_n, b2_n = update_parameters(W1, b1, W2, b2, dJdW1, dJdb1, dJdW2, dJdb2)

print("W1 = ", W1_n)
print("b1 = ", b1_n)
print("W2 = ", W2_n)
print("b2 = ", b2_n)
```

```
W1 =  [[ 0.00347218  0.02212057]
 [-0.01480456  0.03587231]
 [-0.02190612 -0.02021007]
 [-0.01150018 -0.06152917]]
b1 =  [[ 0.00634298]
 [ 0.00544435]
 [-0.0032976 ]
 [-0.01372336]]
W2 =  [[-0.00636346 -0.02882642  0.03648701  0.04598025]]
b2 =  [[-0.59958264]]
```

Saída esperada:

W1 = [[0.00347218 0.02212057] [-0.01480456 0.03587231] [-0.02190612 -0.02021007] [-0.01150018 -0.06152917]]
b1 = [[0.00634298] [0.00544435] [-0.0032976] [-0.01372336]]
W2 = [[-0.00636346 -0.02882642 0.03648701 0.04598025]]
b2 = [[-0.59958264]]

3.7 - Integração das tarefas 3.1 a 3.6 na função rna()

Exercício #8:

Programe a sua rede neural na função `rna()`. Inclua tanto a propagação para frente como a retro-propagação. A sua rede deve seguir o algoritmo do Quadro 3 da Aula 5 - Classificação binária com RNA rasa, que em linhas gerais é o seguinte:

```
Inicializa parâmetros
  for e=1 to n_epocas
     zera gradientes
     Iniciliza função de custo com zero
     for i=1 to m
         Calcula a propagação para frente para cada exemplo
         Calcula função de erro
         Atualiza função de custo
         Calcula a propagação para trás para cada exemplo
         Acumula os gradientes de cada exemplo nos gradientes de cada parâmetro da
rede
     Atualiza os parâmetros
```

Instruções:

- A sua função `rna()` deve usar as funções programadas anteriormente.
- Um comando `for` em python para um contador `i` variando de `0` até `n-1` é implementado por: `for i in range(n):
- Em python para acumular valores em uma variável `dx` pode-se usar `dx += dx`.

In [17]:

```
# PARA VOCÊ FAZER: programação da rede neural

def rna(X, Y, n_h, num_epocas = 10000, print_cost=False):
    """
    Argumentos:
    X = matriz de dados de entrada de dimensão (2, número de exemplos)
    Y = vetor com as classes dos exemplos de dimensão (1, número de exemplos)
    n_h = número de neurônios da camada escondida
    num_epocas = número de épocas
    print_cost = se for True, imprime o valor do custo a cada 1000 épocas

    Retorna parâmetros calculados no treinamento:
    W1 = matriz de pesos de dimensão (n_h, n_x)
    b1 = vetor de vieses de dimensão (n_h, 1)
    W2 = matriz de pesos de dimensão (n_y, n_h)
    b2 = vetor de vieses de dimensão (n_y, 1)
    """

    #np.random.seed(3)
    n_x = layer_sizes(X, Y)[0]
    n_y = layer_sizes(X, Y)[2]
    m = X.shape[1]

    # Inicializa parâmetros. Entradas: "n_x, n_h, n_y". Saídas: "parameters".
    ### COMECE AQUI ### (≈ 1 linha)
    W1, b1, W2, b2 = initialize_parameters(n_x,n_h,n_y)
    ### TERMINE AQUI ###

    # Iteração nas épocas
    for e in range(num_epocas):

        # No início de cada época deve-se inicializar os gradientes dos parâmetros p
ara todos os exemplos com zeros
        ### COMECE AQUI ### (≈ 4 linhas)
        dJdW1 = 0
        dJdb1 = 0
        dJdW2 = 0
        dJdb2 = 0
        ### TERMINE AQUI ###

        # Incializa função de custo
        custo = 0

        # Iteração nos exemplos
        for i in range(m):
            # Propagação para frente. Entradas: X[:,i] e parameters. Saída: za.
            ### COMECE AQUI ### (≈ 1 linha)
            z1, a1, z2, a2 = forward_propagation(X[:,i],W1,b1,W2,b2)
            ## TERMINE AQUI ###

            # Função de erro. Entradas: a2, Y[0][i]. Saída: erro.
            ### COMECE AQUI ### (≈ 1 linhas)
            # Calcula erro usando a função logistica
            erro = logistica(a2, Y[0][i])
            ## TERMINE AQUI ###

            # Atualiza função de custo somando o erro do exemplo "i" e dividindo pel
o número total de exemplos "m".
            ### COMECE AQUI ### (≈ 1 linha)
```

```
            custo += erro / m
            ## TERMINE AQUI ###

            # Retro-propagação. Entradas: parameters, X[:,i], Y[0][i], za, parameter
s. Saídas: grads.
            ### COMECE AQUI ### (≈ 1 linha)
            dW1, db1, dW2, db2 = backward_propagation(X[:,i],Y[0][i],z1,a1,z2,a2,W2)
            ## TERMINE AQUI ###

            # Acumula os gradientes de cada exemplo em dJdpar dividindo pelo numero
 de exemplos. Use o dicionário grads.
            ### COMECE AQUI ### (≈ 4 linhas)
            dJdW1 += dW1 / m
            dJdb1 += db1 / m
            dJdW2 += dW2 / m
            dJdb2 += db2 / m
            ## TERMINE AQUI ###

         # Atualização dos parâmetros. Entradas: "parameters, dJ". Saídas: "parame
rs".
        ### COMECE AQUI ### (≈ 1 linha)
        W1, b1, W2, b2 = update_parameters(W1,b1,W2,b2,dJdW1,dJdb1,dJdW2,dJdb2)
        ### TERMINE AQUI ###

        # iMPRESSÃO DO CUSTO A CADA 1000 épocas
        if print_cost and e % 1000 == 0:
            print ("Custo após época %i: %f" %(e, custo))

    return W1, b1, W2, b2
```

In [18]:

```
W1, b1, W2, b2 = rna(X, Y, 4, num_epocas=1, print_cost=True)
print("W1 = ", W1)
print("b1 = ", b1)
print("W2 = ", W2)
print("b2 = ", b2)
```

```
Custo após época 0: 0.693048
W1 =  [[-0.00445085  0.0019323 ]
 [-0.02161288  0.01854112]
 [-0.01778975 -0.00971864]
 [ 0.00564676 -0.01784282]]
b1 =  [[-1.28217636e-07]
 [ 1.16508870e-06]
 [ 8.64015190e-08]
 [-3.63658142e-07]]
W2 =  [[-0.01055846 -0.01353296  0.00702278  0.02599079]]
b2 =  [[1.30707768e-05]]
```

Saída esperada:

Custo após época 0: 0.693048
W1 = [[-0.00445085 0.0019323] [-0.02161288 0.01854112] [-0.01778975 -0.00971864] [0.00564676 -0.01784282]]
b1 = [[-1.28217636e-07] [1.16508870e-06] [8.64015190e-08] [-3.63658142e-07]]
W2 = [[-0.01055846 -0.01353296 0.00702278 0.02599079]]
b2 = [[1.30707768e-05]]

4 - Treinamento e teste da RNA

4.1 - Previsão das saídas

Exercício #9:

Use a sua rede neural para realizar previsões programando na célula abaixo o método `predict()` .

Use a propagação para frente para calcular as previsões. Como a saída da rede neural será um valor entre 0 e 1, para definir as classes faz-se:

$$previsão = y_{prediction} = \begin{cases} 1 & \text{se } saída > 0,5 \\ 0 & \text{caso contrário} \end{cases}$$

Genericamente, na equação acima pode-se usar no lugar de 0,5 um valor de limiar genérico, assim, tem-se:

$$y_{prediction} = \begin{cases} 1 & \text{se } saída > limiar \\ 0 & \text{caso contrário} \end{cases}$$

In [19]:

```
# PARA VOCÊ FAZER: função predict

def predict(W1, b1, W2, b2, X):
    """
    Usando os parâmetros calculados no treinamento, prevê a classe para todos os exe
mplos na matriz X

    Argumentos:
    W1 = matriz de pesos de dimensão (n_h, n_x)
    b1 = vetor de vieses de dimensão (n_h, 1)
    W2 = matriz de pesos de dimensão (n_y, n_h)
    b2 = vetor de vieses de dimensão (n_y, 1)
    X = matriz de entradas de dimensão (n_x, m)

    Retorna
    predictions = vetor de previsões (vermelho: 0 / azul: 1)
    """
    # Incicliza vetor de previsões para os m exemplos
    m = X.shape[1]
    predictions = np.zeros((m, 1))

    # Calcula as probabilidades usando a propagação para frente e classifica como 0/
1 usando um limiar de 0,5.
    # utilize um comando de repetição for para percorrer todos os exemplos
    ### COMECE AQUI ### (≈ 2 linhas)
    for i in range(m):
        # calcula a propagação para frente iusando a função foward_propagation
        z1, a1, z2, a2 = forward_propagation(X[:,i],W1,b1,W2,b2)

        # Calcule a classe prevista pela rede usando o limiar de 0,5
        predictions[i] = a2 > 0.5
    ### TERMINE AQUI ###

    return predictions
```

In [20]:

```
#parameters, X_assess = predict_test_case()

predictions = predict(W1, b1, W2, b2, X)
print("Média das previsões = " + str(np.mean(predictions)))
```

```
Média das previsões = 0.48
```

Saída esperada:

Média das previsões = 0.48

4.2 - Treinamento da RNA

Agora verifique o desempenho do seu modelo no conjunto de dados, após o treinamento da rede neural com 7.000 épocas. Execute o programa abaixo para testar o seu modelo de uma única camada intermediária com n_h neurônios.

In [21]:

```
# Treinamento e excecução da rede neural de uma única camada
W1, b1, W2, b2 = rna(X, Y, n_h = 4, num_epocas = 7001, print_cost=True)
```

```
Custo após época 0: 0.693048
Custo após época 1000: 0.288083
Custo após época 2000: 0.254385
Custo após época 3000: 0.233864
Custo após época 4000: 0.226792
Custo após época 5000: 0.222644
Custo após época 6000: 0.219731
Custo após época 7000: 0.217504
```

Saída esperada:

Custo após época 0: 0.693048
Custo após época 1000: 0.288083
Custo após época 2000: 0.254385
Custo após época 3000: 0.233864
Custo após época 4000: 0.226792
Custo após época 5000: 0.222644
Custo após época 6000: 0.219731
Custo após época 7000: 0.217504

4.3 - Resultados

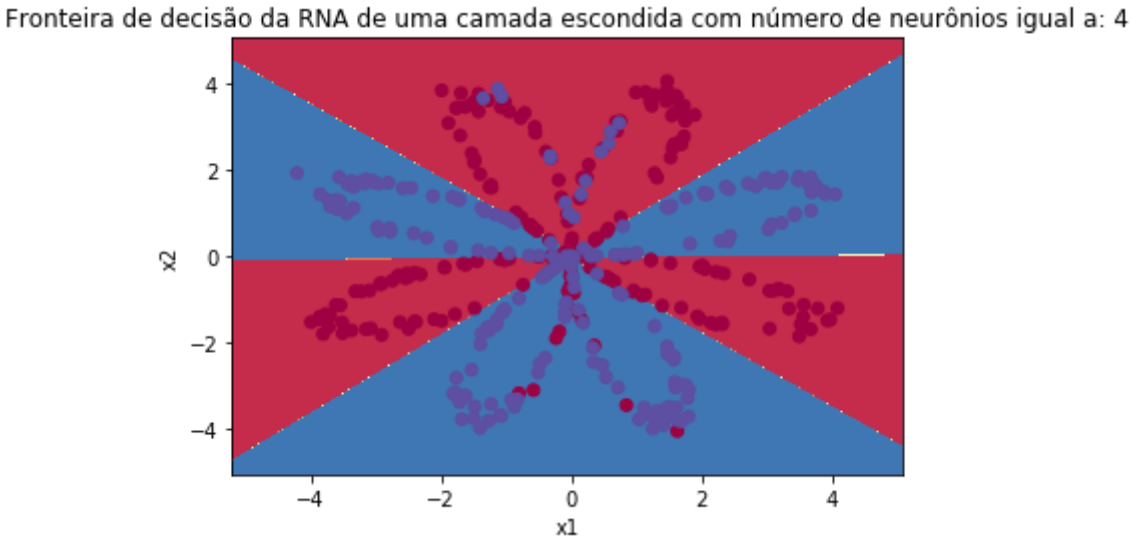
Execute a célula abaixo para fazer o gráfico dos dados com a fronteira de decisão

In [22]:

```
plot_decision_boundary(predict, X=X, Y=Y.ravel(), W1=W1, b1=b1, W2=W2, b2=b2)
plt.title("Fronteira de decisão da RNA de uma camada escondida com número de neurôni
os igual a: " + str(4))
```

Out[22]:

```
Text(0.5, 1.0, 'Fronteira de decisão da RNA de uma camada escondida com
número de neurônios igual a: 4')
```



Exercício #10:

Implemente na célula abaixo o cálculo da exatidão obtida para todos os exemplos de treinamento. A equação que implementa o cálculo da exatidão éa seguinte:

$$exatidão = 100 * (1 - \frac{1}{m}\sum_{i=1}^{m} |y_{real}^{(i)} - y_{previsto}^{(i)}|)$$

Use as funções `np.abs` e `np.sum` para calcular o módulo de um número e a somatória dos elementos de um vetor.

Cuidado com as dimensões das previsões e do vetor de saídas `Y` .

In [23]:

```
# PARA VOCÊ FAZER: Calculo da exatidão

# Calcule as previsões da rede usando a função predict
### COMECE AQUI ### (≈ 1 linha)
predictions = predict(W1, b1, W2, b2, X)
### TERMINE AQUI ###

# Calcule a exatidão obtida pela rede. Para acertar as dimensões use o transposto de
predictions
### COMECE AQUI ### (≈ 1 linha)
exatidao = 100 * ( 1 - np.sum( np.abs( Y - predictions.T ) ) / m )
### TERMINE AQUI ###

print('Exatidão: ' + str(exatidao) + ' %')
```

```
Exatidão: 90.75 %
```

Saída esperada:

Exatidão: 90.75 %

Por esse resultado podemos conlcuir que a RNA foi capaz de aprender o padrão das pétalas da flor! Redes neurais são capazes de aprender fronteiras de decisão muito complexas e não lineares, mesmo com poucos neurônios.

Você sabe como calcular o número total de parâmetros dessa RNA?

4.4 - Ajuste do número de neurônios da camada escondida

Agora, tente outros números de neurônios na camada escondida. Para isso, execute o seguinte programa. Pode levar alguns minutos para executar. Você deve observar comportamentos diferentes para cada número de neurônios na camada escondida.

A execução dessa célula vai demorar alguns minutos.

In [ ]:

```
plt.figure(figsize=(16, 32))
hidden_layer_sizes = [2, 3, 4, 5, 20, 50]
for i, n_h in enumerate(hidden_layer_sizes):
    plt.subplot(5, 2, i+1)
    plt.title('Numero de neurônios = %d' % n_h)
    W1, b1, W2, b2 = rna(X, Y, n_h, num_epocas = 5000)
    plot_decision_boundary(predict, X=X, Y=Y.T.ravel(), W1=W1, b1=b1, W2=W2, b2=b2)
    predictions = predict(W1, b1, W2, b2, X)
    exatidao = 100*(1-np.sum(np.abs(Y-predictions.T))/m)
    print ("Exatidão para {} neurônios: {} %".format(n_h, exatidao))
```

```
Exatidão para 2 neurônios: 67.25 %
Exatidão para 3 neurônios: 90.75 %
Exatidão para 4 neurônios: 90.5 %
Exatidão para 5 neurônios: 91.25 %
```

Interpretação:

- Quanto maior a RNA (maior o número de neurônios) melhor o seu desempenho para aprender os dados de treinamento, até que eventualmente um modelo muito grande apresenta sobre-ajuste dos dados.
- O melhor número de camadas escondidas parece ser algo em torno de n_h = 5.
- Veremos com mais detalhes como desenvolver usar RNAs grandes sem problemas de sobre-ajuste dos dados.

4.5 - Desempenho com outros padrões de dados

Exercício #11:

Treine novamente a sua RNA para cada um dos seguintes conjunto de dados. Após o treinamento execute a RNA para calcular as suas previsões e a sua exatidão. Para isso, modifique o programa abaixo para cada conjunto de dados de cada vez.

Dica: use como base parte do programa do item 4.4.

In [ ]:

```
# PARA VOCÊ FAZER: treinar e executar modelo com outros conjuntos de dados

# Conjunto de dados
noisy_circles, noisy_moons, blobs, gaussian_quantiles, no_structure = load_extra_dat
asets()

datasets = {"noisy_circles": noisy_circles,
            "noisy_moons": noisy_moons,
            "blobs": blobs,
            "gaussian_quantiles": gaussian_quantiles}

for key, value in datasets.items():
    ### COMECE AQUI" (≈ 1 linha)
    dataset = key
    ### tERMINE AQUI ###

    X, Y = datasets[dataset]
    X, Y = X.T, Y.reshape(1, Y.shape[0])

    # make blobs binary
    if dataset == "blobs":
        Y = Y%2

    # Visualização dos dados
    plt.scatter(X[0, :], X[1, :], c=Y.ravel(), s=40, cmap=plt.cm.Spectral);

    # Número de neurônios da camada escondida
    n_h = 5

    ### COMECE AQUI ### (≈ 4 linhas)
    W1, b1, W2, b2 = initialize_parameters(n_x,n_h,n_y)
    plot_decision_boundary(predict, X=X, Y=Y.ravel(), W1=W1, b1=b1, W2=W2, b2=b2)
    predictions = predict(W1, b1, W2, b2, X)
    exatidao = 100 * ( 1 - np.sum( np.abs( Y - predictions.T ) ) / m )
    ### TERMINE AQUI ###

    print ("{} -> exatidão para {} neurônios: {} %".format(key, n_h, exatidao))
```

Saídas esperadas:

noyse_circles exatidão 78,0%
noyse_moons: exatidão 99,0%
bloobs: exatidão 83,0%
gaussian_quantiles: exatidão 100,0%

Referência:

- http://scs.ryerson.ca/~aharley/neural-networks/ (http://scs.ryerson.ca/~aharley/neural-networks/)

O que você aprendeu nesse trabalho:

- Construir uma RNA de uma única camada intermediária
- Implementar a propagação para frente e a retro-propagação
- Treinar uma RNA
- Observar o impacto de variar o número de neurônios da camada intermediária