Tema 03. Redes de Neuronas

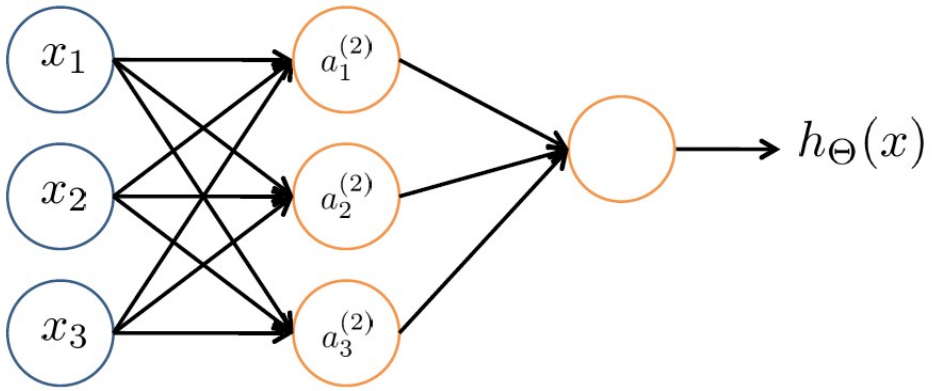
Autor: Ismael Sagredo Olivenza

3.4 How learn an ANN?

Remember forward propagation:

$$a_i^j = g((\sum_{k=1}^{|a^{j-1}|} \Theta_{i,k}^j a_k^{j-1}) + b_i^j) \forall i \in [1..|a^j|], j \in [1..|C|]$$

Where C is the layers in the model, |C| numer of layers, $|a_j|$ number of neurons in the layer j, $|a^{j-1}|$ number of neurons in the previous layer and g the activation function (ReLU, sigmoidal, etc)



$$a_1^2 = g(\Theta_{1,1}^1 x_1 + \Theta_{1,2}^1 x_2 + \Theta_{1,3}^1 x_3 + b_1^1)$$

$$a_2^2 = g(\Theta_{2,1}^1 x_1 + \Theta_{2,2}^1 x_2 + \Theta_{2,3}^1 x_3 + b_2^1)$$

$$a_3^2 = g(\Theta_{3,1}^1 x_1 + \Theta_{3,2}^1 x_2 + \Theta_{3,3}^1 x_3 + b_3^2)$$

$$y' = h_\Theta(x) = a_1^3 = g(\Theta_{1,0}^2 + \Theta_{1,1}^2 a_1^2 + \Theta_{1,2}^2 a_2^2 + \Theta_{1,3}^2 a_3^2)$$

As in the algorithms we have seen before, we need to calculate a cost function J. We start from the logistic regression cost function:

$$J(\vec{w}, b) = -\frac{1}{m} \sum_{i=1}^{m} [y_i log(y'_i) + (1 - y_i) log(1 - y'_i)]$$

But we can have more than one output y'_i = i-th output

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} [y_{k,i} log(y'_{k,i}) + (1 - y_{k,i}) log(1 - y'_{k,i})]$$

The sum of the error of all K output neurons is the error produced by an example, the mean error - the average of the m examples.

3.4.1 How modify the weights of the ANN to learn?

Gradient descent is your friend :), in logistic regression:

$$\frac{\partial J(w,b)}{\partial w_j} = \sum_{i=1}^{m} y_i' - y_i x_{j,i}$$

Neural network:

$$\frac{\partial J(\Theta)}{\partial \Theta_{i,j}^{l}} = \sum_{k=1}^{m} \delta_{i,k}^{l+1} a_{j,k}^{l}$$

The partial derivative of any weight $(\Theta^l_{i,j})$ connecting neuron j in layer l to neuron i in layer l+1 of the weight matrix Θ^l is equal to the error produced in layer l+1 (which we call delta δ), for example k, by the activation of neuron j in layer l for example k.

Now we have to determine what the error is.

In the output layer (l = L or last layer) the error will be the difference between the expected output and the actual output multiplied by the derivative of the output.

$$\delta^L_j = (y'_j - y_j)g'(y'_j) = (a^L_j - y_j)g'(a^L_j)$$

In the penultimate layer, however, we must propagate the error through the network using the weights.

$$\delta_i^{L-1} = \Theta_{i,j}^{L-1} \delta_j^L g'(a_i^{L-1})$$

Where g' is the derivative of the logistic function, which in this case is :

$$g'(a_i^l) = a_i^l(1 - a_i^l)$$

In other words, the derivative of the function is to multiply the activation obtained by 1- the activation previously obtained by the function.

Generalising for any layer, the delta factor of the error can be expressed as follows:

$$\delta_i^l = \Theta_{i,j}^l \delta_j^{l+1} g'(a_i^l)$$

The gradient decrease for any weight in any layer can therefore be expressed as:

$$\Theta_{i,j}^l = \Theta_{i,j}^l - \alpha \frac{1}{m} \frac{\partial J(\Theta)}{\partial \Theta_{i,j}^l}$$

$$\frac{\partial J(\Theta)}{\partial \Theta_{i,j}^l} = \sum_{k=1}^{m} \delta_{i,k}^{l+1} a_{j,k}^l$$

And the successive deltas would be calculated as follows:

$$\delta_{i,k}^{l} = \Theta_{i,j}^{l} \delta_{i,k}^{l+1} g'(a_{i,k}^{l})$$

for all layers from 2 to L-1 and for layer L

$$\delta_{i,k}^{L} = a_{i,k}^{L} - y_{i,k} g'(a_{i,k}^{L})$$

$$g'(a_{i,k}^{L}) = a_{i,k}^{L} * (1 - a_{i,k}^{L})$$

For all k training examples, and for each of the i neurons of the corresponding l layer.

3.4.2 Algorithm:

- Initialise the network weights and thresholds to random values.
- For each input pattern
  - Load the inputs into the input neurons.
  - Propagate the input through the network (forward propagation).
  - Calculate the error of that pattern
  - Calculate all δ for all connections in the network.
  - Calculate the weights starting from the output to the input (Backpropagation)
  - Store the error made
- Calculate the average error

3.4.3 Regularization

If we apply L2 regularisation, the error is slightly different. The sum of all lattice weights squared must be added to it

$$JL2(\Theta) = J(\Theta) + \lambda \frac{1}{2m} \sum_{l=1}^{L-1} \sum_{j=1}^{Dim(l)} \sum_{i=1}^{Dim(l+1)} (\Theta_{i,j}^{l})^2$$

The L1 regularisation is similar, but calculating the absolute value. The gradien term in regulation:

$$\partial JL2(\Theta) = \partial J(\Theta) + \lambda \frac{1}{m} \sum_{l=1}^{L-1} \sum_{j=1}^{Dim(l)} \sum_{i=1}^{Dim(l+1)} (\Theta_{i,j}^{l})$$

3.4.4 Example

Para el siguiente ejemplo:
Neuronas de entrada 2
Neuronas capa oculta 2
Neuronas de salida 2
Salida esperada 0.7,0.1
Pesos iniciales
Tetha1 = 2X2 [[0.1 , 0.2], [0.2 , 0.1]]
Tetha2 = 2X2 [[0.2 , 0.3], [0.1 , 0.1]]
Salida = [0.5 , 0.4]

Calculamos el backpropagation:

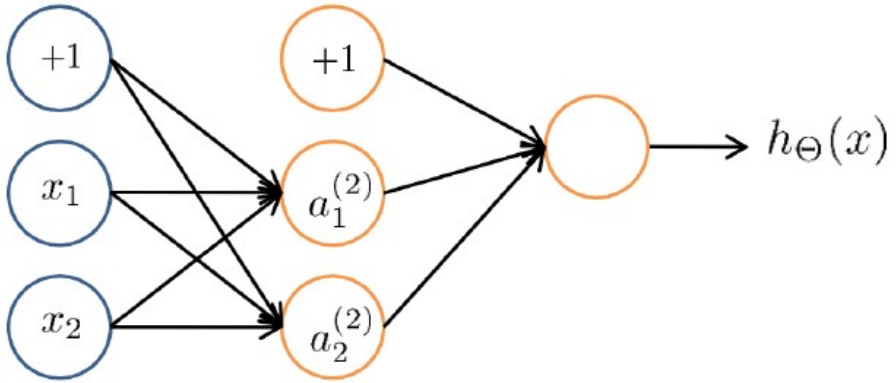
Calculamos:

$$\delta_1^3 = (0.5 - 0.7) * (0.7) * (1 - 0.7) = -0.042$$
$$\delta_2^3 = (0.4 - 0.1) * (0.1) * (1 - 0.1) = 0.027$$

$$\delta_1^2 = \Theta^2[1,1] * \delta_1^3 * g'(a_1^2) + \Theta^2[1,2] * \delta_2^3 * g'(a_1^2)$$
$$\delta_2^2 = \Theta^2[2,1] * \delta_1^3 * g'(a_2^2) + \Theta^2[2,2] * \delta_2^3 * g'(a_2^2)$$

$$\Theta^1 = \Theta^1 - \alpha * \delta^2(a^1)$$
$$\Theta^2 = \Theta^2 - \alpha * \delta^3(a^2)$$

Zero initialization



$$\Theta_{ij}^{(l)} = 0 \text{ for all } i, j, l.$$

$$\Theta_{01}^{(1)} = \Theta_{02}^{(1)} \qquad a_1^{(2)} = a_2^{(2)} \qquad \delta_1^{(2)} = \delta_2^{(2)}$$

$$\frac{\partial}{\partial \Theta_{01}^{(1)}} J(\Theta) = \frac{\partial}{\partial \Theta_{02}^{(1)}} J(\Theta)$$

and again: $\Theta_{01}^{(1)} = \Theta_{02}^{(1)}$

After each update, parameters corresponding to inputs going into each of two hidden units keep being identical. Every hidden unit is computing the same combination of inputs, i.e., the same “hidden feature”

Random initialization: Symmetry breaking

Initialize each $\Theta_{ij}^{(l)}$ to a random value in $[-\epsilon, \epsilon]$
(i.e. $-\epsilon \leq \Theta_{ij}^{(l)} \leq \epsilon$)

E.g.

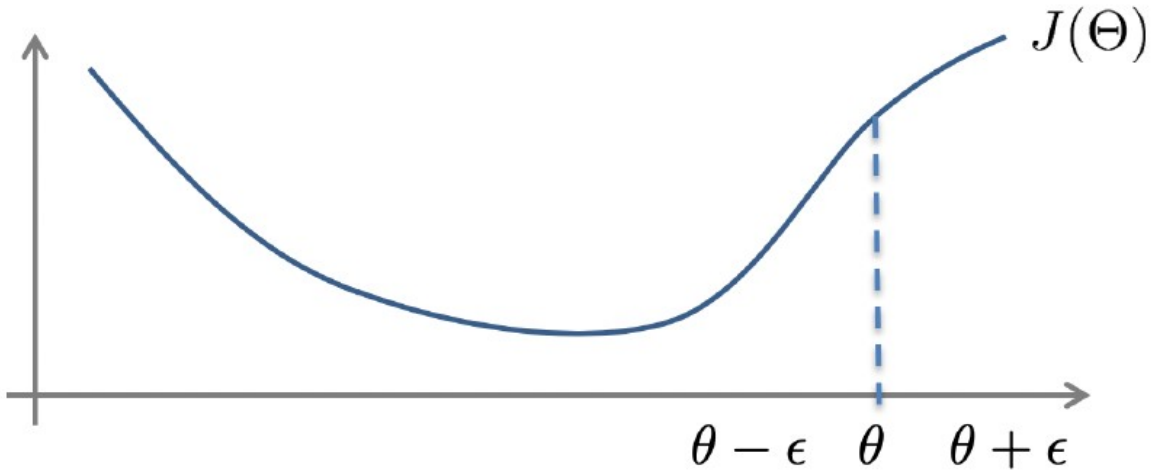
```
Theta1 = np.random.random((10,11))*(2*INIT_EPSILON)
         - INIT_EPSILON

Theta2 = np.random.random((1,11))*(2*INIT_EPSILON)
                  - INIT_EPSILON
```

3.5 Optimization algorithms

- **Gradient Descent**: explained before.
- **BFGS / L-BFGS**: using a limited amount of computer memory. It is a popular algorithm for parameter estimation in machine learning. The algorithm's target problem is to minimize over unconstrained values of the real-vector
- **ADAM**: gradient-based optimization of stochastic objective functions. Used in problems that are large in terms of data and/or parameters o with very noisy and/or sparse gradients (paper)
- **Stochastic gradient descent**: estimate gradient using also a subset of the data. Is used in high-dimensional problems.

Numerical estimation of gradients



$$\frac{d}{d\Theta} J(\Theta) = \frac{J(\Theta + \epsilon) - J(\Theta - \epsilon)}{2\epsilon} \qquad \epsilon = 10^{-4}$$

Implement:

```
gradApprox = ( J(theta + EPSILON) –
               J(theta – EPSILON)    ) / ( 2*EPSILON )
```

We can use it to check that the calculations are being done correctly. The value of the numerical estimate should be similar to the value obtained by gradient descent.

```
for i in range(len(thetaVec)):
  thetaPlus = thetaVec
  thetaPlus[i] = thetaPlus[i] + EPSILON
  thetaMinus = thetaVec
  thetaMinus[i] = thetaMinus[i] - EPSILON
  gradApprox[i] = (J(thetaPlus) - J(thetaMinus))
                  /(2*EPSILON)
```

Example MLP

```
def MLP(X,Y,hidden, fun, funDer, alpha, epochs):
        theta1 = np.random.rand(hidden, X.shape[1])
        b2 = np.random.rand(hidden)
        theta2 = np.random.rand(Y.shape[1], hidden)
        b3 = np.random.rand(Y.shape[1])
        histoy = []
        m=X.shape[0]
        c = 0
        delta2 = np.zeros(Y.shape[1])
        delta3 = np.zeros(hidden)
        for epoch in range(epochs):
                for i in range(m):
                        a1,a2,a3 = ForwardProp(X[i],fun,theta1,theta2,b2,b3)
                        y_ = Y[i].getA1()
                        e = cost(y_,a3)
                        c = np.sum(e)
                        theta1, theta2 = BackPropagation(y_,a1,a2,a3,theta1,theta2,b2,b3,funDer,alpha)
                delta3 = delta3 / m
                delta2 = delta2 / m
                J = - c / m
                histoy.append(J)
                print("cost "+str(J)+", Epoch "+str(epoch))
```

ForwardPropagation

```
def ForwardProp(x,fun,theta1,theta2,b2,b3):
        a1= np.array(x)
        z2 = np.dot(a1, theta1.T)+b2
        a2 = fun(z2)

        z3 = np.dot(a2, theta2.T) + b3
        a3 = fun(z3)
        return a1[0],a2[0],a3[0]
```

BackPropagation

```
def BackPropagation(y,a1,a2,a3,theta1,theta2,b2,b3,funDer,alpha):
        #generamos los deltas
        delta3 = DeltaLast(a3,y,funDer)
        delta2 = Delta(theta2,delta3,a2,funDer)
        for j in range(delta3.shape[0]):
                for k in range(a2.shape[0]):
                        theta2[j,k] = theta2[j,k] - alpha * delta3[j]*a2[k]

        for j in range(delta2.shape[0]):
                for k in range(a1.shape[0]):
                        theta1[j,k] = theta1[j,k] - alpha * delta2[j]*a1[k]

        #Faltaría la actualización de de los umbrales b3, b2.
        return theta1, theta2
```

Delta rules

```
def DeltaLast(h,y,funDer):
        D = np.zeros(h.shape[0])
        for j in range(h.shape[0]):
                D[j] = (h[j]-y[j])*funDer(h[j])
        return D

def Delta(thetaL,deltaNext,aL,funDer):
        D = np.zeros(aL.shape[0])
        for i in range(aL.shape[0]):
                for j in range(deltaNext.shape[0]):
                        D[i] += thetaL[j,i]*deltaNext[j]*funDer(aL[i])
        return D
```

Miscelanea

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def sigmoidPrime(z):
        return (1-z)*z

def cost(y,h):
        J = y * np.log(h) + (1 - y) * np.log(1 - h)
        return J
```