Notas TFG

Juan

11 de marzo de 2017

Backpropagation

Entre las ANN's existentes, una de las más empleadas por su alta eficiencia es el Perceptrón Multicapa (MLP) mediante el uso del algoritmo de *Back-Propagation*. A diferencia del Perceptrón de una única capa, el MLP puede implementar gran variedad de funciones complejas, incluída la función XOR, que no puede ser realizada por el de una única capa como demostraron Minsky y Papertˆ{ojo} (Perceptrons: an introduction to computational geometry is a book written by Marvin Minsky and Seymour Papert and published in 1969).

En un MLP una unidad solo puede conectarse a la capa adyacente siguiente, no permitiéndose conexiones recurrentes ni en la misma capa. Sea K el número de capas, M el número de inputs y N el número de outputs. El input en una unidad (siempre que no sea en la capa de entrada) es la suma de los outputs de unidades conectadas en la capa anterior. Sea x_i^j el input de la unidad i en la capa j, w_{ij}^k el peso de la conexión entre la unidad i en la capa k y la unidad j de la capa $k+1$, y_i^j el output de la unidad i en la capa j y, por último, θ_i^j el umbral (o sesgo) de la unidad i en la capa j. Entonces tenemos que los input de la capa $k+1$ son:

$$x_j^{k+1} = \sum_i w_{ij}^k y_i^k$$

donde

$$y_i^j = f(x_i^j) \quad (1)$$

siendo f la función de activación. Nosotros utilizaremos como f la función sigmoide

$$f(x_i^j) = \frac{1}{1 + e^{-\frac{x_i^j - \theta_i^j}{T}}}.$$

Dependiendo de las aplicaciones un output que tome valores negativos es necesario, y podemos utilizar entonces la función

$$f(x_i^j) = \frac{1 - e^{-\frac{x_i^j - \theta_i^j}{T}}}{1 + e^{-\frac{x_i^j - \theta_i^j}{T}}}.$$

Hay dos fases en el algoritmo de *Back-Propagation*, la primera es computar el cálculo a través de las capas utilizando (1). La segunda fase es actualizar los pesos, operación que se realiza computando el error entre el valor esperado y el valor real calculado en la primera fase. Este proceso clasifica el algoritmo de *Back-Propagation* dentro de la categoría de los algoritmos de aprendizaje supervisado. Básicamente el algoritmo de *Back-Propagation* es un algoritmo de gradiente descendente.

A continuación explicaremos cómo se realiza la actualización de los pesos, una vez obtenido el output a través de (1). Primero definiremos una función de error $\mathcal{E}$ que nos dará cuenta de la discrepancia entre el valor calculado y el real. Sea N el número de outputs, d_i el output deseado y y_i el obtenido, con $i \in \{1, \ldots, N\}$. Entonces para un perceptrón simple

$$\mathcal{E} = \frac{1}{2} \sum_j (d_j - y_j)^2, \quad (2)$$

El error total será simplemente $\mathcal{E}_{total} = \sum_{k=1}^{N} \mathcal{E}_k$. Nuestro objetivo será minimizar este error total, para ello utilizamos un algoritmo de gradiente descendente. Este tipo de algoritmo busca un mínimo en la función (la función error en nuestro caso) dando pasos proporcionales al negativo del gradiente. Es por ello que es tan importante que la función que usamos como función de activación sea derivable.

Así, el ajuste de los pesos es proporcional a la derivada

$$\Delta w_{ij}^k = -\eta \frac{\partial \mathcal{E}}{\partial w_{ij}^k} \quad (3)$$

donde η es el tamaño del paso, importante para asegurar la convergencia. También se puede añadir un término de inercia, mediante una dependencia de recursividad, que nos ayudará a mejorar la convergencia evitando rápidos cambios en Δw_{ij}^k.

$$\Delta w_{ij}^k(t) = -\eta \frac{\partial \mathcal{E}}{\partial w_{ij}^k(t)} + \alpha \Delta w_{ij}^k(t-1) \quad (4)$$

donde α es un positivo real pequeño.

Estrictamente hablando, la $\mathcal{E}$ que se utiliza en (3) debería ser $\mathcal{E}_{total}$. Sin embargo es mucho más práctico actualizar pesos con cada input de una muestra de entrenamiento, en vez de usar toda la muestra al completo. Estos dos casos se denominan batch y online, respectivamente. En este caso puede utilizarse el $\mathcal{E}$ definido en (2).

Así, para cada capa de un perceptrón simple, tendríamos el output dado por

$$y_j = f(\sum_{i=1}^{n-1} w_i I_i + \theta_j) = f(s)$$

El sesgo θ_j puede añadirse como input siempre activo ($I_n = 1$) con peso $w_n = \theta_j$

$$y_j = f(\sum_{i=1}^{n} w_i I_i) = f(s)$$

Si el ouput deseado es d_j, entonces de (2) tenemos

$$\frac{\partial \mathcal{E}}{\partial w_i} = \frac{\partial \mathcal{E}}{\partial y_j} \frac{\partial y_j}{\partial w_i}$$

$$= -(d_j - y_j) f'(s) I_i$$

Sea

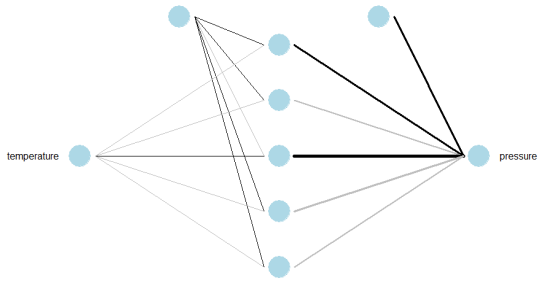
$$\delta_j = (d_j - y_j)$$

Entonces el ajuste de pesos viene dado por

$$\Delta w_i = -\eta \frac{\partial \mathcal{E}}{\partial w_i} = \eta \delta_j f'(s) I_i$$

Este esquema de ajuste de pesos es denominado la regla Delta. El algoritmo de Back-Propagation en realidad es una generalización de la regla Delta.

Temperature	Pressure
0	0.0002
20	0.0012
40	0.0060
60	0.0300
80	0.0900
100	0.2700
120	0.7500
140	1.8500
160	4.2000
180	8.8000
200	17.3000
220	32.1000
240	57.0000
260	96.0000
280	157.0000
300	247.0000
320	376.0000
340	558.0000
360	806.0000



Sea una red MLP, con N capas y suponemos que nuestra función de activación f es la sigmoide logística, entonces

$$\frac{\partial \mathcal{E}}{\partial y_j^N} = -(d_j - y_j^N)$$

Donde el superíndice N indica que el output es de la capa N y de (1) obtenemos

$$\frac{\partial y_j^N}{\partial x_j^N} = f'(x_j^N) = y_j^N(1 - y_j^N).$$

Como $x_j^{k+1} = \sum_i w_{ij}^k y_i^k$, entonces

$$\frac{\partial x_j^N}{\partial w_{ij}^{N-1}} = y_i^{N-1}$$

Y por lo tanto, podemos escribir la parcial del error respecto de los pesos como

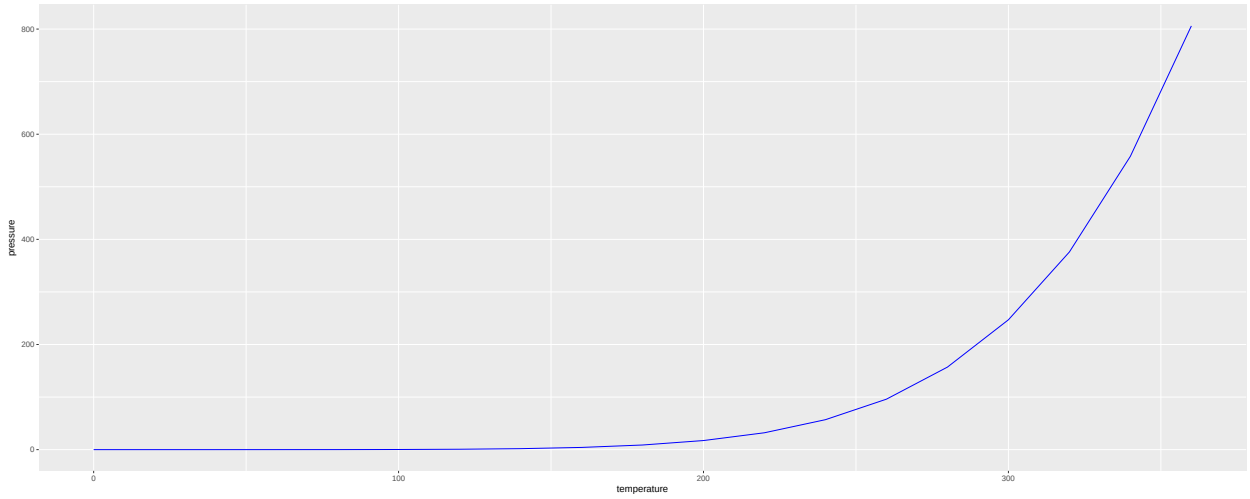
$$\frac{\partial \mathcal{E}}{\partial w_{ij}^{N-1}} = \frac{\partial \mathcal{E}}{\partial y_j^N}\frac{\partial y_j^N}{\partial x_j^N}\frac{\partial x_j^N}{\partial w_{ij}^N}$$
$$= -(d_j - y_j^N)y_j^N(1 - y_j^N)y_i^{N-1}$$

Sea

$$\delta_j^{N-1} = (d_j - y_j^N)y_j^N(1 - y_j^N) \quad (5)$$

Entonces, teniendo en cuenta (4) y (5) tenemos que en la iteración $t+1$ los pesos serán

$$w_{ij}^{N-1}(t+1) = w_{ij}^{N-1}(t) + \eta\delta_j^{N-1}y_i^{N-1} + \alpha[w_{ij}^{N-1}(t) - w_{ij}^{N-1}(t-1)] \quad (6)$$



$$\forall i,j : \Delta_{ij}(t) = \Delta_0$$
$$\forall i,j : \frac{\partial \mathcal{E}}{\partial w_{ij}(t-1)} = 0$$

```
GHMM-ANN = function(returns, w, p, n_states=2, n_samples=10, Tolerance=7*10^{-2}){

  sample=returns
  n_samples=10
  TL = length(sample)

  #STDP Rule
  w_change = function(w,L, n_neurons, n_states, stdp){
      aux=list(data.frame())
      for(i_1 in 1:L){
        aux[[i_1]]=w
        if(i_1 %in% stdp[[1]]){poisson_train=m[,i_1]}else{poisson_train=rev(m[,i_1])}
          for(i_2 in 1:n_trains){
            if(poisson_train[i_2]==1){aux[[i_1]][i_2,]=
            exp(-aux[[i_1]][i_2,]+1)-1}else{aux[[i_1]][i_2,]=rep(-1,n_inputs)}
        }
    }
  return(aux)
  }

  p_new=p
  w_new=w
  for (s in 1:n_samples){
  p=p_new
  w=w_new
  #Sample
  returns=sample[seq((s-1)*trunc(TL/n_samples)+1,s*trunc(TL/n_samples)),]


  returns=as.data.frame(returns)
  L=length(returns)
  for(i in 1:n_samples){samples[[i]]= returns[((i-1)*round(L/n_samples)+1):(i*round(L/n_samples)),]}
```

```
  w = as.data.frame(w)
  pi=rep(1/n_states, n_states)
  #pi=c(0.6,0.4)


  m=matrix(nrow = n_neurons, ncol = L)
  R=list(as.data.frame(m), as.data.frame(m))
  R[1:n_states]=list(data.frame(c(rep(0,L))))
  iteration=1

  #Generate Poisson Train
  n_trains=n_neurons*n_states
  high_freq=n_trains*.75
  low_freq=n_trains/10
  df=1/n_trains
  m = matrix(ncol = L, nrow = n_trains)
  for(i in 1:L){
    m[,i] = as.double(runif(n_trains)<ifelse(i %in% stdp[[1]], high_freq*df, low_freq*df))
  }
  eta=apply(m, 1, function(x)sum(x))

  #Baum-Welch Adaptation
  for(i in 1:n_states){
    for(j in 1:L){
      aux=pi[i]*exp(p[seq(((i-1)*n_inputs+1),i*n_inputs),1])*
                exp(apply(w_new[[j]][seq(((i-1)*n_inputs+1),i*n_inputs),]*
                as.double(returns[j,]),1,function(x)sum(x)))
      R[[i]][j,] = aux/sum(aux)
    }
  }
  aux=returns
  mu=list()
  for(i in 1:n_states){
    for(j in 1:L){
      aux[j,] = R[[i]][j,]*returns[j,]
    }
    mu[[i]]=sum(aux[j,])/sum(R[[i]])
    pi[[i]]=sum(R[[i]])/nrow(R[[i]])
    pi=pi/sum(pi)
  }


    w_new=w_change(w,L,n_neurons, n_states, stdp)
    for(i in 1:L){w_new[[i]]= mu[[i]] + 0.01*eta*w_new[[i]]}
    p_new=pi*exp(-p+1)-1
  }

return(list(mu=mu, pi=pi, p = p))
}
```