Redes de Elman y Jordan

García Prado, Sergio

16 de enero de 2017

Resumen

Este trabajo consiste en el estudio de las redes neuronales recurrentes (RNN) de Elman *y* Jordan*. En este documento se describen las fases de dichos algoritmos así como las diferencias entre ellos. Además, se ha realizado una implementación en el lenguaje* Octave *basada en la aproximación a una función temporal de valores escalares procedentes de los resultados de cotización en bolsa de una empresa.*

I. Introducción

Las **redes neuronales** consisten en una técnica, que se basa en una gran conjunto de neuronas artifiales. Cada unidad neuronal está conectada con otras de una determinada manera. Estos enlaces pueden poseer una función de combinación. Cada neurona puede tener una función encargada de combinar los valores de su entrada. Además, puede haber una función de activación en cada enlace o en la propia neurona: de tal manera que la señal debe superar el límite antes de la propagación hacia otras. Estos sistemas tienen capacidad de autoaprendizaje. Esto se ilustra gráficamente en la figura 1.

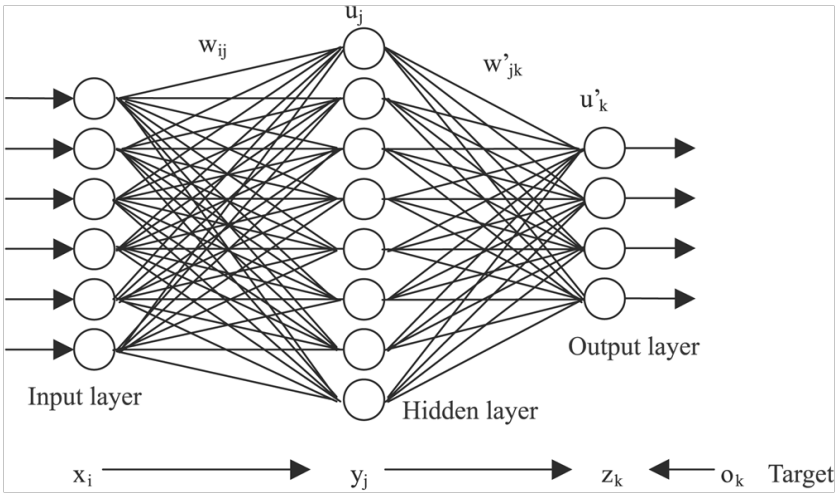

Figura 1: *Topología de una red neuronal.*

En este documento se va a profundizar en las **redes neuronales recurrentes**. Estas son un tipo concreto de redes neuronales cuya principal caracterísa es la interconexión recurrente de neuronas. Esto significa que una neurona puede poseer ciclos con otras neuronas, o incluso con ella misma. Esta idea fue desarrollada por *Jordan* en *Serial Order: A parallel distributed approach*(1986)[1] y por *Elman* en *Finding structure in time*(1990)[2].

Por otro lado, el conjunto de datos que se ha utilizado como ejemplo consiste en un conjunto de 254 casos correspondientes a los valores de cotización en bolsa de la empresa **Iberdrola** durante un periodo de 12 meses (Noviembre de 2015 - Diciembre de 2016). En este fichero aparecen vaios

campos, a pesar de ello nos hemos centrado únicamente en el valor de *Cierre*, que es el que se ha utilizado como entrada para la red neuronal.

II. Redes de Elman y Jordan

En sus respectivos artículos, tanto *Elman* como *Jordan* proponen una red neuronal con estructura de **3 capas** correspondiendose la primera a la *capa de entrada*, la segunda a la *capa oculta* y la tercera a la *capa de salida*. Además de esto, proponen añadir un conjunto de neuronas (que denominan *capa de contexto*) y las cuales pretenden dotar a la red de capacidad de recuerdo, permitiendo que estas almacenen información acerca de casos anteriores.

Las diferencias entre estas dos propuestas consisten en la topología de la red, en concreto, en las interconexiones con la *capa de contexto*. Por tanto, la diferencia es la siguiente:

- **Red de Jordan**: La capa de contexto está conectada con la **capa de salida**.
- **Red de Elman**: La capa de contexto está conectada con la **capa oculta**.

Esto tiene incidencia en las fases de propagación sobre la red (de manera explicita hacia delante y de manera implicita hacia atrás), ya que determina a partir de qué conjunto de neuronas se almacena información en la capa de contexto. Una representación gráfica de estas diferencias se muestra en la figura 2

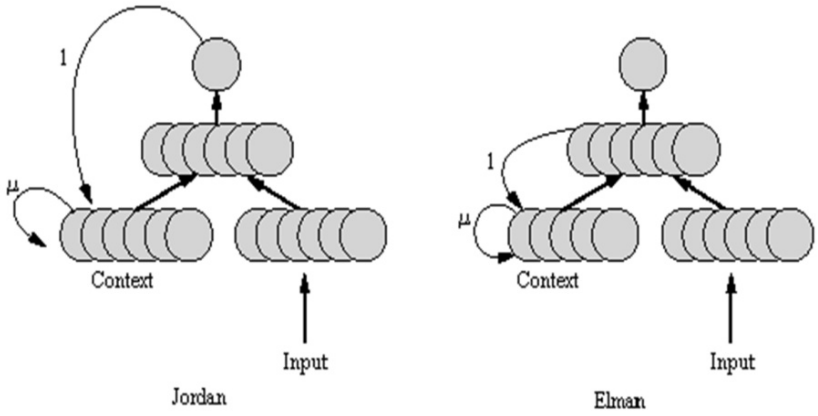

Figura 2: *Topología de las redes neuronales recurrentes de Elman y Jordan.*

Obviando esta diferencia topológica, el resto de la red funciona de la misma forma, por tanto, a continuación se describen el conjunto de pasos que suceden. A continuación se describe el significado de las variables utilizadas en la formulación matemática:

- $y(t)$ = valores de las neuronas en el periodo t
- $f_H()$ = función sigmoide.
- $\partial f_H()$ = derivada de la función sigmoide.
- $x(t)$ = valores de entrada en el periodo t.
- $e(t)$ = error en el periodo t.
- $W(t)$ = matriz de pesos en el periodo t.

- $dW(t)$ = diferencial de pesos en el periodo t.
- l_{rate} = ratio de aprendizaje.
- m_{rate} = ratio de importancia de casos anteriores.

I. Encaminamiento hacia adelante

El encaminamiento hacia adelante se corresponde con la fase de propagación de los datos desde la capa de entrada hacia la capa de salida ajustandose a los pesos que tiene cada enlace. Este algoritmo es diferente para la red cada una de las dos redes neuronales, por tanto se describe su formulación matemática de manera separada ya que las diferencias en las interconexiones producen variaciones en cuanto a los índices de las mismas. A pesar de ello, la idea es común en ambas: propagar los valores según la repercusión que tengan los pesos en cada caso almacenando en la capa de contexto el contenido de la capa oculta en el caso de *Elman* y la capa de salida en el caso de *Elman*

I.1. Encaminamiento hacia adelante: Elman

$$y_i^1(t+1) = f_H(\sum_{j=0}^{N}[0]w_{ij}^1 y_j^0(t+1)) \tag{1}$$

$$y_j^0(t+1) = f_H(\sum_{j=0}^{N}0w_{jk}^0 x_k(t+1) + \sum_{j=0}^{N}[0]w_{j,N0+i}^0 y_i^0(t)) \tag{2}$$

I.2. Encaminamiento hacia adelante: Jordan

$$y_i^1(t+1) = f_H(\sum_{j=0}^{N}[0]w_{ij}^1 y_j^0(t+1)) \tag{3}$$

$$y_j^0(t+1) = f_H(\sum_{j=0}^{N}0w_{jk}^0 x_k(t+1) + \sum_{j=0}^{N}[1]w_{j,N0+i}^0 y_i^1(t)) \tag{4}$$

II. Encaminamiento hacia atrás

La fase de encaminamiento hacia atrás se corresponde con la fase de retropropagación de los errores surgidos durante la fase de encaminamiento hacia adelante de la red para así tratar de minimizar el error en futuras iteracciones y producir el aprendizaje de la red.

$$e(t) = x(t) - y_1(t) \tag{5}$$

$$\delta_1(t) = e * \partial f_H(y_1(t)) \tag{6}$$

$$\delta_0(t) = (\delta_1(t) * w) * \partial f_H(y_0(t)) \tag{7}$$

$$d_w(t) = x(t) * \delta(t) \tag{8}$$

$$w(t) = l_{rate} * \delta(t) + m_{rate}\delta(t-1) \tag{9}$$

III. Implementación

La sección de implementación se divide en un primer apartado que explica la ejecucción del código de ejemplo seguido de los apartados que explican en detalle cada una de las partes del código fuente

I. Guía de ejecucción

La implementación realizada se ha llevado a cabo en el lenguaje Octave. Se incluyen todos los ficheros de código fuente para poder ejecutar la implementación. Estos se alojan en el directorio /**src**. Además, se ha añadido un script bash de prueba denominado **run.sh** el cual permite ejecutar la implementación a partir del terminal si así se desea.

También se puede iniciar desde el entorno gráfico de Octave asignando el directorio /**src** como directorio de trabajo y seguidamente llamando a la funcion **main()**.

II. Datos de Entrada

Tal y como se ha expuesto brevemente al principio de este documento, el conjunto de datos se corresponde con una serie temporal de valores escalares. Además, se indica que tan solo los **últimos 20 valores** tienen repercusión en el valor próximo que se desea predecir. Este hecho plantea dos enfoques diferentes de entrada hacia la red neuronal:

- **Enfoque Secuencial**: Para cada periodo de aprendizaje o predicción previamente se introducen las 20 muestras previas de manera secuencial para después obtener la siguiente. Esto implica que la red tenga una única neurona en la entrada.
- **Enfoque en Paralelo**: Para cada periodo de aprendizaje o predicción se introducen las 20 muestras previas de una sola vez para después obtener la siguiente. Esto implica que la red tenga tantas neuronas en la entrada como muestras previas relevantes, en este caso 20.

Tras realizar pruebas utilizando las dos alternativas no se encontraron diferencias significativas en los resultados obtenidos por ninguna de las dos estrategias, por lo cual se ha optado por el **Enfoque en Paralelo** debido a la equivalencia de resultados y su menor coste computacional.

En cuanto a la *capa oculta* se ha escogido un tamaño de **10 neuronas** basandose en las pruebas realizadas así como en el artículo *Financial Time Series Prediction Using Elman Recurrent Random Neural Networks*[3].

III. Normalización

La red neuronal está diseñada para trabajar con valores normales (en el rango $[0, 1]$) debido a la función sigmoide que se define en el siguiente apartado. Por lo tanto, es necesario realizar una fase previa de normalización de los mismos. Debido a la restricción que indica que la relevancia de sucesos pasados se limita a los 20 últimos casos, se normaliza sobre estos en cada fase del algoritmo.

Para ello se utilizan las siguientes funciones para normalización y desnormalización, que colapsan el conjunto en torno a su máximo y mínimo:

$$S(t)' = fracS(t) - minS(t)maxS(t) - minS(t) \tag{10}$$

$$S(t) = S(t)'(maxS(t) - minS(t)) + minS(t) \tag{11}$$

```
function y = normalize(x, max, min);
  y = (x .- min) ./ (max - min);
end;
```

Figura 3: *Octave: /src/common/normalize.m*

```
function y = denormalize(x, max, min);
  y = x .* (max - min) .+ min;
end;
```

Figura 4: *Octave: /src/common/denormalize.m*

IV. Función Sigmoide

La función sigmoide o función de activación es la encargada de determinar el grado de importancia que tendrá un determinado nodo de la red en el valor del siguiente. A continuación se formula la función logística así como su derivada:

$$f_H(x) = \frac{1}{1 + e^{(}- x)} \tag{12}$$

```
function y = sigmoid(x)
  y = 1 ./ (1 .+ (e .^ (0.-x)));
end;
```

Figura 5: *Octave: /src/common/sigmoid.m*

$$\partial f_H(x) = \frac{e^x}{(1 + e^x)^2} \tag{13}$$

```
function y = dsigmoid(x)
  y = e .^ x ./ (1 .+ e .^ x) .^2;
end;
```

Figura 6: *Octave: /src/common/dsigmoid.m*

V. Inicialización

La fase de inicialización se corresponde con la creación de la matriz de capas así como la inicialización de la matriz de pesos, que en este caso se han decido generar aleatoriamente en el rango $[-0{,}25, 0{,}25]$.

La función de generación de pesos aleatorios se muestra en la figura 7 y es común a los dos métodos (*Elman* y *Jordan*)

```
function W = random_weights(x, y)
  W = [ (2 .* rand(x, y) .- 1) * 0.25 ];
end;
```

Figura 7: *Octave: /src/common/random_ weights.m*

Por contra, la generación de la estructura de la red es diferente debido a las diferencias topológicas entre las dos redes. El código fuente utilizado para estas partes se muestra en las figuras 10 y 9 en el caso de *Elman* y en las figuras 10 y 11 en el caso de *Jordan*.

```
function L = elman_generate_layers(shape)
  L{1} = [ones(shape(1) + 1 + shape(2),1)];
  for i = 2:size(shape,2);
    L{i} = [ones(shape(i), 1)];
  end;
end;
```

Figura 8: *Octave: /src/elman/elman_generate_layers.m*

```
function W = elman_generate_weights(shape)
  W{1} = random_weights(shape(1) + shape(2) +1, shape(2));
  for i = 2:size(shape,2)-1;
    W{i} = random_weights(shape(i), shape(i+1));
  end;
end;
```

Figura 9: *Octave: /src/elman/elman_generate_weights.m*

```
function L = jordan_generate_layers(shape)
  L{1} = [ones(shape(1) + 1 + shape(size(shape,2)) ,1)];
  for i = 2:size(shape,2);
    L{i} = [ones(shape(i), 1)];
  end;
end;
```

Figura 10: *Octave: /src/jordan/jordan_generate_layers.m*

```
function W = jordan_generate_weights(shape)
  W{1} = random_weights(shape(1) + shape(size(shape,2)) +1, shape(2));
  for i = 2:size(shape,2)-1;
    W{i} = random_weights(shape(i), shape(i+1));
  end;
end;
```

Figura 11: *Octave: /src/jordan/jordan_generate_weights.m*

VI. Propagación hacia adelante

La fase de propagación hacia adelante, tal y como se ha explicado en la sección anterior, difiere según el tipo de red dado que las interconexiones entre neuronas son diferentes. Por tanto, el código fuente de la versión de *Elman* se ilustra en la figura 12 y en el caso de *Jordan* en la figura 13.

```
function [L, O] = elman_forward(L, W, S, I)
  % L = layers, W = weights, S = shapes
  % I = input, O = output

  L{1}(1:S(1)) = I;

  L{1}(S(1)+2:size(L{1},1)) = L{2};

  for i = 2:size(S, 2);
    L{i} = sigmoid((L{i-1}' * W{i-1})');
  end;

  O = L{size(S,2)};
end;
```

Figura 12: *Octave: /src/elman/elman_forwards.m*

```
function [L, O] = jordan_forward(L, W, S, I)
  % L = layers, W = weights, S = shapes
  % I = input, O = output

  L{1}(1:S(1)) = I;
  L{1}(S(1)+2:size(L{1},1)) = L{size(L,2)};

  for i = 2:size(S, 2);
    L{i} = sigmoid((L{i-1}' * W{i-1})');
  end;

  O = L{size(S,2)};
end;
```

Figura 13: *Octave: /src/jordan/jordan_forward.m*

VII. Propagación hacia atrás

Respecto de la fase de propagación hacia atrás, en la cual se trata de corregir el error hacia el valor esperado, las dos redes neuronales recurrentes comparten la misma implementación, que se corresponde con la de la figura 14.

VIII. HoldOut

Para realizar tests de rendimiento en cuanto a la tasa de aciertos de las implementaciones, se ha utilizado el método de *HoldOut*. Para crear las particiones se ha decidido generar 2 subconjuntos de índices respecto del conjunto global de datos, con los cuales particionar la muestra en la parte de entrenamiento así como en la de prueba. Para esto se ha utilizado la función de la figura 15.

```
function [W, dw_g, e] = backward(L, W, S, dw_g, I, lrate=0.1, momentum=0.1)
  % L = layers, W = weights, S = shapes, I = input, e = error

  e = I - L{end};
  delta = e .* dsigmoid(L{end});
  deltas = {};
  deltas{size(S,2)-1} = delta;
  for i = size(S,2)-2:-1:1;
    deltas{i} = (deltas{size(S,2)-1} * W{i})' * dsigmoid(L{i});
  end;
  for i = 1:size(W,2);
    dw = L{i} * deltas{i}';
    W{i} += lrate * dw + momentum * dw_g{i};
    dw_g{i} = dw;
  end;
  e = sum(e .^ 2);
end;
```

Figura 14: *Octave: /src/common/backward.m*

```
function [L, T] = generate_hold_out(A, frac = 2/3);
  % A = index set
  % frac = fraction of division
  % L = learning index set
  % T = test index set
  L = randperm(A, round(frac * A));
  T = setdiff([1:A], L);
end;
```

Figura 15: *Octave: /src/common/generate_hold_out.m*

IV. Resultados

Para el análisis de resultados de esta implementación se ha utilizado, como se ha comentado previamente, un método de particionamiento de *HoldOut* en dos particiones (entrenamiento y test). Este método de particionamiento se caracteriza por generar subconjuntos disjuntos, es decir, cada uno de los datos pertenece a 1 y solo 1 de los dos subconjuntos. El particionamiento se ha seguido utilizando la **regla de los** 2/3. Esto consiste en utilizar 2/3 del conjunto para entrenar la red neuronal mientras que el 1/3 restante se utiliza para realizar los tests.

Dado que el conjunto de datos se corresponde con una serie temporal, y sabemos que tan solo tienen repercusión en los resultados los últimos 20 valores, se ha decidido que los casos sean solapamientos de estos junto con el valor esperado. Sea X el conjunto de valores de entrada y D el conjunto de muestras de prueba. Por tanto, los conjuntos de datos se han formado de la siguiente manera:

$$x_i \in X, i \in [1, 254] \tag{14}$$

$$d_j \in D, j \in [1, 233] \tag{15}$$

- $d_1 = [x_1, x_2, ...x_{20}, x_{21}]$
- $d_2 = [x_2, x_3, ...x_{21}, x_{22}]$
- $d_3 = [x_3, x_4, ...x_{22}, x_{24}]$
- ...
- $d_{233} = [x_{233}, x_{234}, ...x_{253}, x_{254}]$

Los resultados de la tasa de aciertos de la implementación no han sido muy satisfactorios ya que no llegan a superar el 50 % salvo casos especiales. En la gráfica de la figura 16 se ilustra la evolución de la tasa de aciertos según aumenta el número de muestras que han realizado el test.

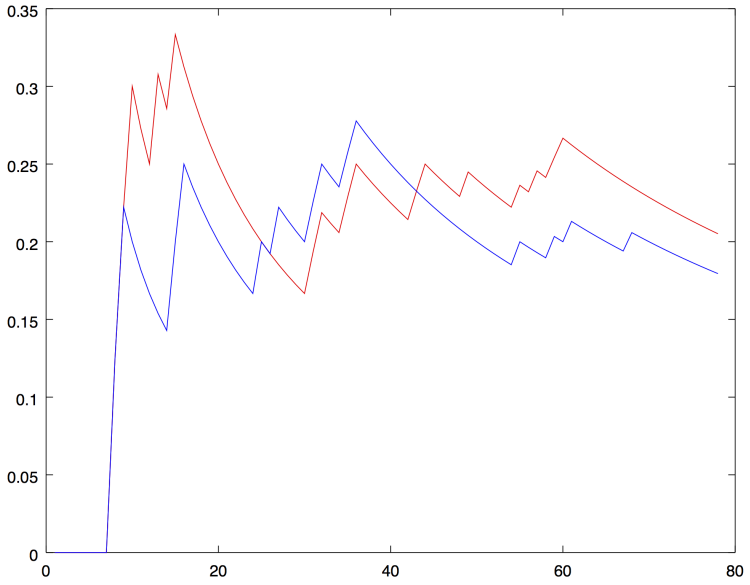

Figura 16: *Resultados: rojo =>Elman, azul =>Jordan*

Referencias

[1] Jordan: Serial Order: A distributed parallel approach (1986)

[2] Elman: Finding Structure in Time (1990)

[3] Jie Wang, Jun Wang, Wen Fang, Hongli Niu: Financial Time Series Prediction Using Elman Recurrent Random Neural Networks (2016) `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4887655/`

[4] Carlos Alonso Gonzalez, Teodoro Calonge Cano: Minería de datos(Uva), Redes Recurrentes (2016)