Práctica 3

# Neurocomputación

*Daniel Garnacho Martín*
*Pablo Molins Ruano*

# Autoencoder

## Entrenamiento y clasificación sin ruido

La red neuronal es capaz de aprender todas las letras usando un número determinado de neuronas en la capa oculta, en nuestro caso a partir de 8 neuronas, la red es capaz de proporcionar la salida correcta, sin ningún fallo.
Para la realización de este ejercicio se ha tenido que modificar la red neuronal dado que antes la entrada era en formato de clases y ahora no es así, los cambios son mínimos pero necesarios, los cambios competen al vector objetivo de la red neuronal.
El código se ejecuta desde main_autoencoder.py

El error medio calculado es $\frac{1}{\#Instancias}\sum_i \#bitsErroneos_i$

| #Neuronas | #Aprendidas | #No aprendidas | Aprendidas | No aprendidas | Error Medio |
|---|---|---|---|---|---|
| 1 | 0 | 26 | | Todas | 7.653.846.154 |
| 3 | 2 | 24 | H, O | | 3.115.384.615 |
| 5 | 13 | 13 | F, H, I, L, N, O, P, S, T, V, W, X, Y | A, B, C, D, E, G, J, K, M, Q, R, U, Z | 0.5769230769 |
| 7 | 24 | 2 | | A, Q | 0.07692307692 |
| 8 | 26 | 0 | Todas | | 0 |
| 10 | 26 | 0 | Todas | | 0 |

## Entrenamiento sin ruido, clasificación con ruido

En la siguiente tabla se puede ver como el Autoencoder es capaz de minimizar errores pequeños, si no fuera capaz el error medio tendería a F siendo F el número de entradas modificadas, se puede ver como es capaz de acertar con K = 1, 260 letras sin ningún error y solo fallar 26, por ejemplo para K = 5 esta tendencia se da la vuelta siendo mayor el número de fallos.

| F | Error Medio | Letras falladas | Letras acertadas |
|---|---|---|---|
| 1 | 0.1013986014 | 26 | 260 |
| 3 | 0.9615384615 | 135 | 151 |
| 5 | 1.947552448 | 198 | 88 |
| 7 | 2.891608392 | 224 | 62 |
| 10 | 4.678321678 | 249 | 37 |

## Entrenamiento con ruido, clasificación con el mismo ruido

Con esta fase buscamos encontrar el número de neuronas en la capa oculta correcto, este número se ha fijado en 12, generamos 5 letras aleatorias desde cada letra anterior, el error de clasificación con F = 5 es 0.

## Entrenamiento con ruido, clasificación con ruido nuevo.

Se ha optado por un conjunto de clasificación que es el de entrenamiento más las nuevas letras generadas aleatoriamente, esta opción nos supone que no se pueden medir de igual manera las errores obtenidos, en igual manera hemos cometido este fallo al obtener las medidas anteriores, en esta sección tratamos de asemejar los errores para poder compararlos.

La tabla sin normalizar es:

| F | Error Medio Train | Letras falladas Train | Letras acertadas Train | Error Medio Test | Letras falladas Test | Letras acertadas Test |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 156 | 0.05528846154 | 21 | 395 |
| 3 | 0 | 0 | 156 | 0.5865384615 | 109 | 307 |
| 5 | 0.03846153846 | 6 | 150 | 1.463942308 | 191 | 225 |
| 7 | 0.01282051282 | 2 | 154 | 2.211538462 | 204 | 212 |
| 10 | 0.1923076923 | 22 | 134 | 3.53125 | 249 | 167 |

Los errores normalizados de las dos tablas para poderlos comparar son:
Sólo se muestran los errores de clasificación, y sólo de las instancias de clasificación, EM no ruido son los datos del apartado anterior eliminando las instancias de entrenamiento, EM ruido Son los datos de la tabla superior a los que se les ha eliminado el error de entrenamiento y las instancias de entrenamiento, la fórmula es: ((EM test*#InstanciasTest)-(EM train*#InstanciasTrain))/(#InstanciasTest-#InstanciasTrain)

| F | EM no ruido | EM ruido |
|---|---|---|
| 1 | 0.1115384615 | 0.08846153846 |
| 3 | 1.057692308 | 0.9384615385 |
| 5 | 2.142307692 | 2.319230769 |
| 7 | 3.180769231 | 3.530769231 |
| 10 | 5.146153846 | 5.534615385 |

Se puede ver como el ruido mejora levemente para K pequeña 1 o 3 y que empeora para K > 3, por lo que podríamos haber evitado entrenar con esas instancias con ruido.

# Series temporales

## Modificaciones hechas en el código

### Función `adapta-fichero-serie`:

Esta función la hemos implementado en `src/RW/adaptaFicheroSerie.py`. Aprovechando la versatilidad que ofrece Python, dicho fichero puede ejecutarse independientemente o incorporarse dentro del proyecto global. Esto es posible gracias a que el fichero se compone de un main y una función. Tanto si se ejecuta independiente como función dentro del proyecto, recibe como parámetro el fichero de entrada, el de salida, $N_a$ y $N_s$, en ese orden. La función se encarga de leer todo el fichero de entrada e ir escribiendo tantos datos como sea indicado en cada línea del nuevo fichero. Abrir y cerrar los ficheros es responsabilidad del módulo que llame a la función

### Cambio de la red para utilizar funciones lineales en la capa de salida:

Aprovechando el diseño modular que en la primera práctica implementamos, hemos creado un nuevo tipo de clasificador, al que hemos llamado red neuronal temporal. Se encuentra implementada en `src/Clasificadores/RedNeuronalTemporal.py`. Es como la red neuronal que ya tenemos implementada, con algunas modificaciones. En concreto, las discutidas en clase: sustituir la función de activación de la capa de salida por la función identidad, tanto en el entrenamiento como en el uso de la red, además de modificar la fase de entrenamiento ya que ahora la derivada es uno.

No fue necesario adaptar el código para que hubiera tantas neuronas de salida como $N_s$, pues nuestra red ya funcionaba así.

### Cambio del conjunto de entrenamiento para que se corresponda con los primeros datos del fichero.

En el módulo de particionado hicimos una modificación a la función principal, para que solo hiciera un shuffle de los datos de entrada si así se indicaba por un nuevo parámetro. Gracias a que Python permite definir en las funciones parámetros opcionales con valor por defecto, pudimos hacer esta modificación sin tocar el código anterior.

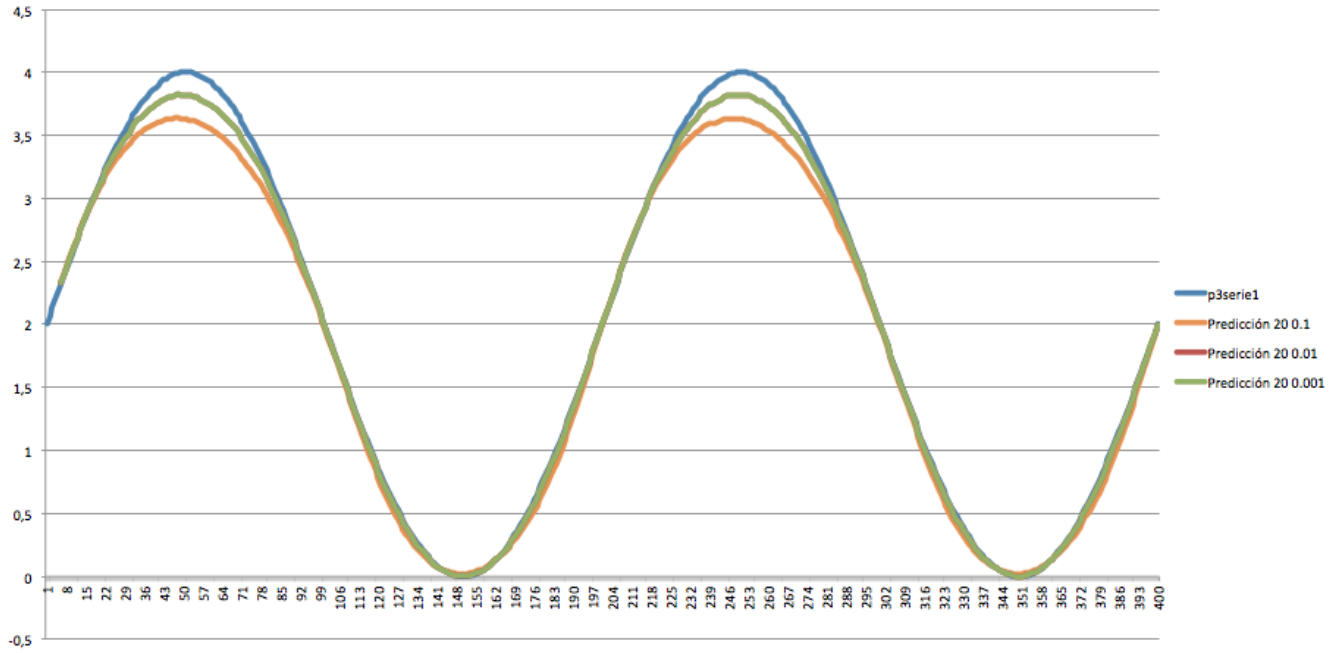
### Cómputo del ECM para el modelo de predicción más sencillo:

En `src/main_temporal.py` hemos incluido la función `calculaErrorSimple`que sencillamente compara el vector objetivo con el vector generado replicando el último valor del vector de entrada. Es muy similar a la función que ya existía de `calculaError`, salvando que no se le pide el vector al clasificador sino que se crea por el modelo sencillo.

## Resultados para la primera serie temporal

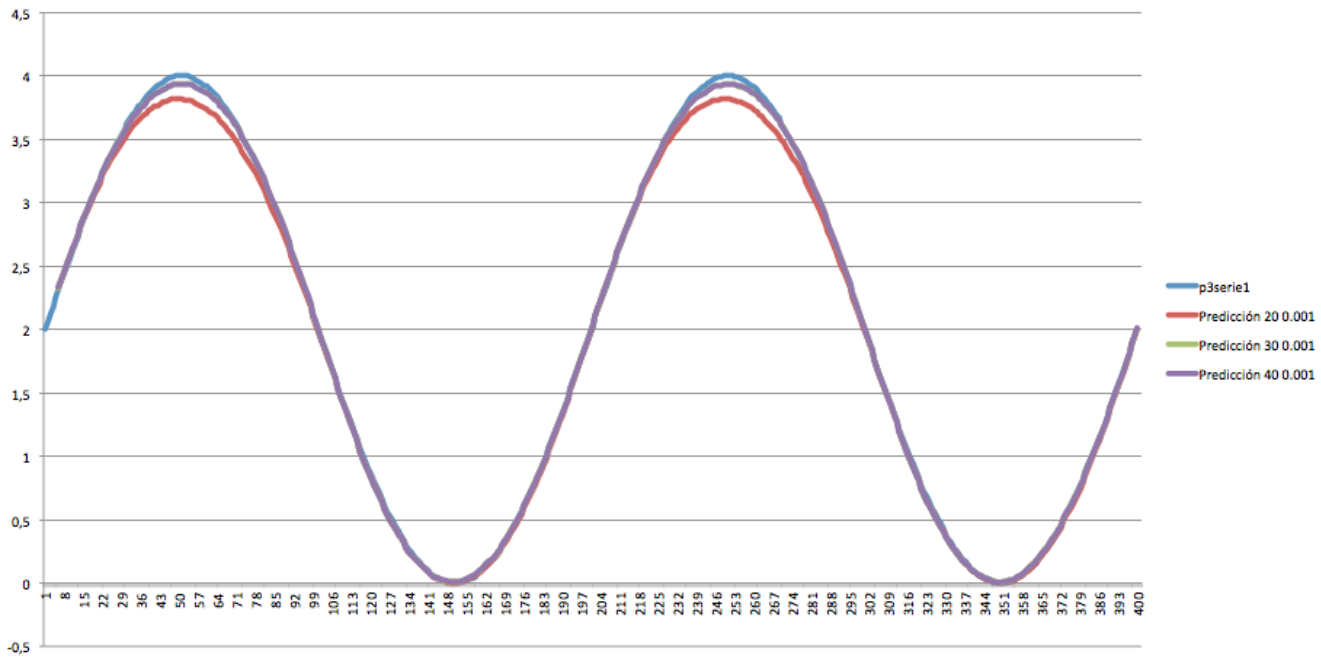
Al mostrar gráficamente los datos de la primera serie temporal en seguida vimos que era una secuencia periódica que se repetía 200 valores. Además, también vimos que se trataba de una función sinusoidal oscilando en torno al 2 con una amplitud de 2. Siendo una serie tan sencilla, en el sentido de que es periódica y bastante fácil de representar matemáticamente, supusimos que la red no tendría demasiado problema a la hora de modelizarla.

Las primeras pruebas que hicimos fue dividiendo el fichero de entrada en 200 casos de train y 200 casos de test. Dentro de esas pruebas, primero probamos distintos valores de la constante de aprendizaje buscando aquellos valores mejores. La siguiente gráfica recoge las pruebas realizadas para 20 neuronas, $N_a = 5$, $N_s = 1$, 1000 épocas y alfas desde 0.1 hasta 0.001:




Se puede ver que un alfa de 0.1 es notablemente peor que alfas menores, especialmente en las predicciones cercanas a los puntos de inflexión. Ahora bien, así mismo se puede observar que un alfa de 0.01 o de 0.001 arroja valores prácticamente idénticos, por lo que decidimos quedarnos con un alfa de 0.001 para las siguientes pruebas.

Determinado el mejor valor de la constante de aprendizaje, pasamos a probar el efecto que tiene en la red el número de neuronas. La siguiente gráfica muestra distintas pruebas en las mismas condiciones que en la gráfica anterior, solo que esta vez fijando el alfa a 0.001 y variando el número de neuronas:

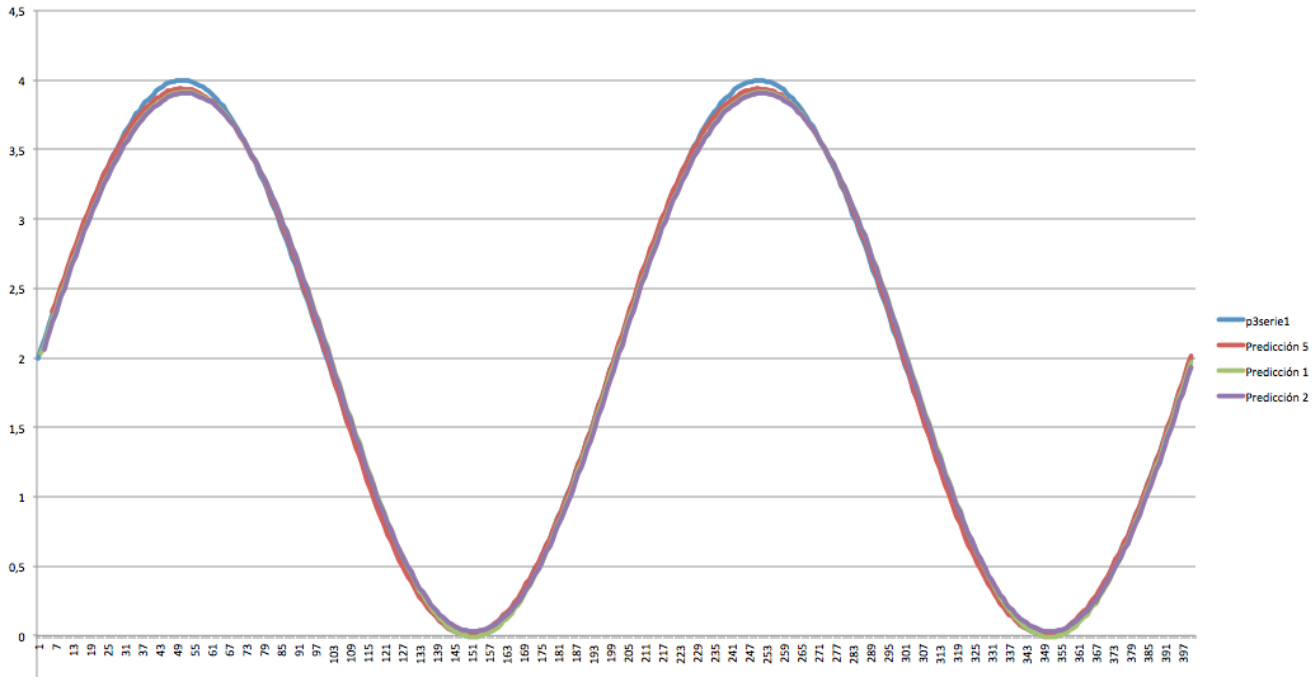



Aunque en este caso se nota una diferencia apreciable entre 20 neuronas y más, las diferencias en el resto de casos son demasiado sutiles como para que una gráfica permita apreciarlas correctamente. Así, decidimos que era mejor observar los valores del ECM para cada caso.

| # neuronas | ECM en train | ECM en test |
|---|---|---|
| 5 | 0.004599 | 0.004603 |
| 10 | 0.002714 | 0.002712 |
| 20 | 0.000686 | 0.000687 |
| 40 | 0.000445 | 0.000447 |
| modelo básico | 0.001985 | 0.001971 |

Viendo los resultados, podemos decir que el número mínimo para que nuestra red supere al modelo más básicos es 20 neuronas aunque con 10 obtenemos un valor superior pero muy cercano, por lo que si la eficiencia es importante, podríamos considerar 10 neuronas. Ahora bien, sin ningún tipo de restricción, las mejoras a mayor número de neuronas son evidentes.

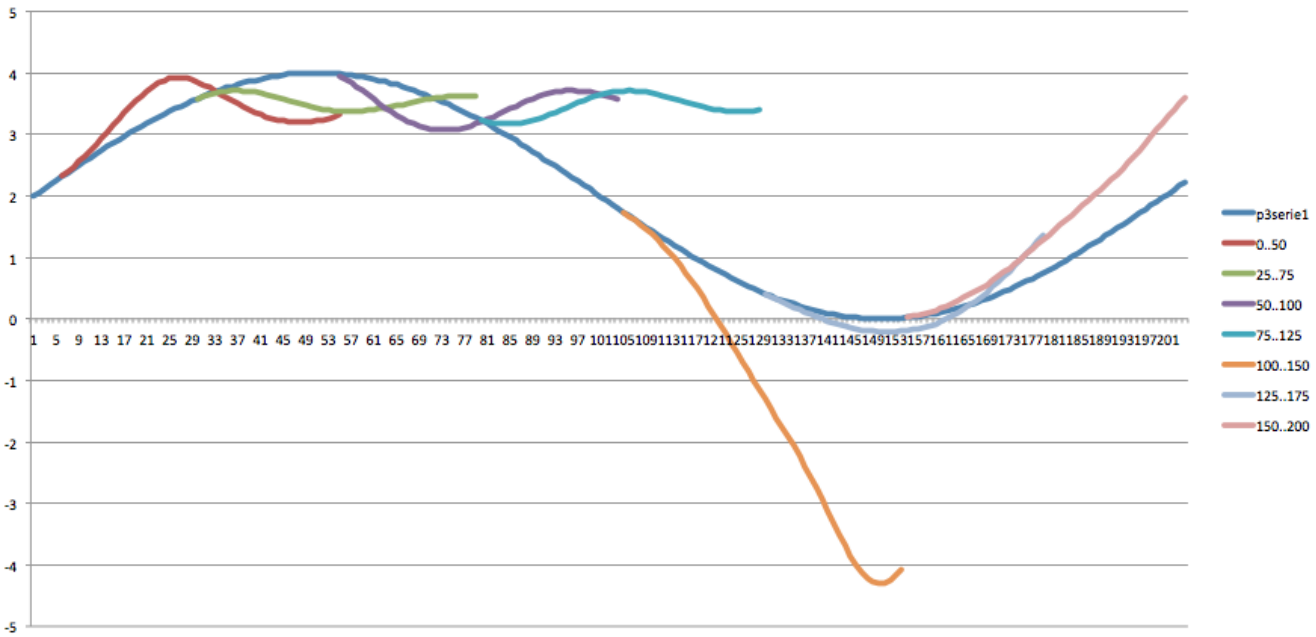
A continuación, observamos los efectos de variar $N_a$ en nuestra red. Para la prueba utilizamos 1000 épocas, 200 train y 200 test, 40 neuronas y cte. aprendizaje a 0.001. Los resultados fueron los siguientes:




| | $N_a = 1$ | $N_a = 2$ | $N_a = 5$ |
|---|---|---|---|
| ECM en test | 0.002860 | 0.004428 | 0.000447 |

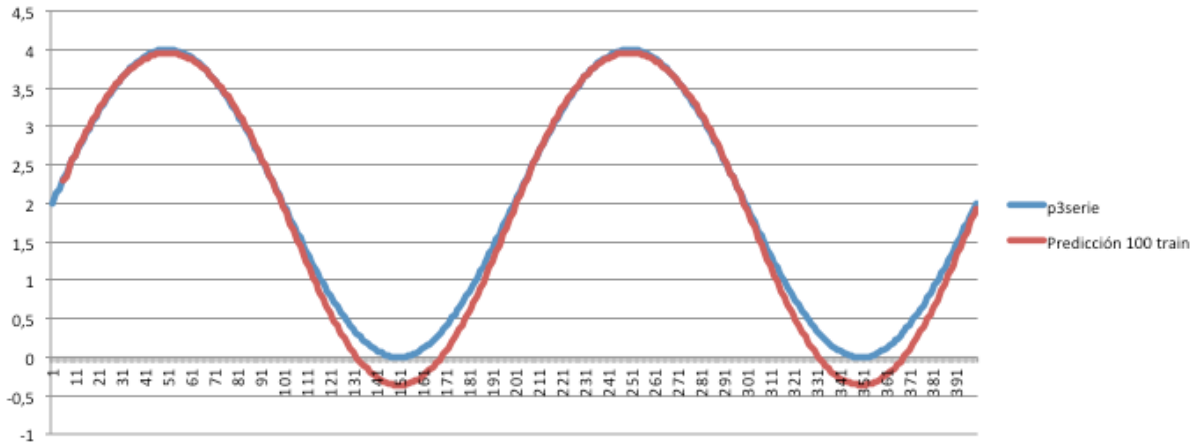
De nuevo, la tabla con los valores no es más útil que la gráfica. Y como es lógico, cuantos más valores antiguos mostramos, mejor predice nuestra red. Después de todo, es algo lógico, ya que para pocos valores es difícil saber si el siguiente debe ser menor o mayor que los anteriores, ya que en esta serie se repiten los mismos valores en subida y en bajada.

También probamos la capacidad de la red para hacer predicciones recursivas. Hemos utilizado los mismos parámetros (200 train, 40 neuronas, alfa 0.001), con $N_a = 5$ y $N_f = 50$



Aunque todas las series que hemos estudiado empiezan bien, todas terminan divergiendo en mayor o menor medida del comportamiento ideal. Algunas, como la 100..150 incluso llegan a valores completamente disparatados. Aún así, en todas las series se puede observar el componente sinusoidal que cabría esperar.

Por último, probamos a utilizar solo 100 patrones de train y 300 de test. Al ser una serie con periodo 200, nuestra hipótesis era que la red no sería capaz de aprender correctamente, ya que no le estaríamos mostrando más que la mitad del periodo.



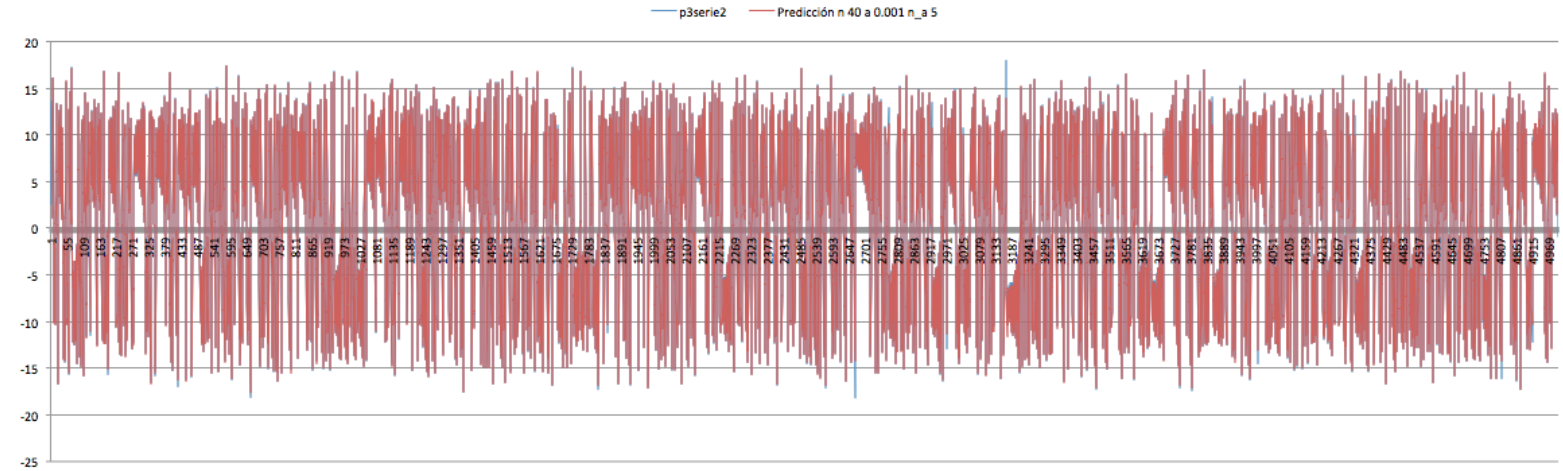

Al hacer la prueba comprobamos que habíamos infravalorado la capacidad de la red para predecir valores. Aunque se notan errores mayores en las zonas más bajas (que son las que no hemos mostrado a la red durante el entrenamiento) sigue teniendo una capacidad de predicción muy buena. De hecho, el ECM en test es solo 0.0510545.

## Resultados para la segunda red temporal

Al mostrar gráficamente los valores del segundo fichero ya no es tan fácil ver a qué corresponden. Esta serie resulta, al menos aparentemente, caótica. Hay fluctuaciones en torno a los valores superiores e inferiores, pero a la vez hay saltos bruscos desde valores altos hasta valores bajos.

Si sacamos el error que logra el modelo más básico, nuestras sospechas de que la serie es caótica parecen confirmarse. Si en la primera serie era un error mínimo, en esta la situación es bien distinta. En este caso el ECM para el modelo básico es de aproximadamente 50.

Para probar nuestra red con esta serie, empezamos utilizando los mejores parámetros del apartado anterior con el objetivo de ir ajustándolos después. Esperábamos unos resultados algo caóticos, pero de nuevo la capacidad de la red nos sorprendió:




Visualmente parece que la red predice perfectamente la serie. Atendiendo al ECM, también lo parece: 0.006958 en train y 0.044836 en test.

La siguiente prueba que hicimos fue variar el número de neuronas, para buscar el más óptimo. Aunque con cualquier número de neuronas logramos un resultado notablemente mejor que con el modelo básico, hasta 40 neuronas no logramos un valor decente. Si superamos ese número, logramos disminuir el ECM, pero no tanto para el incremento en tiempo que tenemos a la hora de calcular. Si miramos los datos, al pasar de 20 a 40 logramos reducir el ECM un 94%, mientras que de 40 a 80, que también es doblar, logramos tan solo una reducción del 68% pero el tiempo se mucho más que el doble.

| # neuronas | Alpha | $N_a$ | % train | ECM train | ECM test |
|---|---|---|---|---|---|
| 10 | 0,001 | 5 | 50% | 1.531115 | 1.706363 |
| 20 | 0,001 | 5 | 50% | 0.112014 | 0.239191 |
| 40 | 0,001 | 5 | 50% | 0.006958 | 0.044836 |
| 80 | 0,001 | 5 | 50% | 0.002184 | 0.027860 |

Variando el $N_a$ obtenemos también un resultado curioso, y es que no siempre aumentar el $N_a$ logra mejores resultados, como podemos observar comparando el caso de 7 y el de 9:

| # neuronas | Alpha | $N_a$ | % train | ECM train | ECM test |
|---|---|---|---|---|---|
| 40 | 0,001 | 2 | 50% | 1.212602 | 1.204220 |
| 40 | 0,001 | 5 | 50% | 0.006958 | 0.044836 |
| 40 | 0,001 | 7 | 50% | 0.008740 | 0.075633 |
| 40 | 0,001 | 9 | 50% | 0.049790 | 0.207250 |

Variando el porcentaje de train, vemos que la red es sensible a cuanto se le muestre, pero tampoco tanto como en el primer caso:

| # neuronas | Alpha | $N_a$ | % train | ECM train | ECM test |
|---|---|---|---|---|---|
| 40 | 0,001 | 7 | 10% | 0.012394 | 0.214420 |
| 40 | 0,001 | 7 | 30% | 0.009478 | 0.062209 |
| 40 | 0,001 | 7 | 50% | 0.008740 | 0.075633 |

Por último, para esta serie también comprobamos la capacidad de la red para predecir recursivamente. Para ello utilizamos $N_a$ = 7, $N_f$ = 50, 30% train, alfa = 0.001 y 40 neuronas: