Introducción

En el área de la informática, las redes neuronales (también conocidas como redes neuronales artificiales, RNA) son sistemas de neuronas artificiales interconectadas que se relacionan entre sí para generar una salida, a partir de una entrada dada.

Estos sistemas son utilizados de forma tal que hagan un aprendizaje automático. En muchos casos, una RNA bien implementada y con los parámetros correctos permite aprender muchos problemas en base a patrones de prueba, que serían muy difíciles o hasta imposibles de resolver de forma explícita.

En este caso, la temática abordada es la predicción de series temporales. Para ello se toma `x(t) = f(x(t-1), x(t-2), ...)` como el valor de la serie en el paso temporal `t`, que depende de los pasos previos de la misma.

La RNA que se busca es aquella que aproxime la función `x(t)`.

Redes Neuronales Artificiales

La idea principal de una RNA es que se cuenta con neuronas distribuidas por niveles. Cada neurona recibe estimulos de entrada y devuelve un estimulo de salida que se calcula aplicando una función de transferencia a la suma ponderada de sus entradas. El objetivo, es que al finalizar la etapa de entrenamiento, la red haya obtenido valores óptimos para los pesos de cada conexión, que satisfagan el problema.

Los RNA clasificadas como perceptrones simples permiten únicamente resolver problemas linealmente separables. Estas RNA solo cuentan con una capa de entrada y una de salida. Análogamente existen perceptrones multicapa que cuentan con capas ocultas que permiten resolver cualquier problema dada la arquitectura y los parametros correctos.

Una red neuronal obtiene una salida en base a una entrada dada, y la compara con la salida esperada. Si esta no es aceptable (se define un error aceptable) se recalculan los pesos de las conexiones mediante el algoritmo de *Backpropagation Learning*.

Definimos arquitectura de la red a la disposición de las neuronas, que determina tanto la cantidad de capas ocultas en la red, como la cantidad de neuronas por capa. Este parámetro es esencial ya que no todas las arquitecturas son capaces de modelar todos los problemas.

Otro parametro importante de una red neuronal es la función de transferencia. En general se usa una función sigmoidea.

Por último, hay que tener en cuenta los pesos (factores de ponderación de las entradas) inciales que se le da a cada neurona, el *learning rate* y las distintas optimizaciones que se le pueden hacer a una implementación (algunos ejemplos son el *momentum*, un *learning rate* adaptivo o la inyección de ruido).

Implementación

Primera versión

En una primera instancia se buscó implementar una RNA multicapa básica, sin agregados ni optimizaciones, en *Matlab*. El objetivo de esta sección es el de mostrar aspectos interesantes de esta implementación.

Uso de *structs*

Prácticamente todas las funciones tienen acceso a un *struct*, llamado `data` que cuenta con el estado actual de la RNA, incluyendo, las entradas, las salidas, los pesos, las variaciones en los pesos, los errores, los parametros, las funciones y las constantes.

Explotación de matrices y vectores

Dada la naturaleza de *Matlab*, se priorizó el uso operaciones de matrices, frente a sus análogos con loops, en pos de la performance. Por lo tanto, se modeló todo el problema de forma matricial y vectorial. Así, valores como los pesos, valores de salida y otros, se modelan como matrices o vectores. Como resultado se cuenta con un código más compacto y sencillo. Adicionalmente, los valores de los umbrales fueron incluídos en estas matrices simplificando aún más las operaciones.

Uso de *cells*

Los *cells* de *Matlab* provee una estructura de datos clara y sencilla para modelar los problemas en cuestión (varias capas con un número variable de neuronas). Por lo tanto gran parte de nuestras estructuras son *cells*.

Segunda versión

Si bien se contaba con una RNA multicapa funcional, se añadieron varias optimizaciones.

Momentum

El *momentum* simpelmente añade una fracción de la corrección previa de los pesos a la corrección actual. Este *momentum* ayuda a prevenir que la RNA se estanque en un mínimo local. Como todo parametro de una RNA, las modificaciones en el *momentum* modifican el comportamiento de la red (velocidad de convergencia, inestabilidad del sistema).

Parametros adaptativos

Se realizó una implementación que adapta los parametros usados al estado del problema. Se define un criterio basado en los errores medios para decidir si un *epoch* es considerado bueno, malo o neutral. En el caso en los que haya habido una serie de *epochs* buenos, se modifica el *learning rate* para incentivar el

aprendizaje. En el caso de un *epoch* malo se modificará en el sentido contrario. Además, el valor del *momentum* sólo es utilizado en los pasos que son considerados como buenos fomentando así una convergencia con mayor velocidad solamente cuando el problema se dirige a una solución que parece ser la correcta.

Una última optimización es la de realizar un *rollback* en los pasos que son considerados malos. Esto implica revertir la actualización de los pesos en este paso, ya que se considera que nos alejan de la solución.

Mezcla aleatoria de las entradas

De forma empírica hemos notado que se obtienen mejores resultados si al comienzo de cada ronda se mezclan los patrones de entrada. De esta manera se ha añadido esta lógica a modo de optimización.

Finalización del aprendizaje

El momento en que se decide cortar el aprendizaje es un factor clave para definir con qué efectividad puede la red generalizar el problema. Por esto, se incluyó un error que representa una cota entre las salidas esperadas y las salidas obtenidas permitiendo detener el aprendizaje una vez que se obtiene un error lo suficientemente pequeño.

Conclusiones

Pruebas previas y formas de evaluación

Uno de los problemas más famosos que son posibles de resolver con una RNA multicapa pero que no es posible de resolver con un perceptrón simple es el problema del *Xor*. Este problema y otro de complejidades similares fueron utilizados a modo de prueba de la RNA tanto en su primerq versión como en la final.

Si bien la primera implementación de nuestra red era capaz de aprender el problema, tendía a estancarse en mínimos locales. Una vez añadidos el *momentum* y la mezcla aleatoria de las entradas, el porcentaje de éxito aumentó considerablemente. Pudimos observar la alta sensibilidad de la red a los parámetros de configuración (*momentum*, *learning rate*, pesos iniciales, etc), que tuvimos que tener en cuenta una vez presentado el problema real.

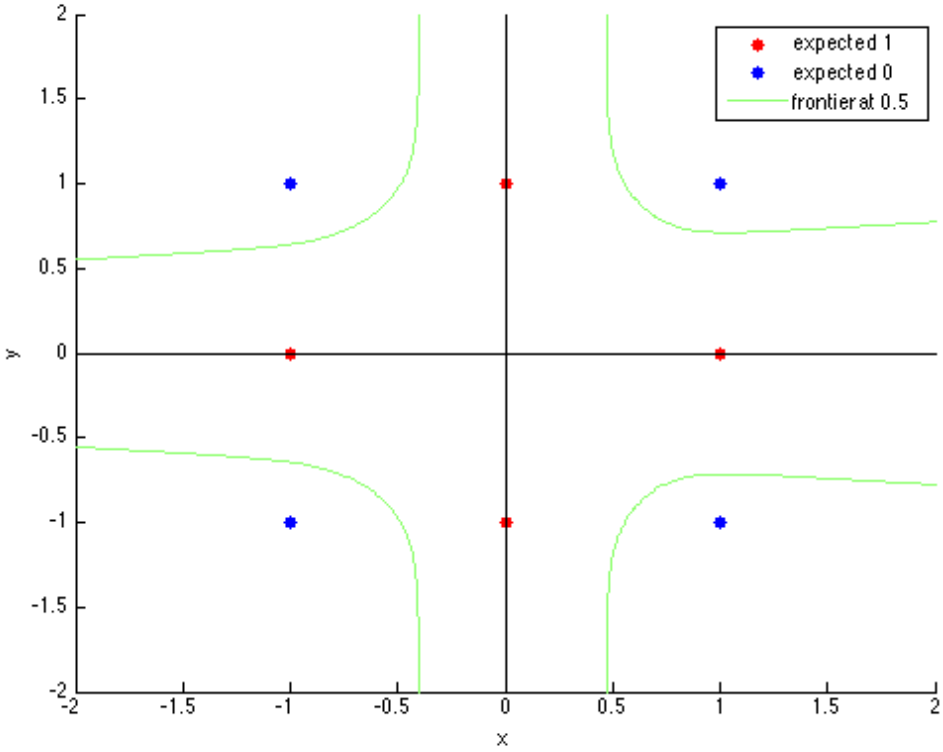
La implementación de parámetros adaptativos nos trajo respuestas mixtas. No sólo era extremadamente sensible a la definición de pasos buenos y malos (a través de un *epsilon*) sino que en la mayoría de los casos era propensa a estancarse en mínimos locales: cuando se llega a un mínimo local, cada vez que se intenta salir (hill climbing), se hace un rollback, por ser este movimiento desfavorable.

Para comprobar el funcionamiento de nuestra red se añadieron parámetros de prueba que permiten visualizar los datos de forma completa en la salida y la generación de gráficos de los pesos, el *learning rate* y los errores. A su vez, el progreso de los pesos puede ser visto gráficamente en tiempo real. Para realizar pruebas más exhaustivas se añadió la posibilidad de poder detener el aprendizaje en cualquier punto de la

ejecución, evaluar su estado y luego retomar desde ese mismo punto.

Testing

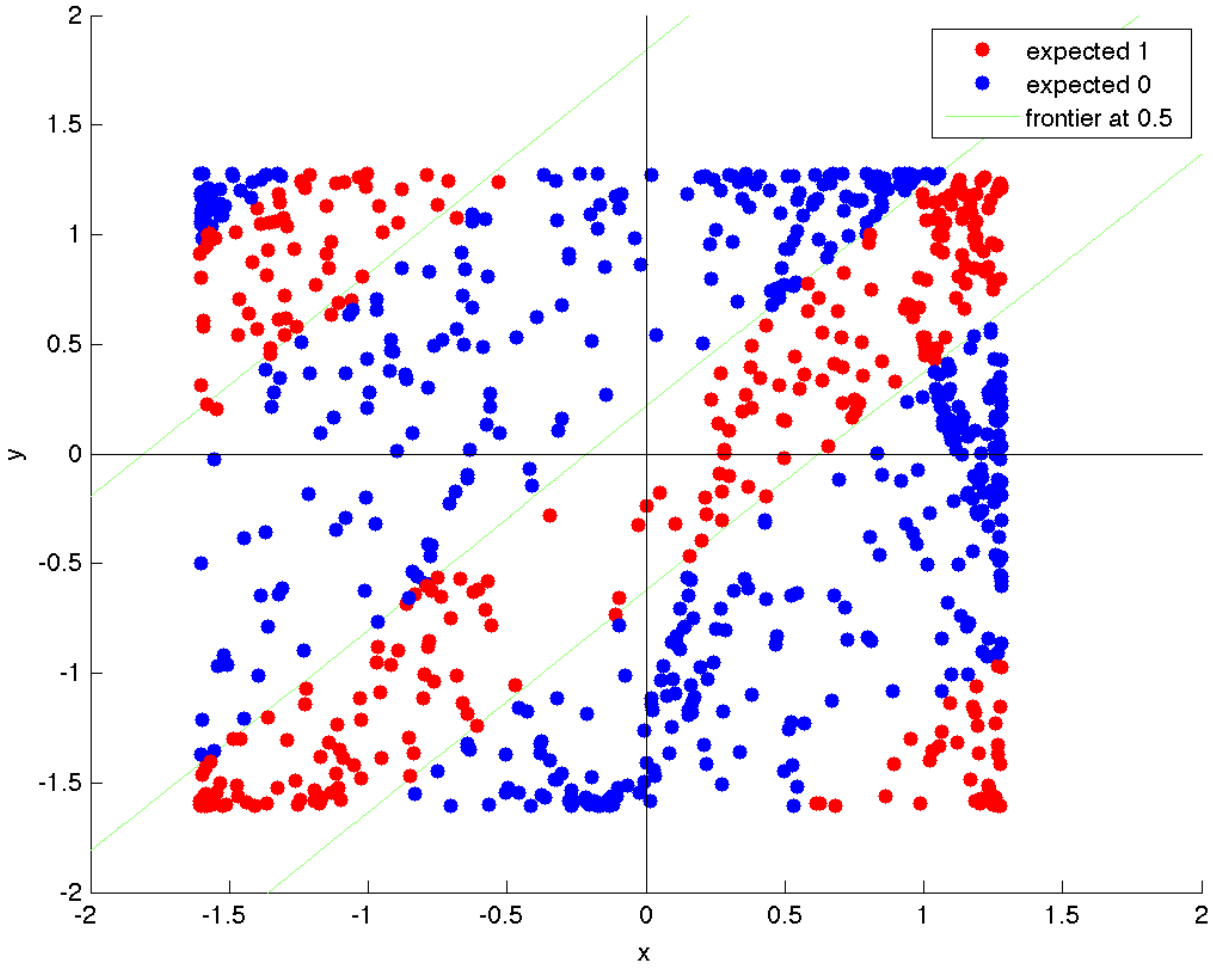
Se testeó la red con problemas que no fuesen linealmente separables, como el *xor*, y la versión más compleja que deja los unos en forma de cruz, y los ceros en las esquinas, de forma exitosa:



Para ver cómo converge la red a la solución rápidamente, se puede ver el video en youtu.be/0LSXqYxPMjw.

Conclusiones relacionadas al problema

Dados los parametros de *debug* y configurando la función para ser dependiente de dos valores previos se pudo ver rápidamente su distribución. A su vez, se pueden ver los hiperplanos encontrados rápidamente por la RNA que trata de resolver el problema.



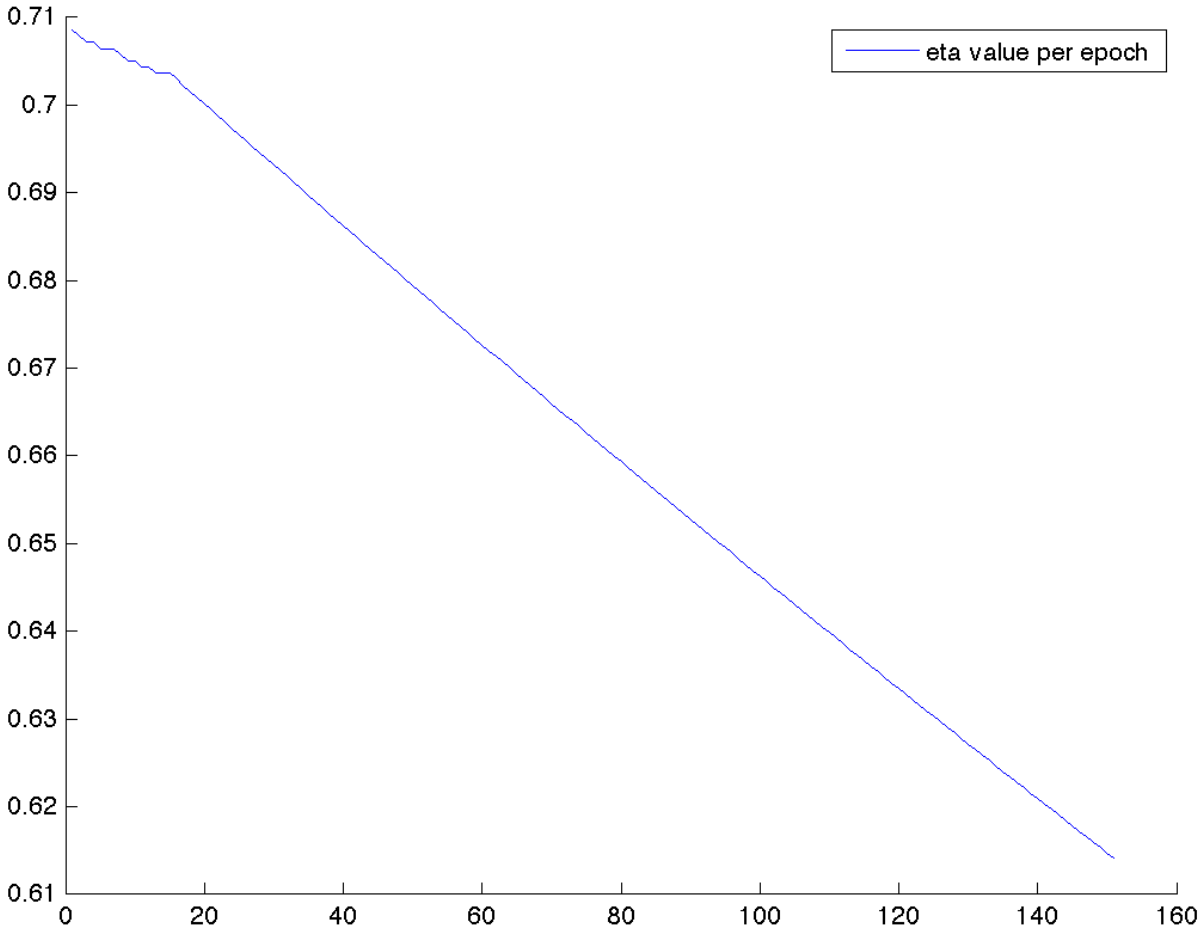
Para determinar los diferentes parámetros de nuestra RNA inicialmente corrimos el problema con sólo cien patrones de entrada que fueron tomados al azar.

Una observación interesante es que dadas las ejecuciones nunca se pudo obtener un hiperplano que separe a los puntos del borde inferior derecho del gráfico, dado la poco frecuencia allí presente. En un futuro se podría agregar un castigo a la función de error por entradas valuadas muy distinto a su valor real.

Al ejecutar la red con distintos parametros pudimos obtener conclusiones sobre la configuración óptima de cada parámetro.

Parametros adaptativos

Al entrenar la RNA con el *momentum* adaptativo y *rollbacks* se llegó a la misma conclusión obtenida en los problemas evaluados previamente. Una vez que la red llega a un mínimo local, nunca sale de él y se queda haciendo *rollback* indefinidamente.



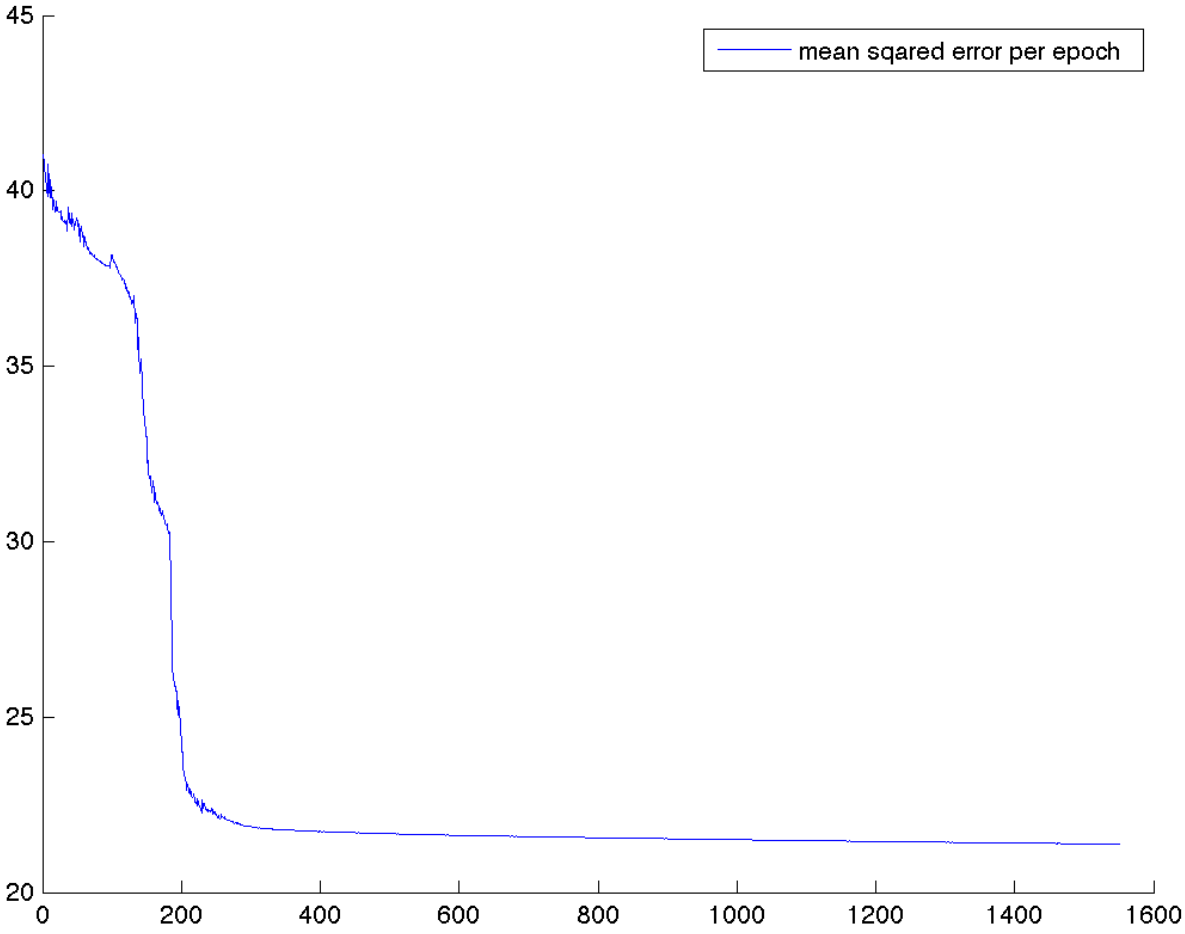
Paralelamente, definir un *learning rate* no perturba al comportamiento de l RNA. Sin embargo, su uso debe ser cauteloso ya que definir sus configuraciones de manera azarosa hace que el *learning rate* disminuya considerablemente haciendo que la red se estanque. Estos parámetros son:

- Epsilon para definir el error
- Incremento en *learning rate*
- Decremento en *learning rate*
- Cantidad de buenos pasos para incrementar el *learning rate*

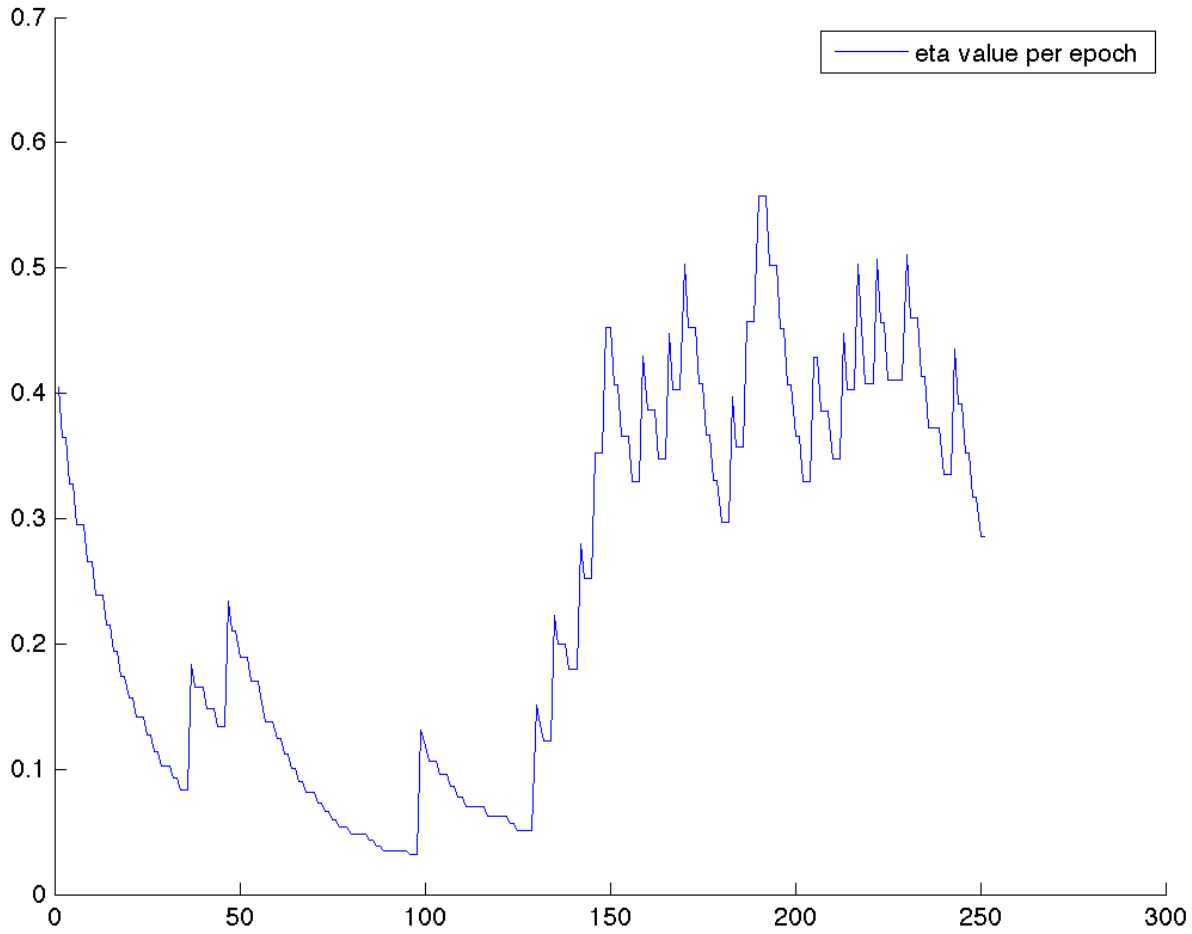
Los siguientes valores probaron ser útiles al entrenar la red:

- *Learning rate* inicial = 0.7
- Epsilon = 0.001
- Incremento = 0.01
- Decremento = 0.001
- Cantidad de pasos buenos = 4

En el siguiente gráfico se pueden ver los errores cuadráticos medios para esta configuración.



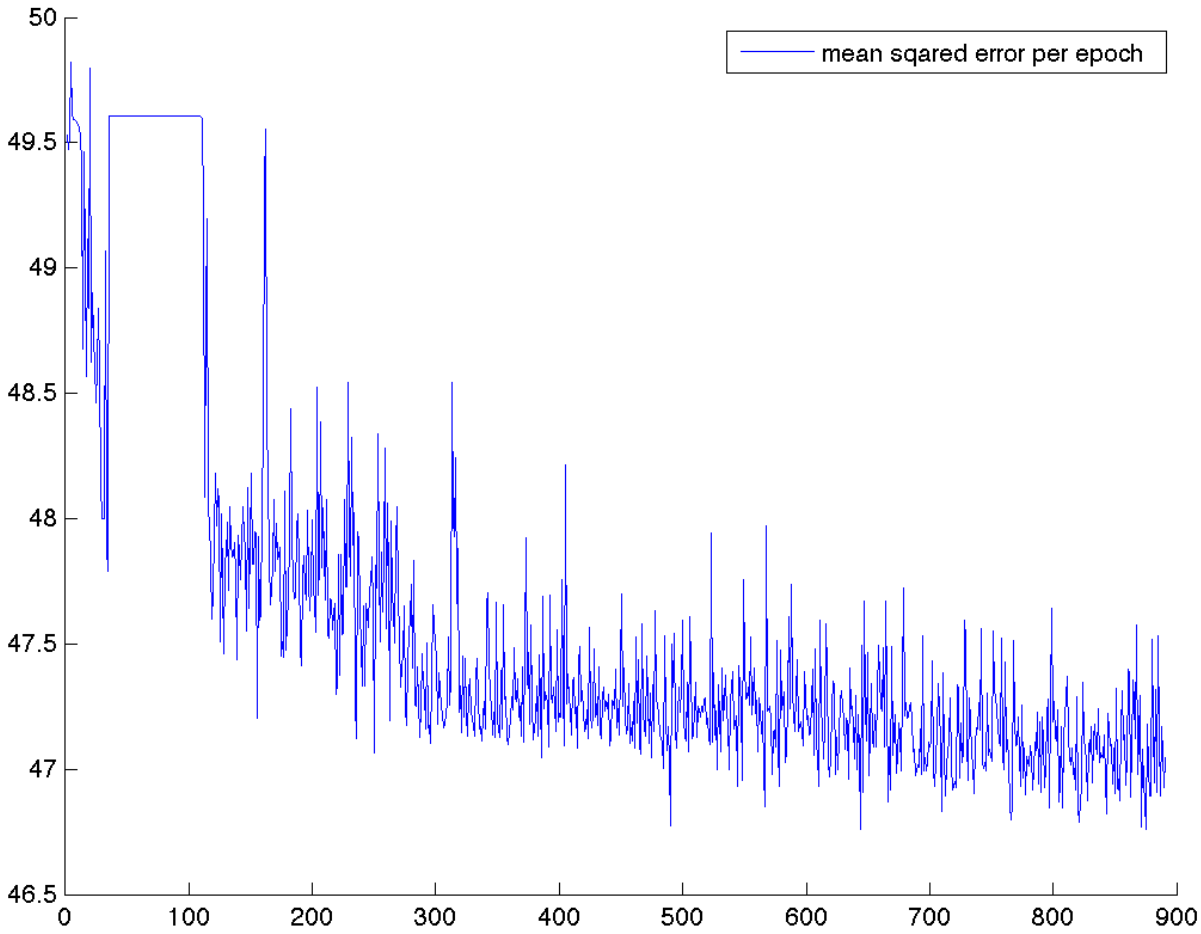
El siguiente gráfico muestra la variación del *learning rate* para esta configuración.



Momentum

Tras diversas pruebas empíricas hemos encontrado que los mejores resultados para la red en este problema se obtienen con un *momentum* de 0.5. Aquellos valores cercanos al 0 o al 1 producen que la red se estanque.

El siguiente gráfico muestra el caso con el *momentum* definido en 0.9 y es fácil ver su estancamiento a lo largo de las iteraciones. También se puede observar que al ser un *momentum* cercano a 1 hay mucho ruido.



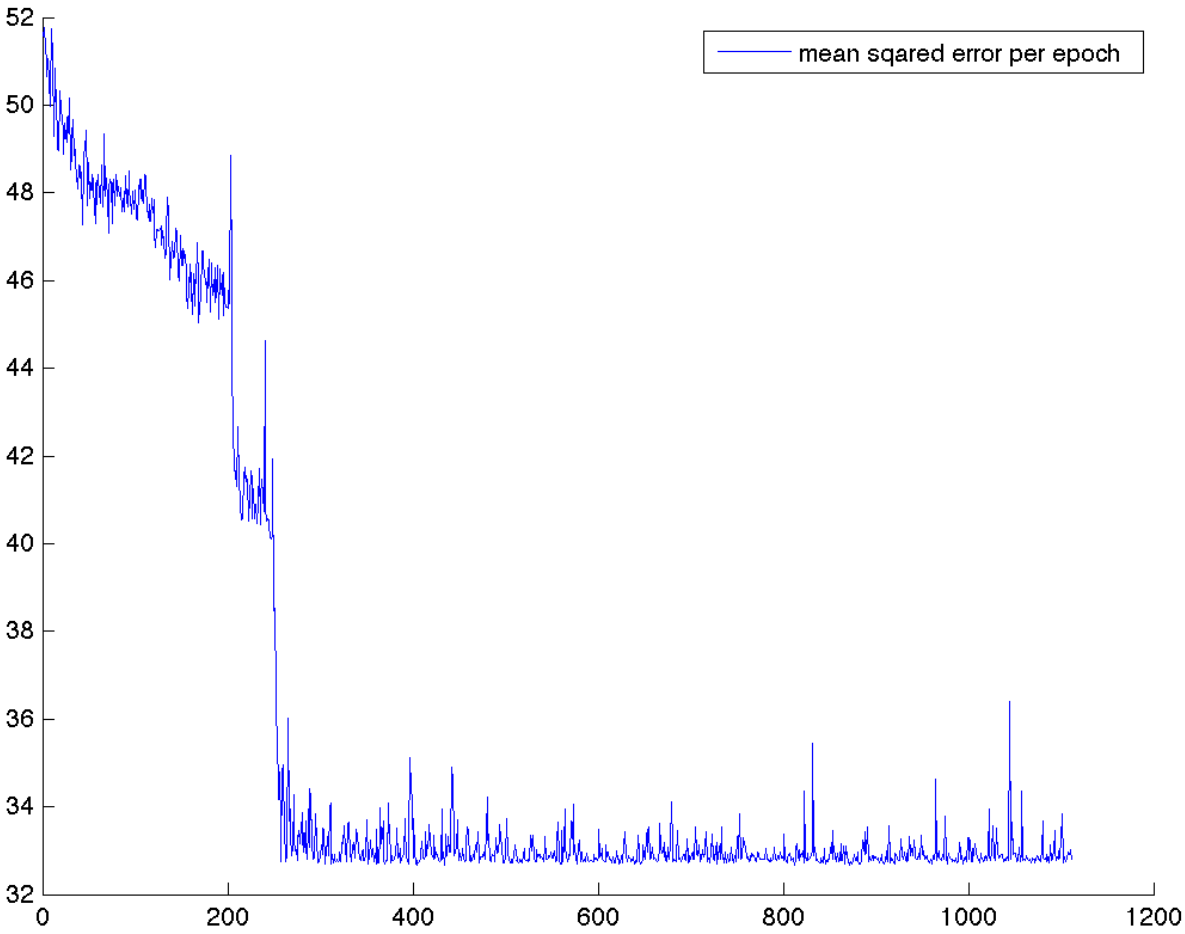
Beta y funciones de transferencia

Dado que normalizamos la función tangente hiperbólica para que su imagen sea `(0,1)`, esta función y la función sigmoidea lógica quedan equivalentes.

Así, basta con variar el parámetro *beta* para ajustar nuestra función de transferencia.

Cuando *beta* es cercano a 1 la función sigmoidea se acerca a la función escalón y por lo tanto su derivada diverge. Esto hace que el gradiente sea muy grande en valor absoluto y por lo tanto los pesos tengan mucho ruido.

A continuación se presenta un gráfico del error por cuadrados mínimos con *beta* = 1



El valor de *beta* elegido fue de 0.5

Estructuras de la RNA

La RNA fue probada con las siguientes arquitecturas:

- [2 5 1]
- [2 6 1]
- [2 6 2 1]
- [2 7 1]
- [2 8 1]
- [3 6 1]
- [3 7 1]
- [3 8 1]
- [2 9 6 1]
- [3 6 2 1]

Al ver la distribución de los datos modelada en el plano, sugerimos a modo de apuesta la solución [2 5 1] ya que notamos que era posible separar las distintas regiones del problema con 5 rectas.

Luego, dado que toda una región del plano estaba cubierta por puntos pensamos que agregar una entrada más (3 neuronas en la capa de entrada) resolvería mejor el problema disminuyendo el error. Sin embargo, al probarlo empíricamente encontramos que la *performance* de esta arquitectura era similar (o incluso peor) que a la de 2 entradas.

A modo de prueba se incluyeron más neuronas en la capa oculta sin encontrar mejoras considerables.

Una conlusión interesante se presentó al tratar de incluir una segunda capa oculta. Como se puede ver en la imagen a continuación, la capa de salida le asignó el mismo peso a ambas neuronas de la segunda capa oculta. Esto nos llevó a la hipotesis de que la segunda capa era redundante.

```
.alg.W{2} =

                --1-->      --2-->      --3-->
      --1-->    20.15110    -8.56963     8.03272
      --2-->     8.65498    -3.65877     4.34386
      --3-->   -14.65781     8.59805    -8.64390
      --4-->     7.65229     7.92913    -7.81909
      --5-->    -1.84570     8.79047    -8.84514
      --6-->    -2.49569   -10.23756    10.24568

.alg.W{3} =

                --1-->      --2-->      --3-->      --4-->      --5-->
      --1-->     0.26983    -2.89921    -3.52365    9.93706     5.30987
      --2-->    -1.01022     6.63504     4.20235    9.69620     6.01835

                --6-->      --7-->
      --1-->   -11.75011     0.42154
      --2-->   -10.56229    -9.16443

.alg.W{4} =

                --1-->      --2-->      --3-->
      --1-->    -9.39846    -9.26422   -13.43572
```

Ejecución óptima

Con los parametros óptimos definidos en la sección previa y con la arquitectura [2 7 1] se obtuvieron los siguientes errores:

Mean Square Error: 307.07 Media entre las normas de las diferencias entre las salidas esperadas y las obtenidas: 0.54

Por último se presentan en el gráfico los pesos finales de las conexiones.

```
alg.W{2} =

                --1-->       --2-->       --3-->
      --1-->      9.28768     10.63877    -10.55919
      --2-->      0.60014     10.91835    -11.05126
      --3-->    -10.72115      6.58050     -6.59290
      --4-->    -19.58779      7.50959     -7.36526
      --5-->      4.92149     12.68807    -12.62336
      --6-->     -0.60000      1.35388     -6.98152
      --7-->     -4.39225     10.25358    -10.27675

alg.W{3} =

                --1-->       --2-->       --3-->       --4-->       --5-->
      --1-->      6.19399    -15.26774     11.01746    -16.53541     10.93564

                --6-->       --7-->       --8-->
      --1-->    -10.77169     -0.32678     12.37113
```