Universidad de Buenos Aires
Facultad de
Ciencias Exactas y Naturales
Departamento de Computación

Redes Neuronales

Primer Cuatrimestre de 2014

Trabajo práctico

Preceptrón simple
Preceptrón multicapa

1ra entrega

Entrega del punto 1:

Fecha de entrega: 24 de Abril

Integrante	LU	Correo electrónico
Cammi, Martín	676/02	martincammi@gmail.com
De Sousa Bispo, Mariano	389/08	marian_sabianaa@hotmail.com

Ejercicio 1: OCR

Arquitectura de la red

Introducción

Para el ejercicio del perceptrón simple de OCR la red neuronal consistirá de 25 nodos de entrada que modelarán las letras del abecedario representadas en las 5 filas y 5 columnas. La cantidad de nodos de la salida a su vez será de 5 nodos que representarán los 5 bits de codificación binaria de cada letra.

Los valores de entrada serán o bien 0 o bien 1. Para los valores de salida si bien se esperará que también sean valores binarios, los resultados obtenidos por el perceptrón serán de números flotantes que intentarán acercarse a estos valores.

Implementación

Para la implementación del Perceptrón Simple se decidió utilizar Python. La clase *SimplePerceptron* posee el código del algoritmo.

El main se encarga de levantar un archivo y generar los parámetros (colaborando con el *FileParser*) para el *SimplePerceptron*, en cada época se encarga de guardar la información de error y de validación.

También se generó un programa que genera inputs que se llama *OCRInputCreator* el cual permite definir letras con ruido, así como modificar el output de cada salida.

La clase *Plotter* es la encargada de, dada la información recopilada guardar y/o mostrar los gráficos de la ejecución.

Conjuntos de entrenamiento

A continuación se presentan varios tests modificando los parámetros de coeficiente de aprendizaje (etta), la cota del error (epsilon), la cantidad de épocas (época) y la función de activación (Exponencial, Signo, Identidad).

En todos los casos la matriz de pesos inicial que representa la red neuronal es inicializada con valores aleatorios entre -0.1 y 0.1

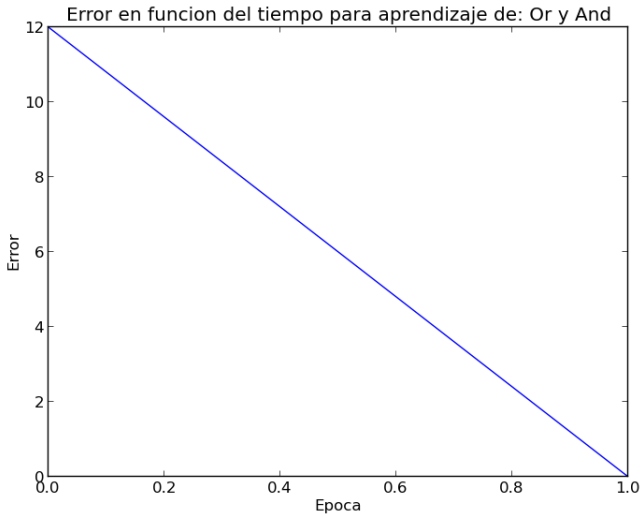
Funciones utilizadas

Signo: Para valores superiores o iguales a 0 devuelve 1, para inferiores a 0 devuelve -1.
Exponencial: Aplica la fórmula 1/(1 + exp(-2 * 0.5 * value))

AND / OR

Test #1

- Función = And y Or
- Etta = 0.15
- Epsilon = 0.1
- Épocas = 50
- Función de activación: Signo

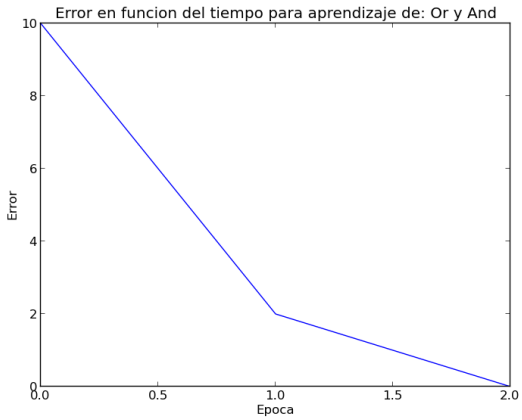


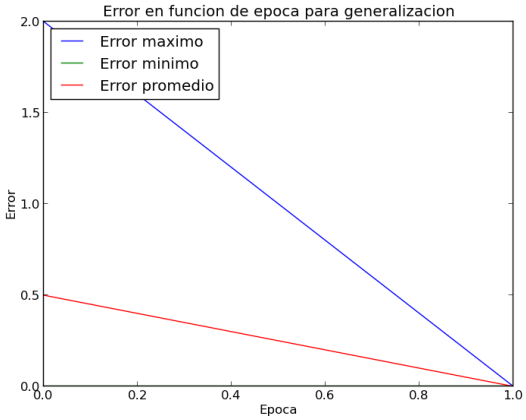
Stop condition: Epsilon bound met on epoch: 2 with epsilon value: 0.0

Explicación: Utilizando la función de activación signo y reemplazando los ceros de input y output de las funciones AND y OR por -1 se logró que la red neuronal del perceptrón simple aprendiera completamente ambas funciones en sólo 2 iteraciones.

Los parámetros iniciales fueron arbitrarios ya que las 50 épocas no fueron necesarias y el epsilon de error fue cumplido satisfactoriamente.

Se realizaron también otras corridas ya que la matriz inicial comienza con valores arbitrarios y en todos ellos se obtuvieron resultados que aproximaron exitosamente.





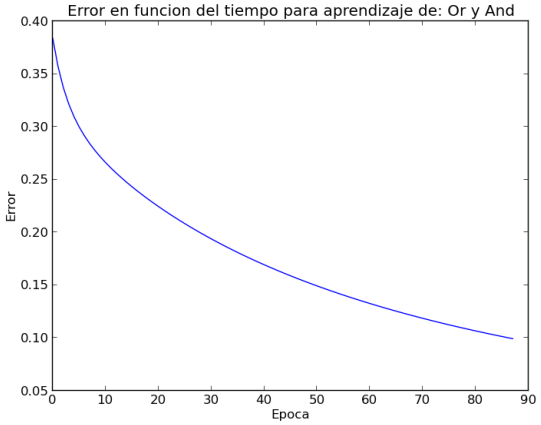
Stop condition: Epsilon bound met on epoch: 3 with epsilon value: 0.0

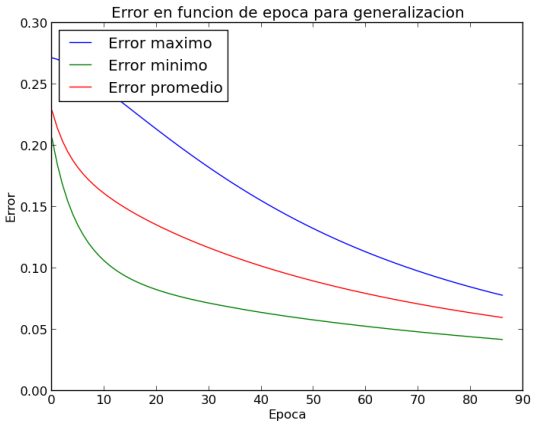
Explicación: En el diagrama de la izquierda figura el error en función de la cantidad de iteraciones se pude ver que en este caso con una matriz de pesos inicial diferente la convergencia empeora levemente. En el diagrama de la derecha se muestran el error máximo, mínimo (que en verde se superpone con el eje x) y el promedio.

Habiendo aproximado exitosamente las funciones de AND y OR, se decidió de todos modos probar on una función de activación sigmoidea para realizar comparaciones. Se utilizó la función exponencial para tal caso.

Test #2

- Función = And y Or
- Etta = 0.15
- Epsilon = 0.1
- Épocas = 100
- Función de activación: Exponencial





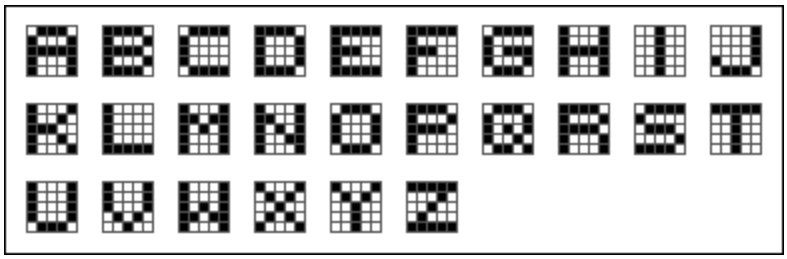
Stop condition: Epsilon bound met on epoch: 88 with epsilon value: 0.09936227

Explicación: En el diagrama de la izquierda donde figura el error puede apreciarse como este requiere de muchas más iteraciones para lograr cumplir el error requerido, velocidad que también se evidencia en el diagrama de la derecha de máximos, mínimos y promedio.

Conclusión: Parece ser que utilizando una función de activación discreta como es Signo aplicada a un problema con valores binarios es mucho más eficiente ya que ajusta los valores para que también sean binarios. En el caso de la exponencial, si bien termina aproximando al error exigido, requiere de muchas más épocas ya que posiblemente sus valores sean continuos.

OCR

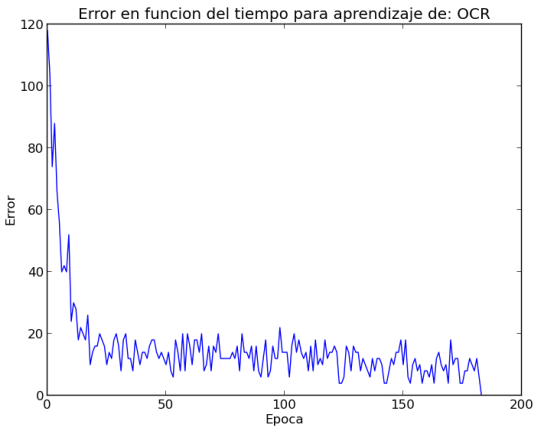
A continuación se muestran los tests correspondientes al aprendizaje de la red neuronal de Perceptrón Simple para el alfabeto de 26 letras codificadas cada una en una matriz binaria de 5 x 5.



Curvas de aprendizaje

Test #3

- Función = OCR
- Etta = 0.15
- Epsilon = 0.1
- Épocas = **1500**
- Función de activación: **Signo**

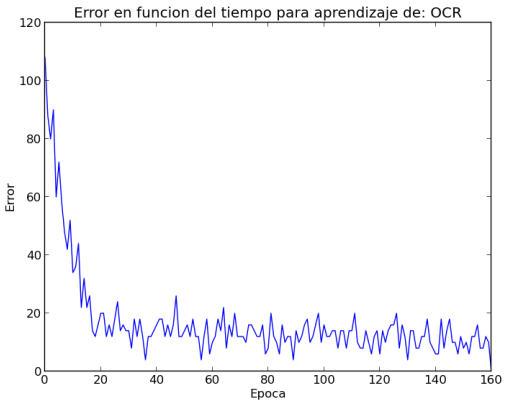


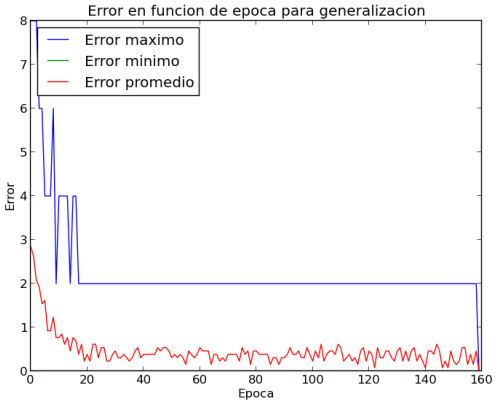
Stop condition: Epsilon bound met on epoch: 184 with epsilon value: 0.0

Explicación: En el diagrama de error se ve inicialmente un abrupto descenso del error pero que luego va fluctuando hasta llegar al aceptado. En el diagrama de errores por época se puede apreciar una marcada cota superior mientras que el promedio del error va descendiendo hasta el aceptado.

Test #4

- Función = OCR
- Etta = **0.85**
- Epsilon = 0.1
- Épocas = 1500
- Función de activación: Signo



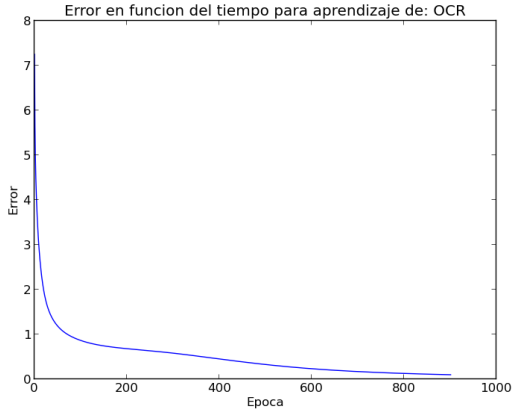


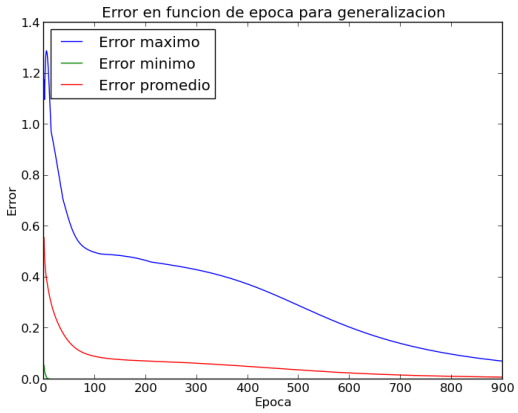
Stop condition: Epsilon bound met on epoch: 161 with epsilon value: 0.0

Explicación: En este caso se ha aumentado el valor del coeficiente de aprendizaje etta llevándolo a **0.85.** No se ha notado demasiado cambio salvo que se pudo llegar al error aceptable en menos epocas (23 épocas menos). También se puede notar la alternancia de error en la función esto se suele suavizar al usar una función de activación exponencial utilizando un etta mayor pero en este caso aún aumentando el etta la alternancia permanece, parece ser en este caso intrínseca a la función de activación utilizada y el problema en particular.

Test #5

- Función = OCR
- Etta = **0.45**
- Epsilon = 0.1
- Épocas = 1500
- Función de activación: **Exponencial**



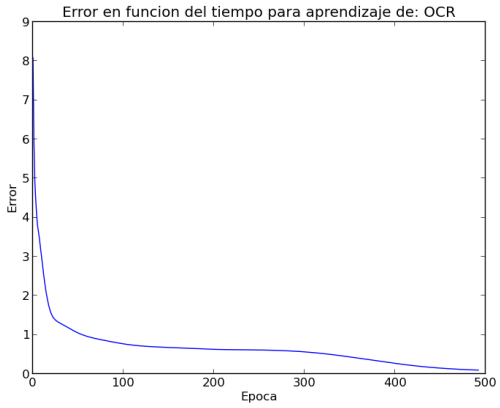


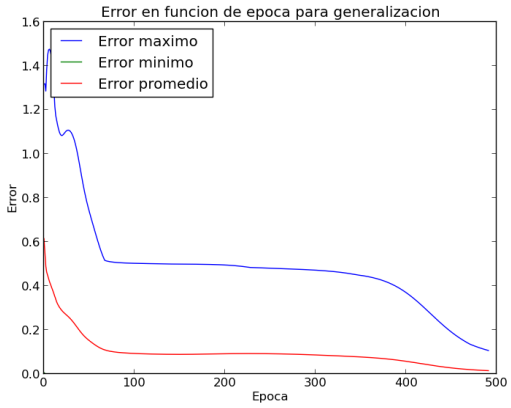
Stop condition: Epsilon bound met on epoch: 902 with epsilon value: 0.09978

Explicación: En este caso se ha utilizado una función de activación sigmoidea, la exponencial. Se ha optado por un etta medio de **0.45** con el cual se ha logrado alcanzar el error deseado en la época 902. Bastantes más épocas que con la función de activación Signo del Test #4 anteriormente descripto. Parece ser entonces que la función Signo es mucho mejor en este caso al momento de hacer a la red aprender. Si bien en el Test #4 se aprecia leves divergencias del error en la curva, que finalmente termina convergiendo, lo logra de una forma más rápida que utilizando la función exponencial.

Test #6

- Función = OCR
- Etta = **0.85**
- Epsilon = 0.1
- Épocas = 1500
- Función de activación: Exponencial



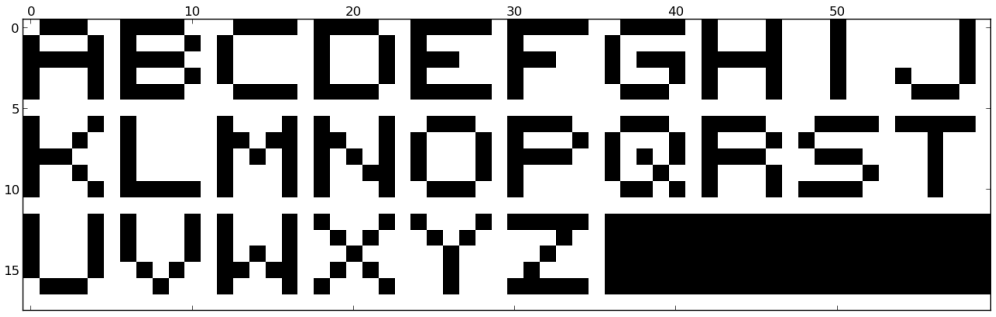


Stop condition: Epsilon bound met on epoch: 493 with epsilon value: 0.09948

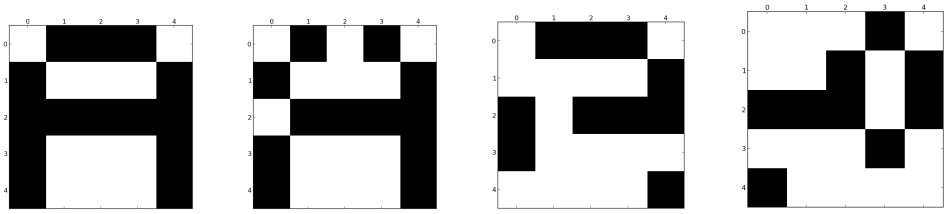
Explicación: Aumentando un poco el etta se ha conseguido alcanzar el error requerido en sólo 493 épocas.

Agregado de Ruido

Una vez la red neuronal ha aprendido todas las letras del alfabeto, es interesante ver que tan capaz es de reconocer una letra si se le ha agregado ruido a la misma previamente.



A la lista de letras con que se cuenta se le ha agregado ruido para los sucesivos tests. Para establecer el ruido se define una función aleatoria que por cada bit de la representación matricial de la letra decide con una cierta probabilidad parametrizable si el bit será alterado o no. De este modo, una letra A se verá inicialmente limpia y levemente modificada con un 10% de ruido y mucho más con un 50% de ruido.

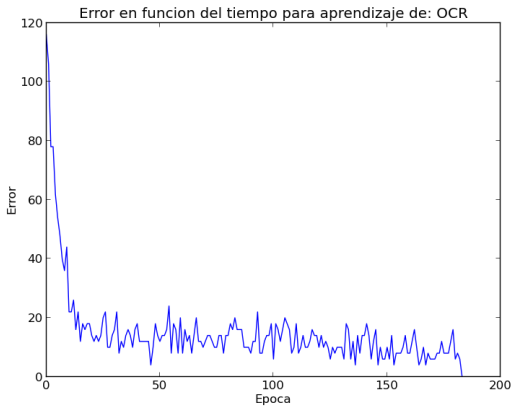


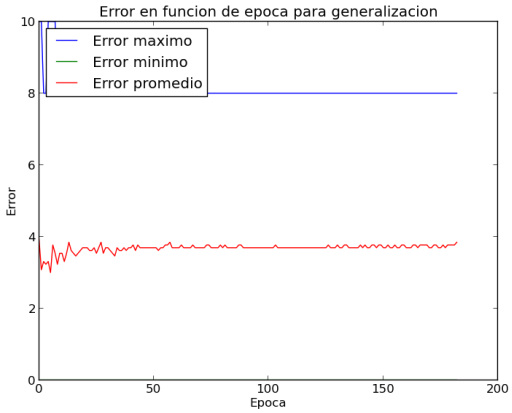
A limpia | 10% de ruido | 30% de ruido | 50% de ruido

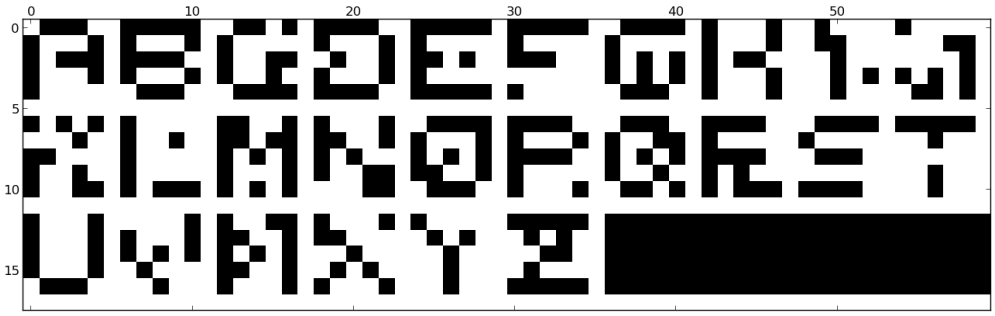
Tests con ruido

Test #7

- Función = OCR
- Etta = 0.85
- Epsilon = 0.1
- Épocas = 1500
- Función de activación: Signo
- Nivel de ruido: **10%**







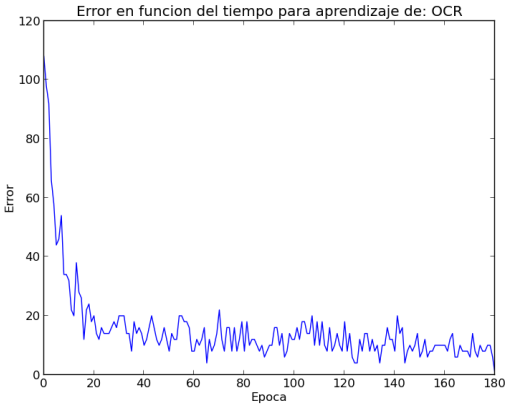
Nivel de ruido de 10%

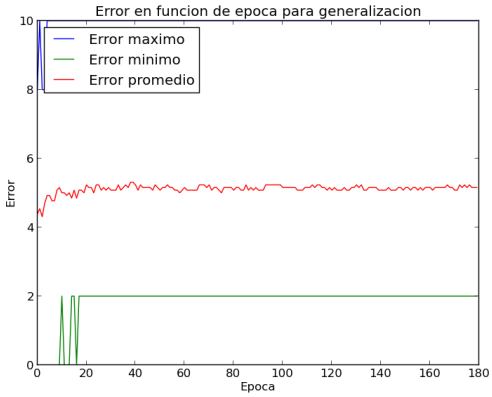
Stop condition: Epsilon bound met on epoch: 184 with epsilon value: 0.0

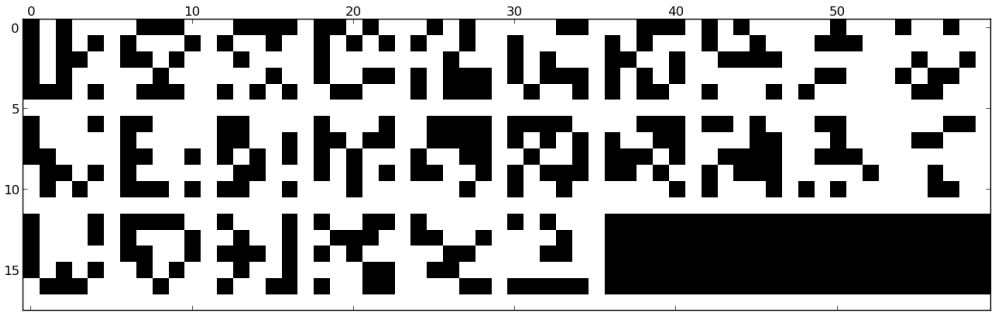
Explicación: Se utilizaron los mismos parámetros del Test #5 por haber sido el que aprendió en la menor cantidad de épocas. En este caso se agregó un 10% de nivel de ruido a las letras que se pueden visualizar arriba. Existe una similitud con el Test #4 en lo que respecta a la convergencia caótica del error y a una cota máxima para lo que parece ser alguna letra en particular que no logra aprender sino hasta el final en que logra finalmente acotarla.

Test #8

- Función = OCR
- Etta = 0.85
- Epsilon = 0.1
- Épocas = 1500
- Función de activación: Signo
- Nivel de ruido: **30%**







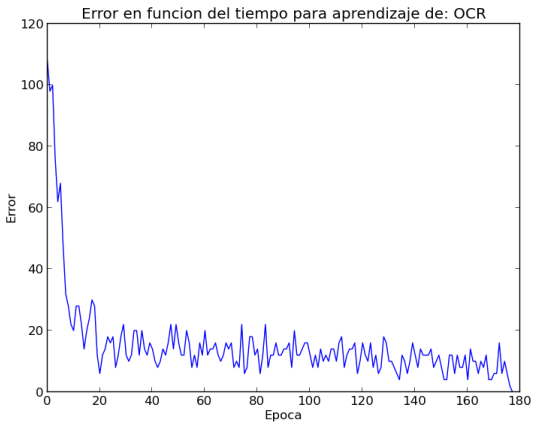
Nivel de ruido de 30%

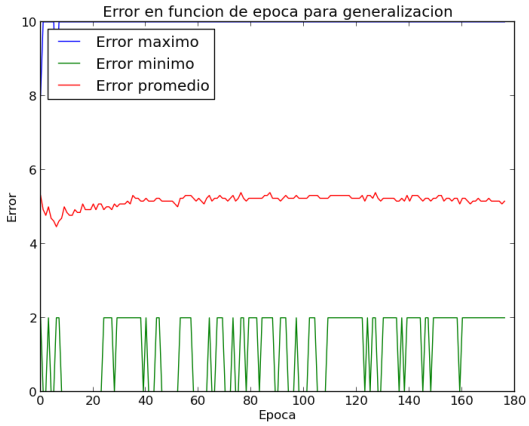
Stop condition: Epsilon bound met on epoch: 181 with epsilon value: 0.0

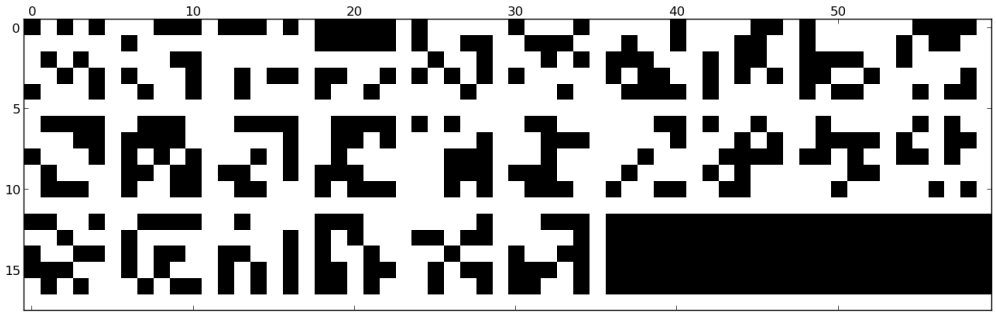
Explicación: En este caso se observa que la fluctuación del error es similar al Test #7 pero sucede algo extraño y es que aunque el error máximo de alguna letra se dispare y el promedio de los errores sea algo más alto parece aprender en inclusive menos épocas (en 3 épocas menos) con el nivel indicado de error. Un cambio significativo es que mientras que el mínimo caso en el Test #7 tenía un error de 0.0 (línea verde) en este caso el mínimo se ha disparado hasta un error 2 que no disminuye sino hasta terminar el aprendizaje.

Test #9

- Función = OCR
- Etta = 0.85
- Epsilon = 0.1
- Épocas = 1500
- Función de activación: Signo
- Nivel de ruido: **50%**







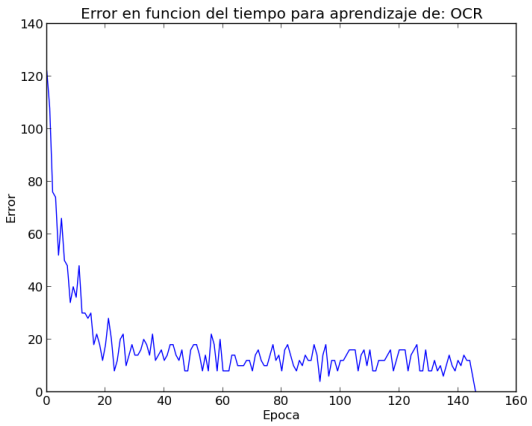
Nivel de ruido de 50%

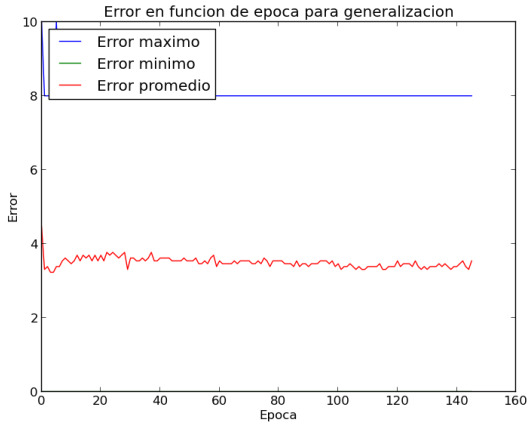
Stop condition: Epsilon bound met on epoch: 178 with epsilon value: 0.0

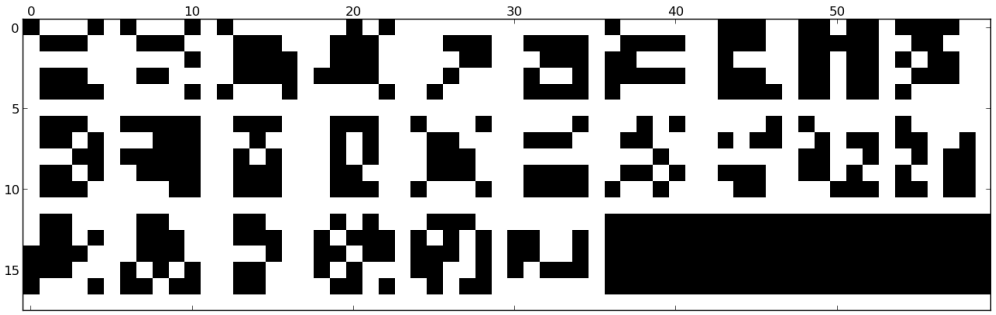
Explicación: La fluctuación observada en este caso se mantiene y el error máximo también pero en el promedio se comporta caóticamente alternando entre disminuciones y aumentos del error aunque mantienen la cota superior de 2 a medida que transcurren las épocas.

Test #10

- Función = OCR
- Etta = 0.85
- Epsilon = 0.1
- Épocas = 1500
- Función de activación: Signo
- Nivel de ruido: **90%**







Nivel de ruido de 90%

Stop condition: Epsilon bound met on epoch: 147 with epsilon value: 0.0

Explicación: Ahora en este nuevo test se observan semejanzas al Test #7 donde había un 10% de ruido. Una posible explicación es que al observar el gráfico de las letras con el ruido del 90% ¡podemos llegar a identificarlas nuevamente! esto sucede porque recién ahora reconocemos que el ruido por como se ha definido es simétrico. Siendo que cada bit se altera con cierta probabilidad si todos ellos son alterados quedará dibujada exactamente la misma letra pero en negativo. Quizás la definición de ruido debió haber sido solamente de agregar bits “1” con una probabilidad creciente. De la manera definida el ruido cuanto mayor es termina

invirtiendo la letra es por eso que puede apreciarse casi por completo en el último gráfico visualizándose en color blanco en vez del original negro.

Es por eso que en los gráficos de ruido el que peor aproximó fue el que se encontraba cerca el punto medio de ruido (viendo las mediciones es el de 30% de ruido, aunque el de 50% se le asemeja). Una conclusión posible de que es que la red identifique el patrón de la letra (ahora en blanco en vez de negro) y pueda deducir de allí la misma estructura que en las letras originales; esto explicaría el porqué logró adaptarse tan bien a letras con ruido de 10% así como de 90% tan bien dada la característica simétrica del ruido aplicado.

Ejercicio 2: Aproximación de funciones

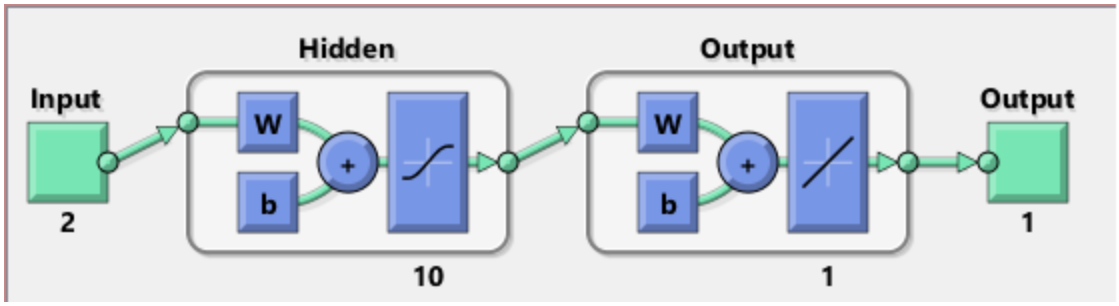
Arquitectura de la red

Introducción

Para el ejercicio del perceptrón multicapa para aproximar la función seno la red neuronal consiste de 2 unidades de entrada que modelarán las variables *X* e *Y* y una tercera correspondiente al bias. Tendremos una unidad de salida la cual será el resultado de aplicar la función dada por el enunciado a *X* e *Y*. Utilizaremos una capa oculta de 10 unidades con función de activación sigmoidea, de todas formas mostraremos algunos gráficos de aprendizaje con distinta cantidad de unidades en la capa oculta.

Para este análisis utilizamos Neural Network Tools de MatLab. Se utilizó Python para generar el código para correr en MatLab.

A continuación que muestra una imagen obtenida de MatLab con la arquitectura de la red



Justificación

La arquitectura fue definida empíricamente.

Conjuntos de entrenamiento

Se generaron diversos conjuntos de entrenamiento los cuales variaron entre 25, 50 y 100 vectores de aprendizaje. Se creó un subset de puntos mínimos que deben estar en el input y el resto se generaron de manera aleatoria para que la suma de X+Y/2 se encuentre entre 0 y 2 pi.

Los puntos que todo set de aprendizaje posee son:

- (0,0)
- (pi,2pi)
- (pi/2,0)
- (0,pi)
- (pi/2,pi)
- (0,pi/2)
- (pi/2, pi/2)
- (pi/4,2pi)
- (pi,pi)
- (5pi/4,pi)

MatLab toma un subconjunto del vector de entrada y en cada época testea contra él. Si bien no conocemos qué vectores tomó, en los gráficos se muestran los errores de validación en cada iteración.

A su vez, se generaron sets de testeo de entre 100 y 50000 entradas de manera aleatoria para analizar tanto la interpolación como la extrapolación.

Utilizando la función *perform* que recibe una red, un vector de inputs y un vector de outputs pudimos hacer una prueba extra para ver qué tan bien generaliza en cada caso.

Se utilizó como parámetro *epsilon* al 0.01 ya que consideramos que genera un balance interesante entre el error bajo y tiempo computacional. Cabe notar que a modo de prueba probamos con *epsilon* igual a 0,001 y no alcanzaron 3000 épocas para que pueda la red obtener error menor a ese valor.

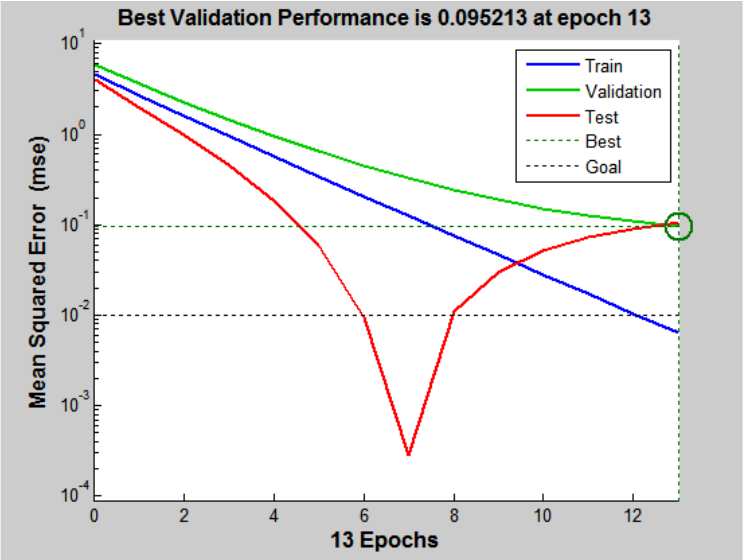
MatLab permite parametrizar más variables, entre ellas el parámetro *Validation Check*, este parámetro estipula que si en una época se incrementa el error con respecto a la época anterior, esto sólo puede ocurrir la cantidad de veces parametrizadas. Si se produce que en en el aprendizaje se supera esta cota, el algoritmo termina.

Curvas de aprendizaje y performance

Algoritmo Gradient Descent Backpropagation

Test #1

Set de entrenamiento: 100
Set de testeo (función *perform)*: 100
Épocas: 3000
Unidades en capa oculta: 10
Check de validación: 100
Etta: 0.07

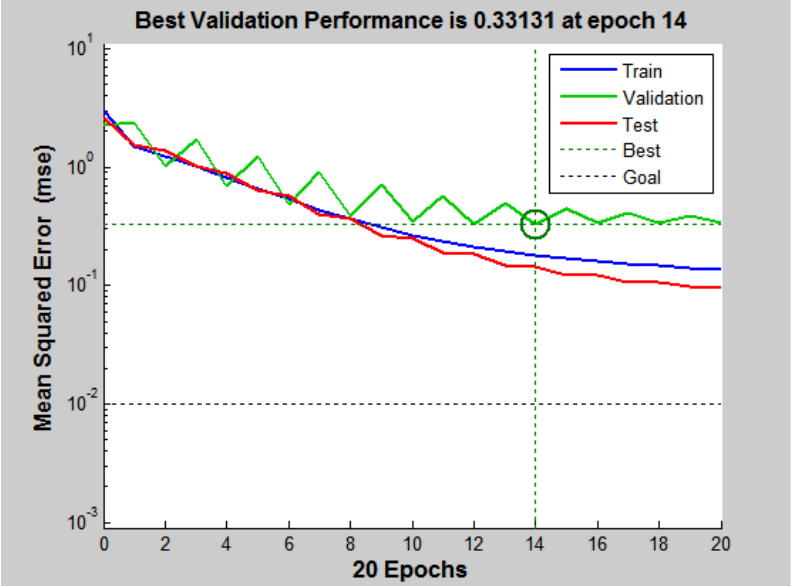


Resultados:

- Épocas insumidas: 13
- Epsilon: <0.01
- Resultado función *perform*:0.2788

Test #2

Set de entrenamiento: 100
Set de testeo (función *perform)*: 100
Épocas: 3000
Unidades en capa oculta: 10
Check de validación: 100
Etta: 0.17

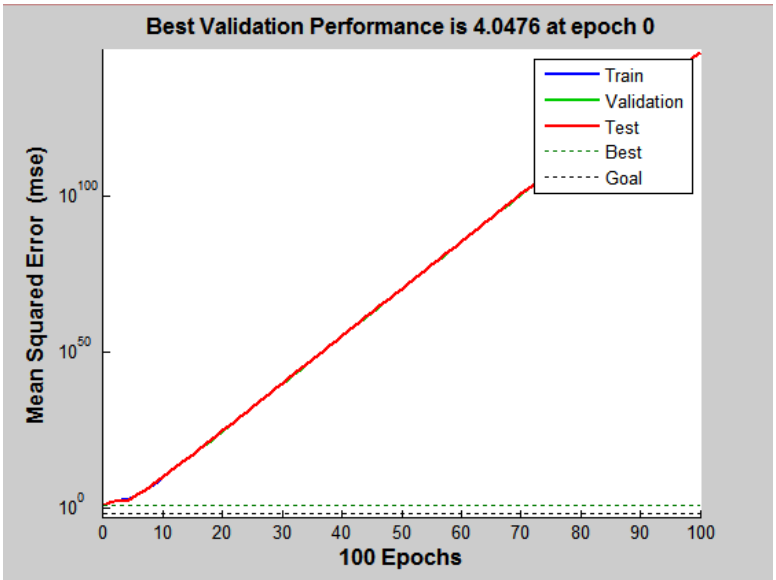


Resultados:

- Épocas insumidas: 20
- Epsilon: <0.01
- Resultado función *perform*:0.5311

Test #3

Set de entrenamiento: 100
Set de testeo (función *perform)*: 100
Épocas: 3000
Unidades en capa oculta: 10
Check de validación: 100
Etta: 0.35



Resultados:

- Épocas insumidas: 100, terminando por cota de checks de validación
- Epsilon: >>0.01
- Resultado función *perform:*4.9066

Conclusiones:

Dependiendo de la matriz inicial, la cantidad de épocas para que el algoritmo encuentre solución puede aumentar o disminuir, por eso suponemos que el caso de etta igual a 0.07 insumió menos épocas que el caso con etta igual a 0.17. Creemos que en el caso del test #1 la fluctuación del error de test se debió a la exploración ya que el error de entrenamiento sigue siendo alto.

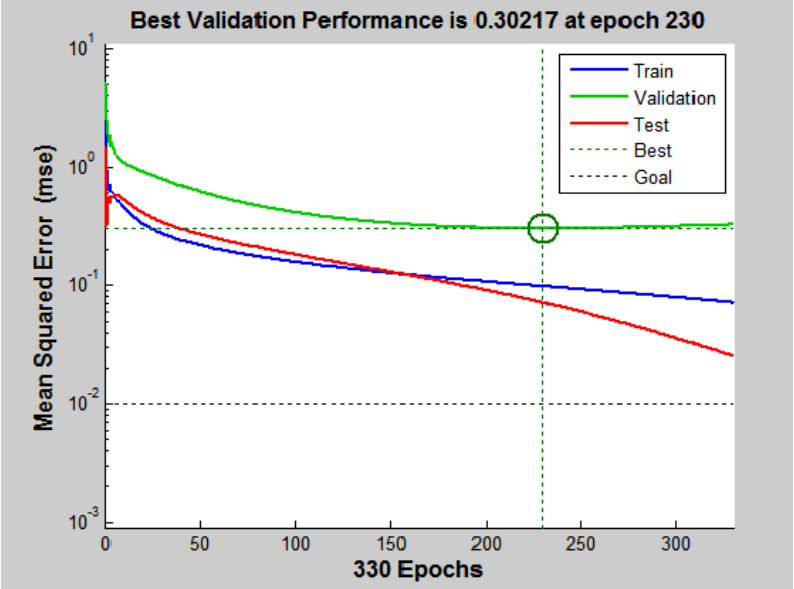
Sin embargo el parámetro etta es crucial a la hora de iniciar el aprendizaje. El aumento de etta puede llevar a que la función de error tome forma de zig zag (ver test #2) ya que al representar la velocidad, en cada iteración se pasa por exceso y debe retomar, o

simplemente puede hacer que el algoritmo no converja a la solución ya que se alejó demasiado del mínimo (ver test #3).

Sets de entrenamiento más pequeños

Test #4

Set de entrenamiento: 25
Set de testeo (función *perform)*: 100
Épocas: 3000
Unidades en capa oculta: 10
Check de validación: 100
Etta: 0.07

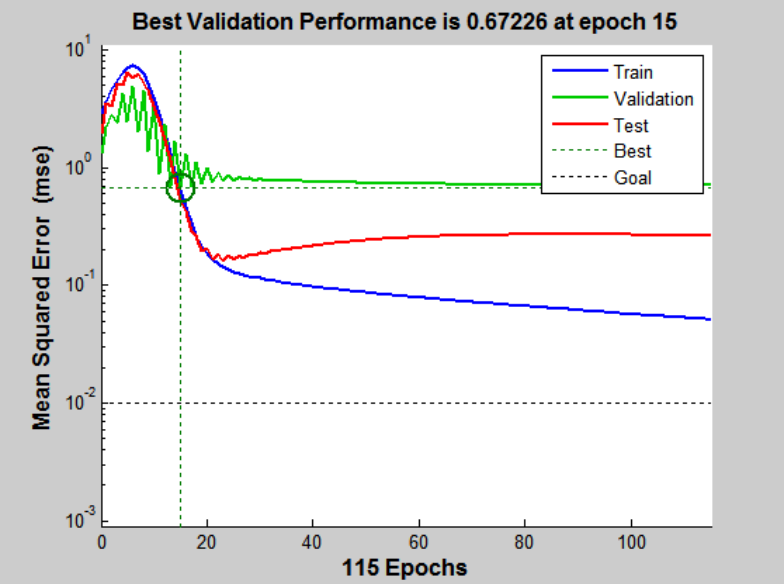


Resultados:

- Épocas insumidas: 330, terminando por cota de checks de validación
- Epsilon: >0.01
- Resultado función *perform*:0.1487

Test #5

Set de entrenamiento: 25
Set de testeo (función *perform)*: 100
Épocas: 3000
Unidades en capa oculta: 10
Check de validación: 100
Etta: 0.17

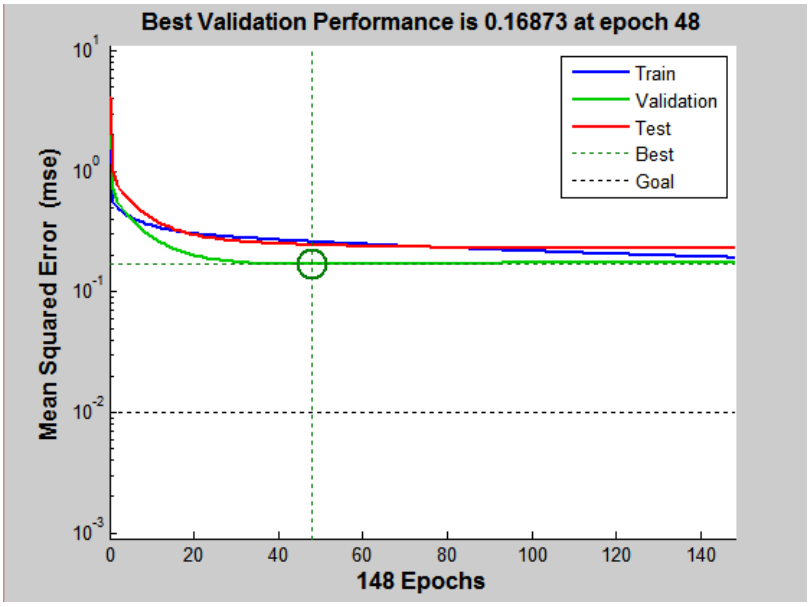


Resultados:

- Épocas insumidas: 115, terminando por cota de checks de validación
- Epsilon: >0.01
- Resultado función *perform*:0.7346

Test #6

Set de entrenamiento: 50
Set de testeo (función *perform)*: 100
Épocas: 3000
Unidades en capa oculta: 10
Check de validación: 100
Etta: 0.07

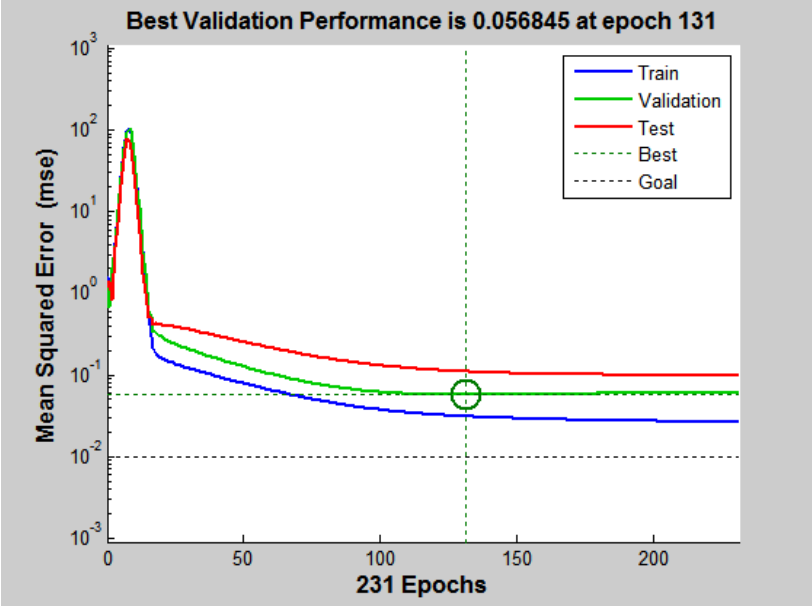


Resultados:

- Épocas insumidas: 148, terminando por cota de checks de validación
- Epsilon: >0.01
- Resultado función *perform*:0.2905

Test #7

Set de entrenamiento: 50
Set de testeo (función *perform)*: 100
Épocas: 3000
Unidades en capa oculta: 10
Check de validación: 100
Etta: 0.17

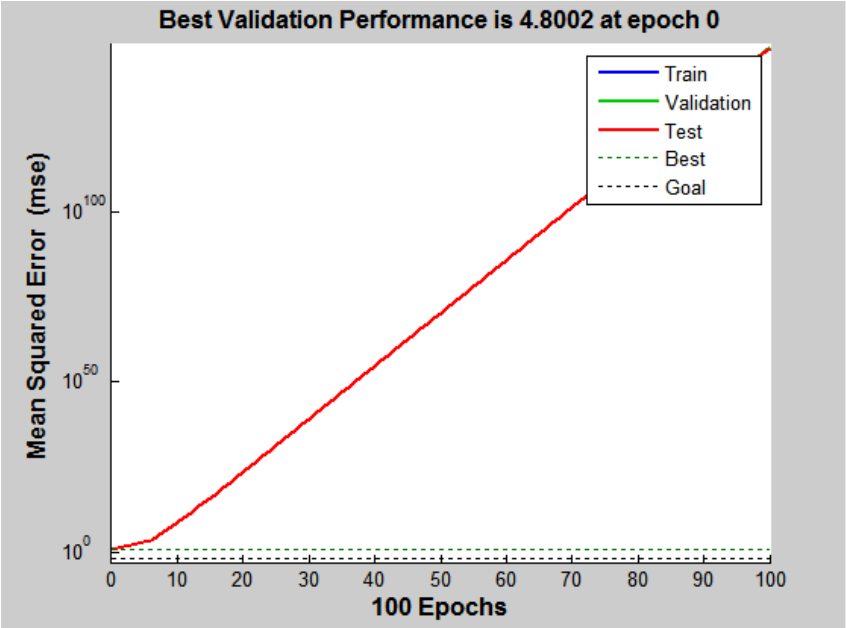


Resultados:

- Épocas insumidas: 231, terminando por cota de checks de validación
- Epsilon: >0.01
- Resultado función *perform*:0.770

Test #8

Set de entrenamiento: 50
Set de testeo (función *perform)*: 100
Épocas: 3000
Unidades en capa oculta: 10
Check de validación: 100
Etta: 0.35



Resultados:

- Épocas insumidas: 100, terminando por cota de checks de validación
- Epsilon: >>0.01
- Resultado función *perform*:3.0962

Conclusiones:

Utilizar pocas muestras para generar el conjunto de aprendizaje puede llevar a que la red neuronal no pueda aprender la solución al problema. Tal como vemos en los casos de 25 y 50, todos ellos terminaron porque la cota de checks de validación fue superada.

Viendo los resultados de *perform,* parece común denominador que cuanto menor es el etta más suave es la aproximación, y si nos mantenemos en vectores cercanos a ellos, el error cometido es bajo. Cuanto más agresivo es etta, más facilidad hay en que el error de entrenamiento y test se dispare, así como la generalización (testeada a través de la función *perform*).

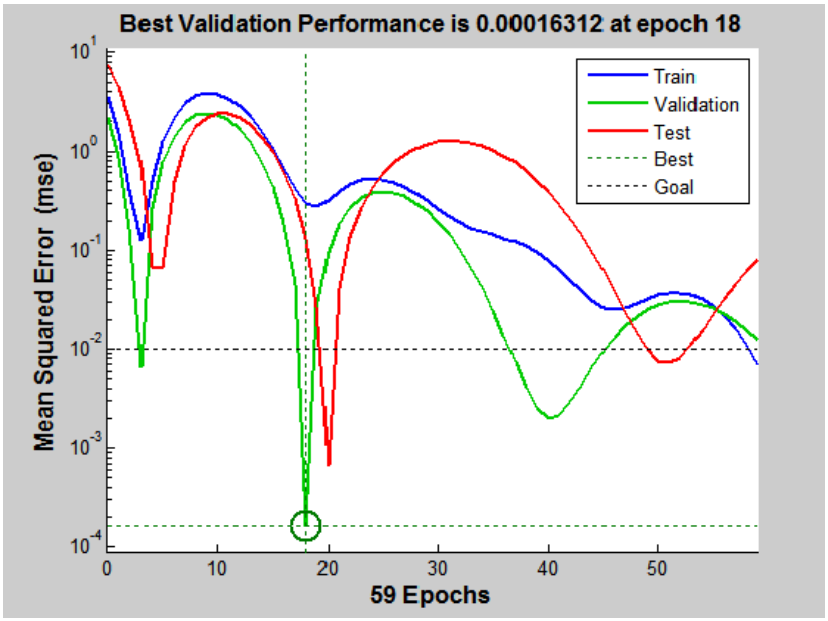
Esto nos lleva a crear la hipótesis que si tenemos un set chico de datos con los que tenemos que entrenar a la red, probablemente utilizar un etta grande no sea una buena opción.

Algoritmo Gradient descent with momentum backpropagation

Se utilizó el momentum default del algoritmo de MatLab que es 0.9.

Test #9

Set de entrenamiento: 100
Set de testeo (función *perform)*: 100
Épocas: 3000
Unidades en capa oculta: 10
Check de validación: 100
Etta: 0.07

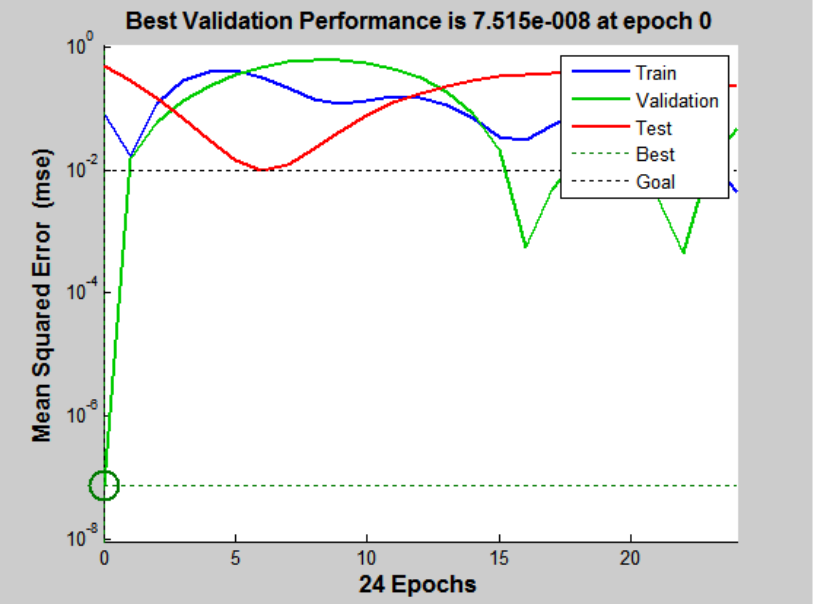


Resultados:

- Épocas insumidas: 59
- Epsilon: <0.01
- Resultado función *perform*:0.5534

Test #10

Set de entrenamiento: 100
Set de testeo (función *perform)*: 100
Épocas: 3000
Unidades en capa oculta: 10
Check de validación: 100
Etta: 0.17



Resultados:

- Épocas insumidas: 24
- Epsilon: <0.01
- Resultado función *perform*:0.6023

Conclusiones:

Consideramos muy interesantes los gráficos del algoritmo de backpropagation con moméntum ya que se observa claramente cómo fluctúa el error con el correr de las épocas. Según la documentación de MatLab, 0.9 es un coeficiente alto de moméntum, el cual resta importancia al valor del gradiente.

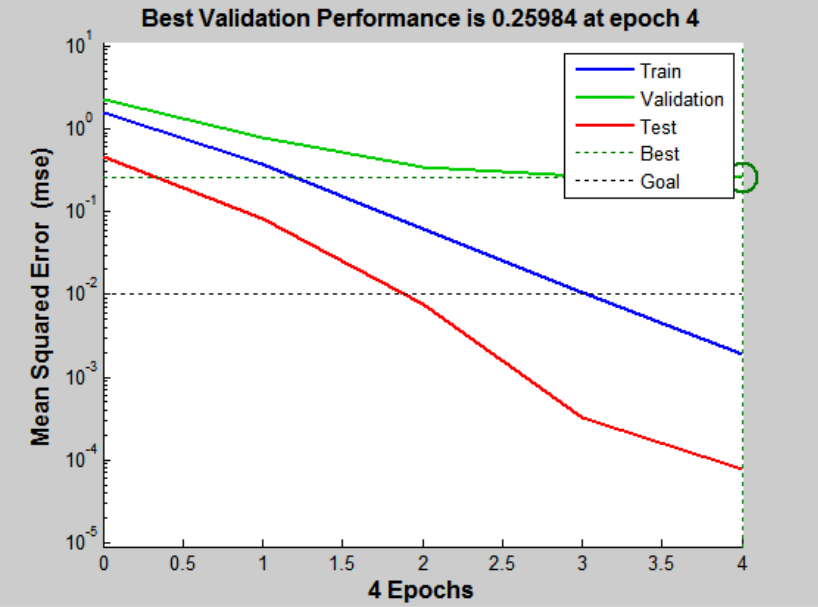
El poseer moméntum ayuda a que el etta pueda aumentar sin que el algoritmo diverja y así disminuir la cantidad de iteraciones necesarias.

Algoritmo Gradient descent with momentum backpropagation and dynamic learning rate

Se utilizó el momentum y learning rate default del algoritmo de MatLab que es 0.9 y 0.1 respectivamente.

Test #11

Set de entrenamiento: 100
Set de testeo (función *perform)*: 100
Épocas: 3000
Unidades en capa oculta: 10
Check de validación: 100
Etta: 0.07

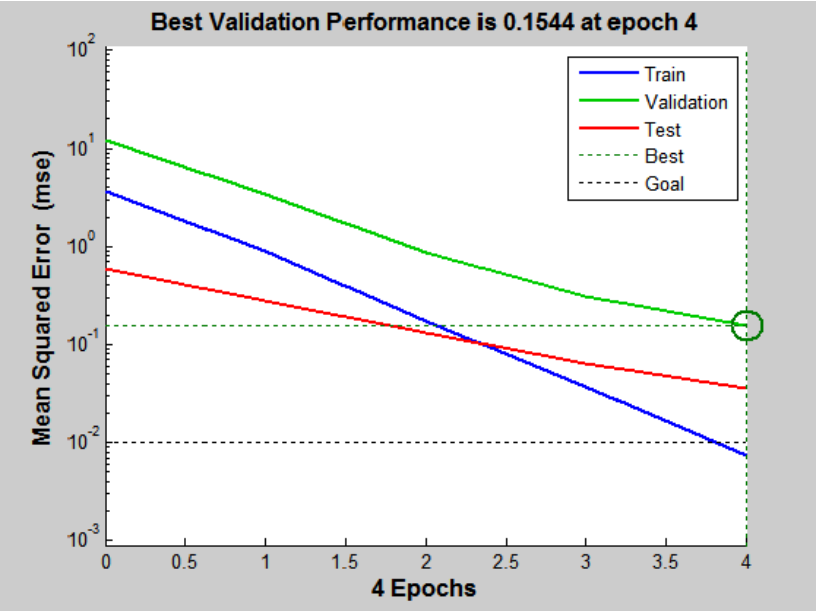


Resultados:

- Épocas insumidas: 4
- Epsilon: <0.01
- Resultado función *perform:*0.5988

Test #12

Set de entrenamiento: 100
Set de testeo (función *perform)*: 100
Épocas: 3000
Unidades en capa oculta: 10
Check de validación: 100
Etta: 0.17

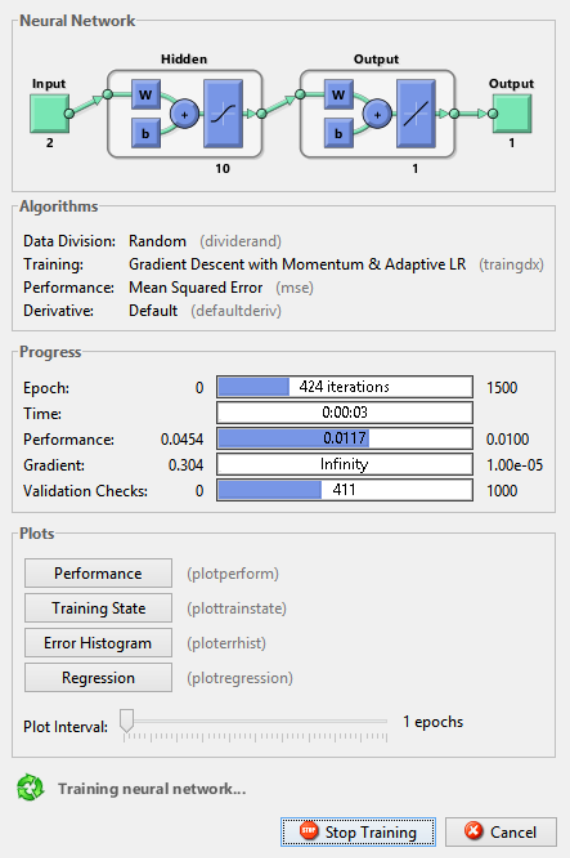


Resultados:

- Épocas insumidas: 4
- Epsilon: <0.01
- Resultado función *perform*:0.3339

Test #13

Set de entrenamiento: 100
Set de testeo (función *perform)*: 100
Épocas: 3000
Unidades en capa oculta: 10
Check de validación: 100
Etta: 0.35



Resultados:

- El gradiente se hace infinito y MatLab se cuelga. Tiene sentido que si el gradiente es infinito no haya forma de volver de él ya que como lo dice, es infinito. Cabe destacar que mientras el algoritmo corre, el gradiente cambia de valores radicalmente ante cada época.

Conclusiones:

Este algoritmo mejora en *performance* sustancialmente tanto al backpropagation clásico como al que posee momentum, vemos que tomando etta 0.07 y 0.17 alcanzaron con 4 épocas para obtener error menor a 0.01.

Además, el gráfico de los errores nos muestra claramente que no suele equivocarse en las correcciones de las matrices en cada época ya que nunca aumenta el error.

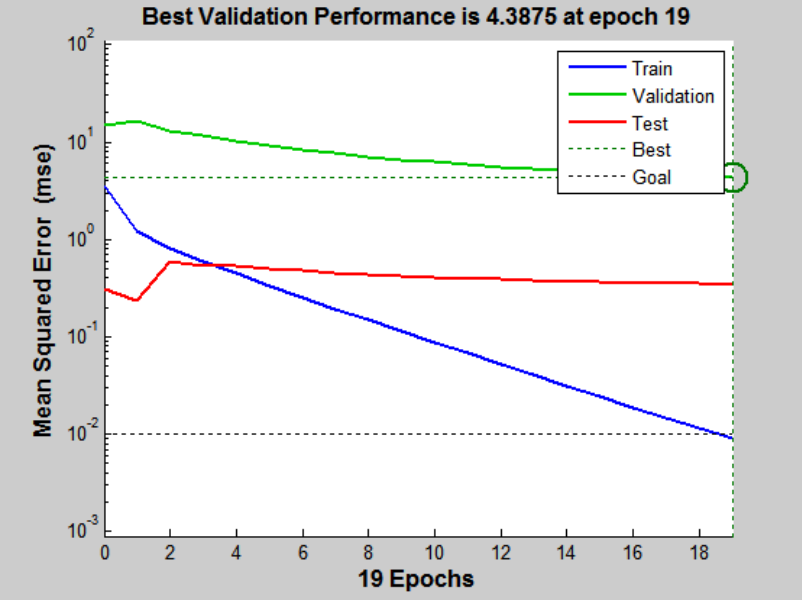
No entendemos por qué el error si etta es mayor (test #13) el algoritmo no termina. De todas formas, refuerza lo que estuvimos analizando que es: a mayor etta, mayor probabilidad de divergencia.

Modificación de cantidad de unidades ocultas

A continuación detallaremos, para el algoritmo Backpropagation, dos gráficos correspondientes a corridas con distinta cantidad de unidades en la capa oculta (25 y 50), utilizando etta 0.07.

Cabe destacar que realizamos varias pruebas teniendo resultados bastante dispares entre sí. Consideramos que los comentados a continuación son los más relevantes.

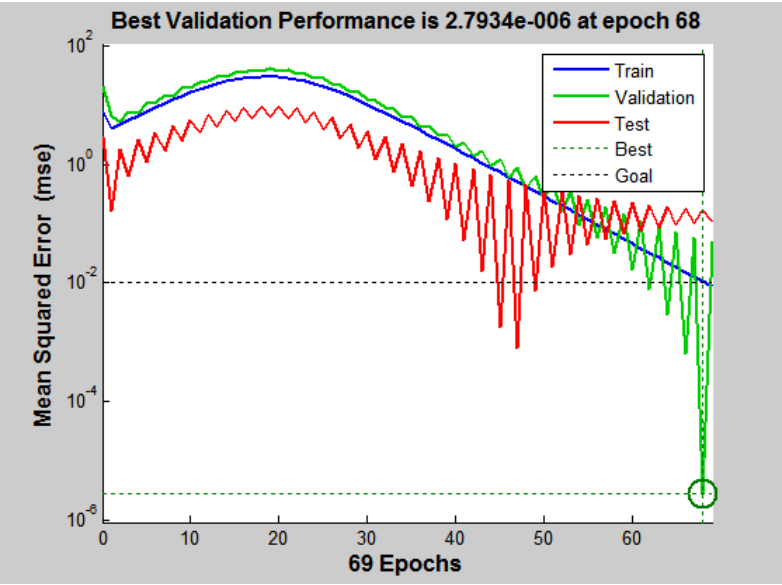
Backpropagation con 25 unidades ocultas



Conclusiones:

Obtuvimos como error de *perform* en las corridas 2 y 3. Similares observaciones se pueden hacer con respecto al gráfico, si bien aprende correctamente, se le hace imposible generalizar.

Backpropagation con 50 unidades ocultas



Conclusiones:

La mayoría de las corridas utilizando 50 unidades en la capa oculta colgaba MatLab diciendo que el gradiente es infinito. Como ya mencionamos, cuando esto sucede, el gradiente cambia mucho de valor ante cada iteración. Atribuimos la fluctuación en las funciones graficadas a este mismo problema.

La función *perform*, si es que MatLab no fallaba, nos dio como resultado valores entre 4 y 6.

Conclusiones generales a modificación de cantidad de unidades en capa oculta

Como fue explicado en clase, a mayor cantidad de unidades, mayor cantidad de dimensiones tiene nuestro problema. Estos gráficos reflejan esto, a medida que la cantidad de unidades aumenta, los errores de validación y testeo aumentan así como el algoritmo se vuelve más inestable al punto de no poder ser resuelto por un programa (en nuestro caso MatLab).

Extrapolación

Se utilizó el algoritmo Gradient Descent Backpropagation para testear extrapolación. Utilizamos etta: 0.07 ya que obtuvimos los mejores resultados con ella. El data set fue de 100 entradas entre 0 y 2pi, epsilon 0.01.

Para que no se convierta en un limitante, la cantidad de épocas se aumentó a 30000 y la cantidad de checks de validación a 10000.

Se realizaron pruebas con dos sets para la función *perform*: Uno de 5000 con valores de *X* e *Y* entre 0 y 100; el segundo de 50000 entradas con valores de *X* e *Y* entre 0 y 1000.

Ambos convergieron en el aprendizaje (ya lo sabíamos dado a que el set era conocido y testeado anteriormente). Sin embargo aplicando la función *perform* sobre el primer set de cinco mil entradas obtuvimos un error promedio de 2.8. En el segundo set, de cincuenta mil entradas, el error fue aún mayor, promediando los 4.3.

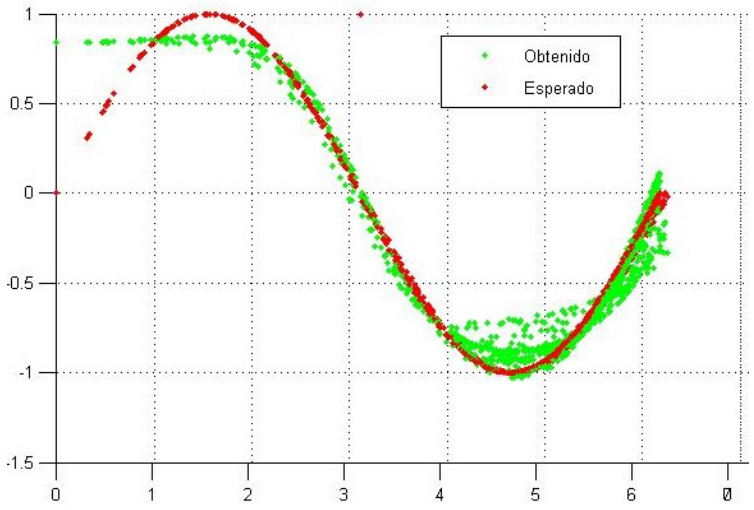
Si bien la función seno posee un período, el algoritmo no puede reconocerlo. A medida que los valores de *X* e *Y* crecen, el error se acentúa cada vez más. Consideramos como comportamiento esperable que al no tener en el set de aprendizaje entradas representativas al dominio sobre el cual se quiere extrapolar, el algoritmo no pueda aproximarlo correctamente.

Comparación Curvas

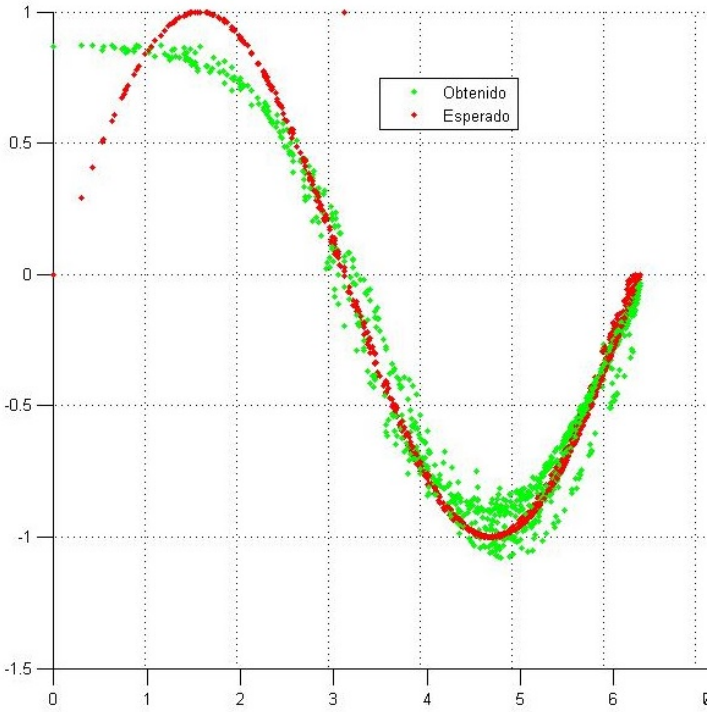
A continuación se detallan los gráficos de la función seno en comparación con cada algoritmo. En la comparación se utiliza el dataset de aprendizaje. El último gráfico posee la comparación pero con el set de test ya que queremos visualizar si el algoritmo puede extrapolar o no.

NOTA: Backpropagation y Backpropagation con moméntum utilizaron epsilon 0.01 y etta 0.07. Para el algoritmo con moméntum y *dynamic learning rate* fue necesario modificar los parámetros ya que no lograbamos que converja, utilizando epsilon 0.025 y etta 0.15

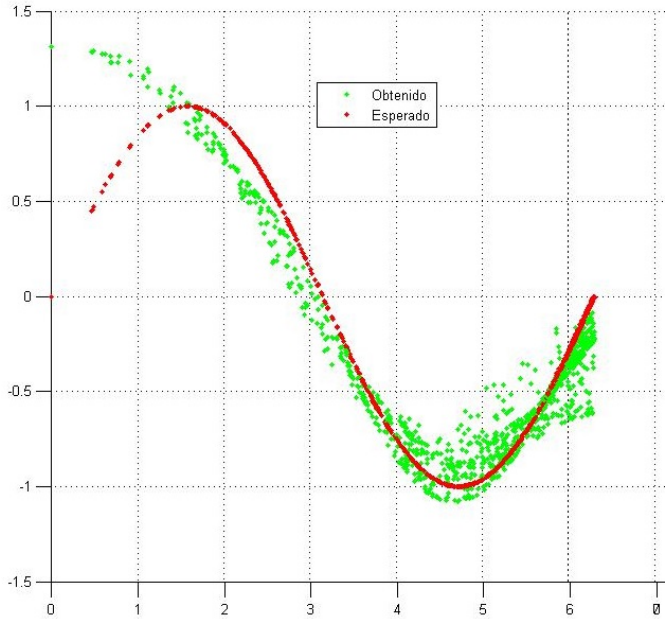
Backpropagation Standard



Backpropagation con moméntum



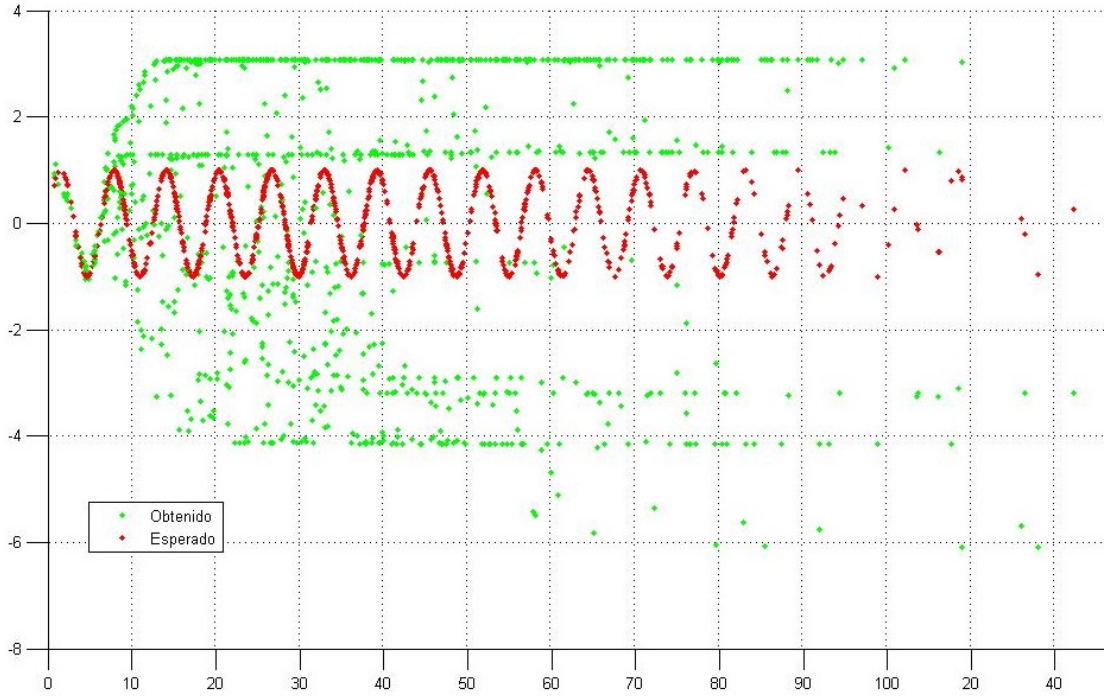
Back propagation con *dynamic learning rate* y moméntum



Los tres algoritmos muestran mayor error donde la derivada de la función es más grande. El algoritmo de backpropagation standard aproxima mejor en las líneas rectas que sus pares.

En los gráficos no se ve, pero fueron necesarias muchas más iteraciones para que el algoritmo standard convergiera. Moméntum y DLR pueden ser algo más inexactos pero convergen mucho más rápido.

Extrapolación backpropagation standard



El aprendizaje fue sobre 0 y 2pi, y la función aproxima bien en ese rango. Sin embargo es imposible la extrapolación.

Es un dato muy interesante notar esto, con las herramientas que tenemos hasta el momento (como alumnos) no podemos crear una red neuronal que generalice bien una función en todo su dominio salvo que este sea acotado.

Conclusiones generales

Las redes neuronales pueden ser de gran utilidad en problemas como los presentados en este trabajo práctico. Permiten evitar el trabajo de implementación ad-hoc de algoritmos, representando los resultados con el nivel de exactitud deseado. Sin embargo, corre por cuenta del científico configurarla correctamente para lograr el objetivo. Encontrar un dataset con una muestra representativa del conjunto puede llegar a ser muy complejo o imposible, así como definir los parámetros como etta, cantidad de unidades ocultas, cantidad de capas, cómo representar la entrada, qué función de activación utilizar y demás.

El aumento de etta en el perceptrón simple hace disminuir radicalmente la cantidad de épocas necesarias para que el algoritmo termine con la solución deseada, en cambio en un perceptrón multicapa, puede llevarlo a que no converja

Vimos mejoras significativas en costo computacional con la utilización moméntum y *dynamic learning rate*. Consideramos que utilizarlas de la manera óptima requiere algún conocimiento del dominio del problema y quizás, alguna prueba de concepto sin estas mejores pueden ser de utilidad para configurar correctamente estos agregados. No estamos seguros de que estas mejoras afinen la exactitud de la solución.