DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# Trabajo Práctico 1

## Perceptrón Multicapa

Redes Neuronales

**Grupo X**


| Integrante | LU | Correo electrónico |
|---|---|---|
| Bonet, Felipe | 668/08 | fpbonet@gmail.com |
| Martínez, Federico | XXX/XX | fedomartinez@hotmail.com |
| Avendano, Demian | XXX/XX | demian.avendano@gmail.com |


**Reservado para la cátedra**

| Instancia | Docente | Nota |
|---|---|---|
| Primera entrega | | |
| Segunda entrega | | |





**Facultad de Ciencias Exactas y Naturales**
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

# Índice

# 1. Introducción

*En este documento se realizan las actividades propuestas en el TP 1, actividades relacionadas con la implementación de una red neuronal feedforward multicapa, con el fin de lograr predicciones sobre un set de datos, mientras se estudia su comportamiento durante el entrenamiento, en el contexto de un paradigma de aprendizaje supervizado.*

## 1.1. Introducción al problema

Las redes neuronales son modelos computacionales, en los que se intenta emular el funcionamiento fisiológico de un conjunto de neuronas biológicas, interconectadas, con el fin de lograr predicciones a partir de un conjunto de datos similares, presentados previamente. Para ello se modelan, en cada unidad de procesamiento, características que tienen que ver con las condiciones de propagación de señales electroquímicas. Estas condiciones se describen y modelan a partir de observaciones de sobre cómo es transmitida información entre una neurona y otra (o sobre si), y sobre como se encuentran interconectadas.

La suma de las interacciones entre estas unidades modeladas en una topología dada, genera propiedades emergentes que permiten resolver cierto tipos de problemas (caracterizados por el *Teorema de la Aproximación Universal*). Para intentar resolver estos problemas utilizando una red neuronal, es necesario recurrir a diversas técnicas para el ajuste de las variables de la red, y en muchos casos se requiere un paso de preprocesamiento de los datos.

Se implemento una red neuronal multicapa para la predicción de dos set de datos: el primer set, tiene que ver con el diagnóstico de cancer de mamas, los datos de entrada son valores reales, mientras que el dato de salida es la presencia o no de la enfermedad; el segundo set de datos tiene que ver con la eficiencia energética de la regulación de la temperatura de un edificio, los valores de entrada son tanto enteros como reales, existen dos valores de salida reales, que tienen que ver con la carga de calefacción y de refrigeración.

## 1.2. Entrega

## 1.3. Requerimientos

- Intérprete python 2.7.
- Librerías estandar, y librería *numpy*.
- Archivo csv con uno de los dos formatos propuestos en el tp.

## 1.4. Modo de uso y opciones

Para usar este programa, se deben posicionar los archivos csv requeridos, junto a los archivos del programa, el programa se ejecuta desde python 2.7 :

```
$python script.py N args
```

donde `N` es el número de ejercicio y `args` son los argumentos optativos.

### 1.4.1. Opciones

**-ep**: Cantidad de epocas por default, 500.

**-eta**: Tasa de aprendizaje, por default $\eta = 0.05$

**-capas**: Capas ocultas de la red, cada numero separado por una coma representa una capa y cada magnitud de la capa representa el numero de neuronas de esa capa, por default = '10,10', o sea, dos capas de 10 neuronas cada una.

**-tr**: Cantidad en % del total de datos utilizado para entrenar a la red, por default = 70.

**-te**: Cantidad en % del total de datos utilizado para testing, por default = 20

**-val**: Cantidad en % del total de datos utilizados como validación, por default = 10

**-fa**: Función de activación, puede ser *tangente, tangente_optimizada o logística*, por default es *tangente*.

**-dp**: Distribución de inicialización de pesos a utilizar, puede ser *normal* o *uniforme*, por default se usa *normal*.

**-rda**: Permite cargar una red entrenada desde un archivo con formato *json*.

# 2. Desarrollo

## 2.1. Decisiones tomadas y su justificación.

# 3. Resultados

En esta sección incluiremos los resultados de la experimentación que realizamos con el perceptrón desarrollado.
La idea general de los experimentos es intentar medir la influencia de alguna(s) variable(s) específicas sobre la performance de la red. Para ello intentamos fijar todas las demás variables en valores óptimos y poner a prueba las que nos interesaban.

A medida que fuimos encontrando valores óptimos para las distintas variables, los fuimos utilizando para el resto de los experimentos subsiguientes.

En el primer experimento, intentamos determinar el mejor número de capas ocultas para utilizar en el ejercicio 1. A continuación, los resultados.

## 3.1. Ejercicio 1

### 3.1.1. Variación del número de capas ocultas.

La idea de este experimento es determinar la cantidad de capas ocultas óptimas para el ejercicio 1. Para ello fijamos las demás variables con esta configuración:

- Épocas: 250
- ETA: 0.05
- 10 neuronas por capa.
- 70 % del dataset como training, 20 % training, 10 % validación.
- Distribución de Pesos: Normal
- Entrenamiento Estocástico
- Sin Momentum
- Sin Early Stopping

Esta configuración es arbitraria y está basada en pruebas informales que realizó el equipo antes de comenzar la experimentación formal. A medida que avanzaron los experimentos y fuimos descubriendo mejores valores, los fuimos actualizando para las siguientes pruebas.

El experimento fue desarrollado así:

- Dividimos el dataset en datos de entrenamiento, validación y testing. Utilizamos la misma división para todas las ejecuciones.
- Con la configuración mencionada, corrimos 8 rondas de cada número de capas ocultas utilizando: 1, 2, 3, 5 y 10 capas. Es decir, primero se ejecutaron 8 veces el entrenamiento, validación y testing de una red con 1 capa. Luego repetimos con 2, con 3, etc.
- Para cada cantidad de capas, promediamos los resultados de cada una de las 8 rondas. Con ello obtuvimos un error final (función de costo) de entrenamiento promedio , un error de validación promedio y una efectividad de testing promedio. Llamamos efectividad de testing a la cantidad de resultados del dataset de testing correctamente predichos.
- El error final de cada ronda corresponde a la función de costo de dicha corrida dividido la cantidad de patrones procesados.

Observemos los resultados:

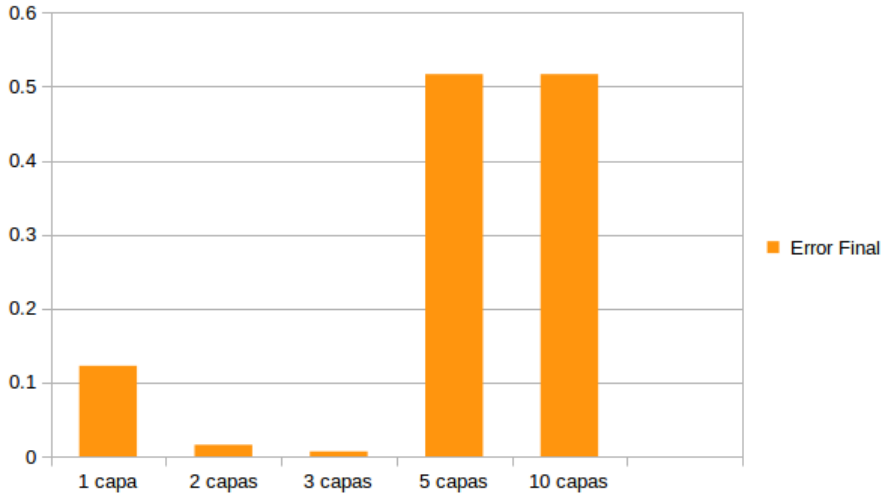



**Figura 1:** Error final (función de costo) promedio para cada cantidad de neuronas utilizando el dataset de entrenamiento

Podemos ver en la figura 1 que los mejores errores fueron obtenidos con 2 y 3 capas. Estos datos pertenecen al error final promediado, usando el dataset de entrenamiento.
Notamos también que con más de 3 capas ya los números se empiezan a distorsionar y perder mucha precisión.

Para el error final en los datos de validación, obtuvimos los siguientes resultados:

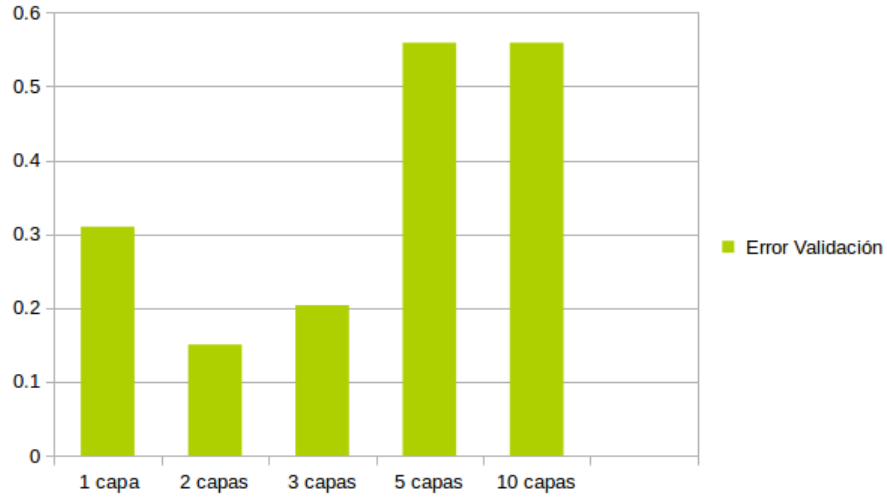



**Figura 2:** Error (función de costo) promedio para cada cantidad de neuronas utilizando el dataset de validación

En la figura 5 podemos notar que el valor de error con el dataset de validación también genera mejores números con 2 y 3 capas. Especialmente se aprecia un mejor valor para la arquitectura de 2 capas.
Observamos nuevamente que los valores más altos no rinden igual de bien que los anteriores.

Para la efectividad en la predicción de los datos de testing, observamos que la red de 1 capa generó aciertos cercanos al 82 %, mientras que las de 2 y 3 capas superan el 90 %, teniendo la de 2 capas un desempeño levemente mejor.
Nos sorprendió ver que las redes de 5 y 10 capas lograron una muy baja eficiencia, cercana al 50 %.

Teniendo en cuenta los resultados presentados en estas pruebas, nos decidimos por una arquitectura de 2 capas ocultas. Aunque se obtuvieron resultados casi igual de buenos con arquitectura de 3 capas, **la de 2 capas presenta mejor velocidad de ejecución y resultados levemente mejores**.

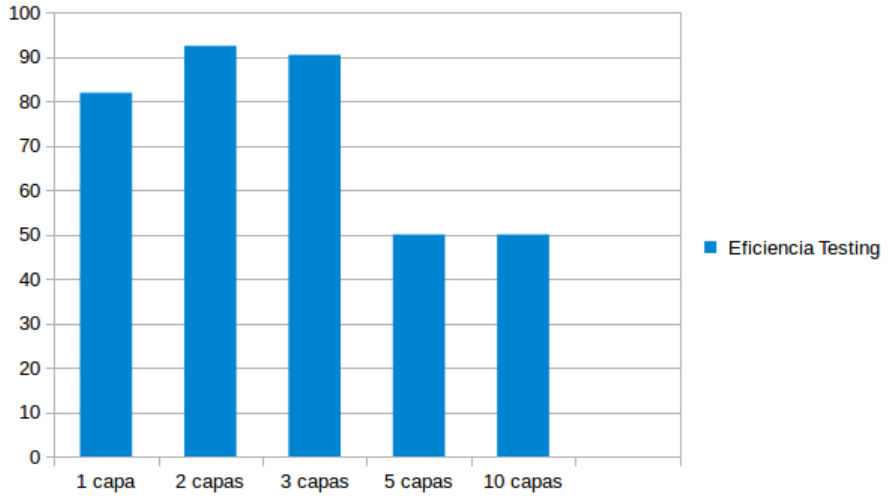



**Figura 3:** Tasa de predicciones correctas, en porcentaje, para el dataset de testing

### 3.1.2. Variación del número de neuronas ocultas.

Una vez obtenido el número óptimo de capas ocultas, quisimos determinar cual era la cantidad de nueronas que debía contener cada una de estas capas. Para simplificar el problema, decidimos que todas las capas tuvieran la misma cantidad de neuronas.
De esta forma, elegimos las siguientes medidas: 2 capas de 2, 5, 7, 10, 15 y 20 neuronas. Estas medidas fueron elegidas para intentar representar una cantidad baja de neuronas como 2,5,7 y otras más altas, por arriba de 10. No consideramos cantidades mayores de neuronas ya que los resultados no mejoran ostensiblemente pasando las 20 neuronas, teniendo en cuenta la relativa sencillez del ejercicio en cuestión. Además, arquitecturas con más de 20 neuronas por capa afectan notablemente la velocidad de ejecución de la red, relentizando las pruebas y haciéndolas engorrosas e innecesarias.

De forma similar al experimento anterior, esta prueba consistió en procesar el dataset completo 8 veces con cada cantidad de neuronas distintas, promediar los errores finales de entrenamiento y de validación junto con la eficiencia obtenida en los datos de testing.
Para el error final, obtuvimos los siguientes resultados:

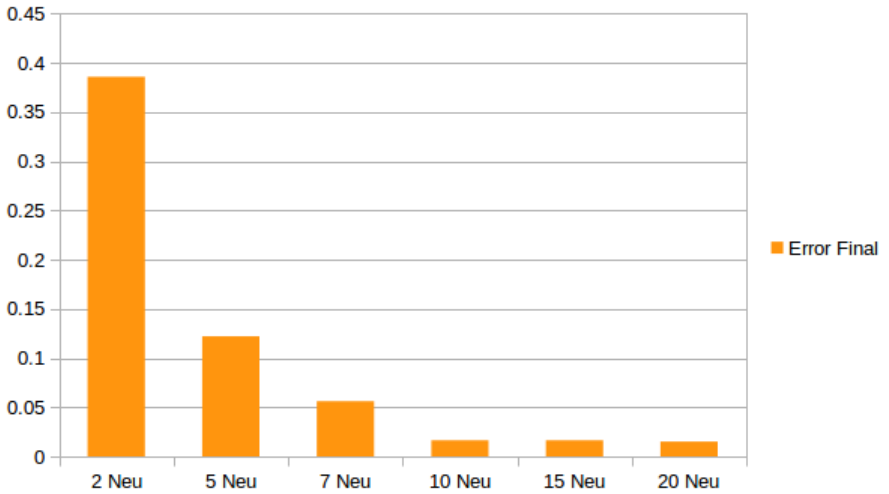



**Figura 4:** Error final (función de costo) promedio para cada cantidad de neuronas utilizando el dataset de entrenamiento

En la figura 4 se puede observar que el valor de la función de costo decrece a medida que la cantidad de

neuronas crece. Este efecto se vuelve mucho menos notable a partir de las 10 neuronas por capa.

Para el error final en los datos de validación, obtuvimos los siguientes resultados:

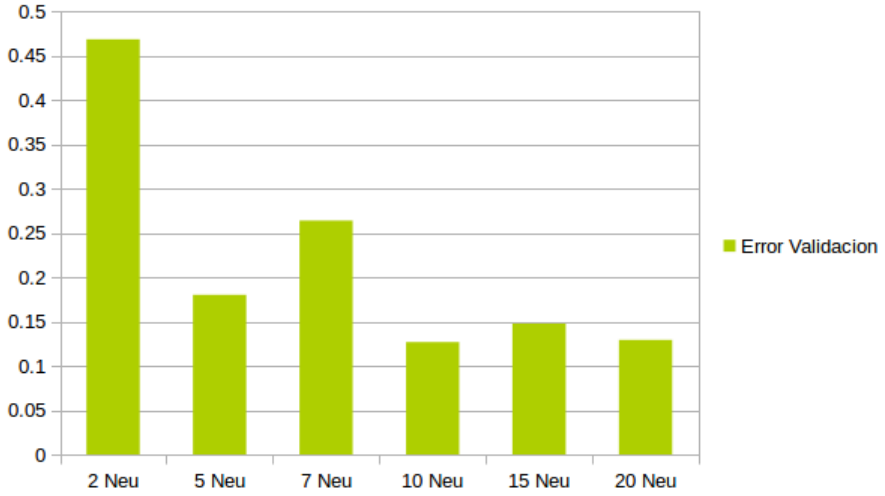



**Figura 5:** Error (función de costo) promedio para cada cantidad de neuronas utilizando el dataset de validación

En la figura 5 podemos notar que el valor de error con el dataset de validación también decrece a medida que sube la cantidad de neuronas, aunque no de forma uniforme. Podemos apreciar que a partir de las 10 neuronas, el error se mantiene por debajo de 0.15.

Para la efectividad en la predicción de los datos de testing, observamos:

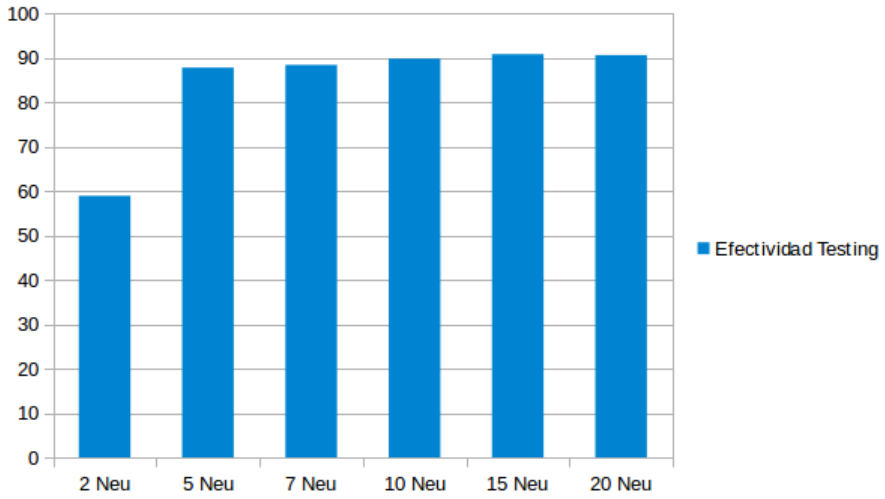



**Figura 6:** Tasa de predicciones correctas, en porcentaje, para el dataset de testing

Nuevamente podemos apreciar en la figura 6, que a medida que crece la cantidad de neuronas, crece la cantidad de aciertos sobre el conjunto de datos de test. Sin embargo, los resultados se comienzan a estancar a partir de las 10 neuronas en valores cercanos al 90 % de aciertos.

Observando los resultados obtenidos en las pruebas, decidimos que la mejor configuración es con 10 neuronas por capa. Esta decisión se justifica teniendo en cuenta que los resultados no mejoran demasiado pasando de 10 y si empeora notablemente la velocidad de la red.

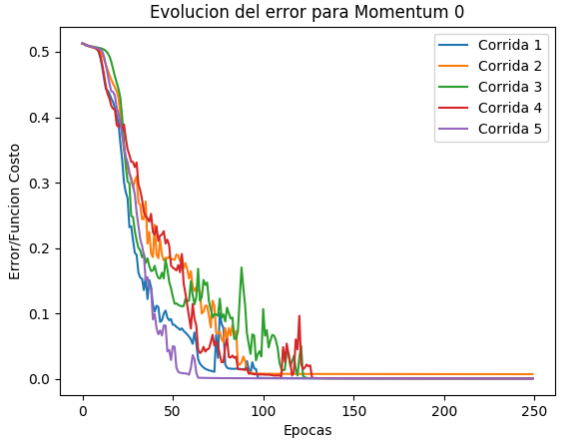
Por ende, **en vistas a los buenos resultados obtenidos con 10 neuronas y no obteniendo mejoras sustanciales con cantidades mayores**, nos quedaremos con este valor.

### 3.1.3. Performance de la red, con entrenamiento sin momentum y con momentum.

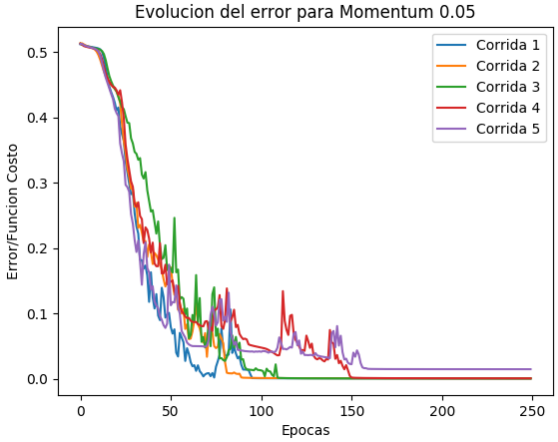
El momentum es una memoria o inercia que nos permite que los cambios en el vector de pesos $w$ sean suaves ya que incluyen información sobre el cambio de peso anterior. En el proceso de backpropagation de la red, cuando actualizamos los pesos, utilizaremos el momentum de la siguiente manera:

$$\Delta w_{ij}^{m} = \eta \delta_i^m V_j^m + \alpha \Delta w_{pq}^{m-1}$$

Para este experimento, mantenemos la configuración que veniamos utilizando en experimentos anteriores con 2 capas de 10 neuronas. Probaremos el comportamiento de la red variando el momentum desde 0 (o sin momentum) hasta un valor de momentum de 0,9. No utilizaremos valores mayores de 0,9 ya que significaría que el peso anterior que tenía el eje se estaría acumulando con el nuevo peso en su completitud, lo cual nos parece una exageración en este caso. Dado que en posteriores experimentos realizaremos pruebas variando el momentum y el learning rate en conjunto, para este experimento, mantendremos el learning rate de 0,05.




**(a)** Evolución del error de entrenamiento para momentum 0 (sin momentum)




**(b)** Evolución del error de entrenamiento para momentum 0.05

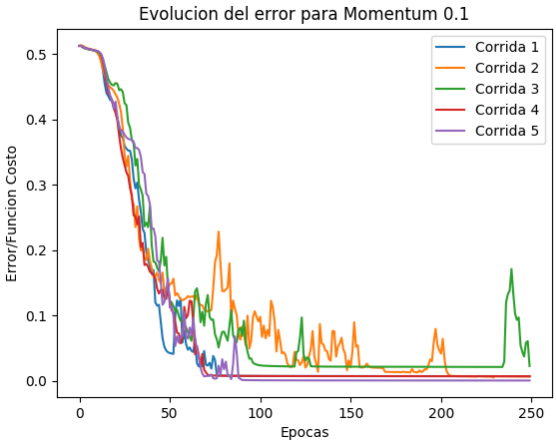
A partir de los resultados podemos intuir que manteniendo el learning rate elegido, no usar momentum o usar momentum 0,05 o 0,5 da los mejores resultados. Claramente usar 0,9 es casi como al nuevo peso de cada eje sumarle el peso anterior, por eso la oscilación tan pronunciada. Sin embargo, y para llegar a una conclusión del mejor valor de momentum, analizemos los resultados de la evolución del error para nuestro conjunto de validación:

Teniendo en cuenta ahora también nuestro error de validación final para todas las corridas y valores de momentum, podemos concluir entonces que manteniendo nuestro learning rate fijo en 0,05, lo mejor es no usar momentum.

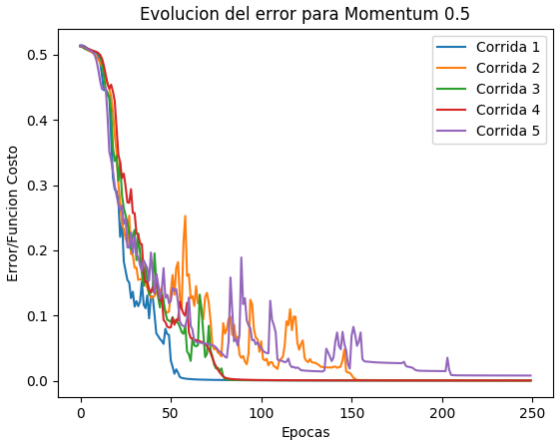
### 3.1.4. Performance de la red, con entrenamiento sin y con parámetros adaptativos.

Utilizar parámetros adaptativos, como variar el learning rate, nos permiten volver a un estado anterior de la red y corregir el learning rate que usamos en ese momento para que esta vez nos de un mejor resultado. Para nuestra configuración adaptativa, si el error crece continuamente reducimos el learning a la mitad, mientras que si el error va en bajada, aumentamos el learning rate en un 10 %. Manteniendo los parámetros de configuración que venimos utilizando, hicimos experimentos sobre entrenamientos estocásticos, batch y mini batch.

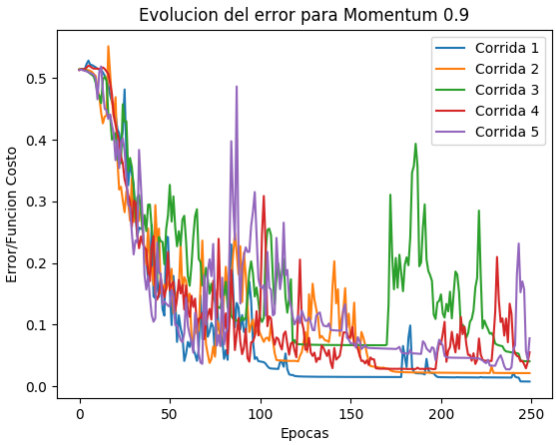
A partir de los resultados de entrenamiento estocástico, podemos observar que usar parámetros adaptativos no solo no ayuda al aprendizaje de la red, sino que además lo perjudica en gran magnitud. Esto puede deberse principalmente a que la red esta continuamente ajustando sus pesos en cada entrada del dataset para poder con el learning rate actual, mejorar la precisión y reducir el error. Si cambiamos el learning rate los pesos que ajustamos en cada paso dejan de tener sentido y el entrenamiento termina dando malos resultados.



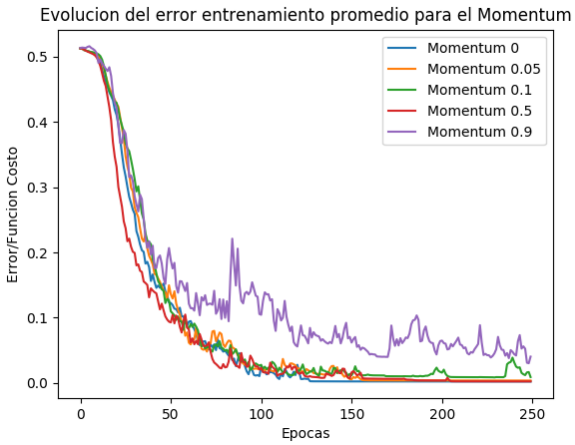

**(a)** Evolución del error de entrenamiento para momentum 0.1




**(b)** Evolución del error de entrenamiento para momentum 0.5




**(a)** Evolución del error de entrenamiento para momentum 0.9




**(b)** Evolución del error de entrenamiento promedio para todos los valores de momentum

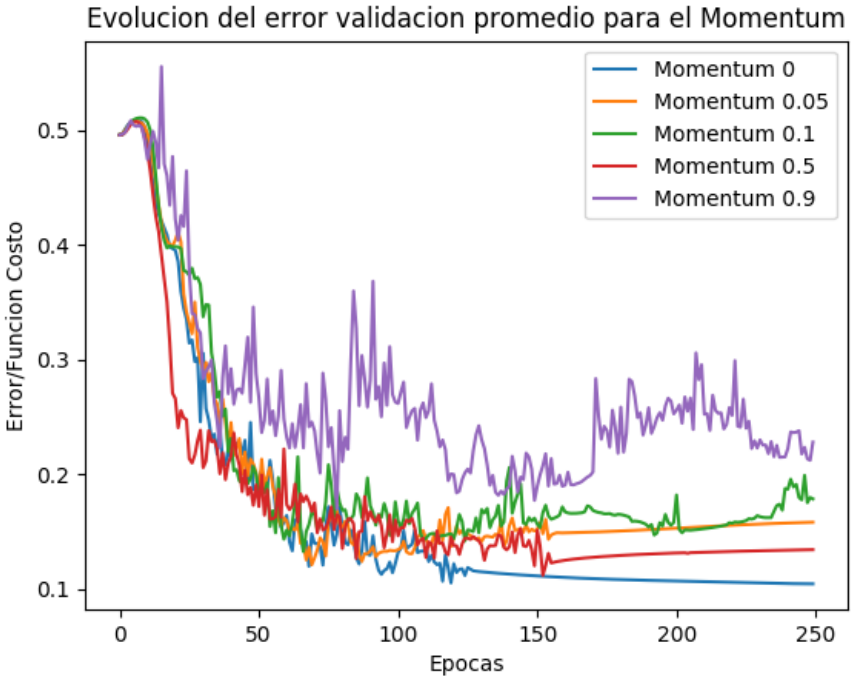
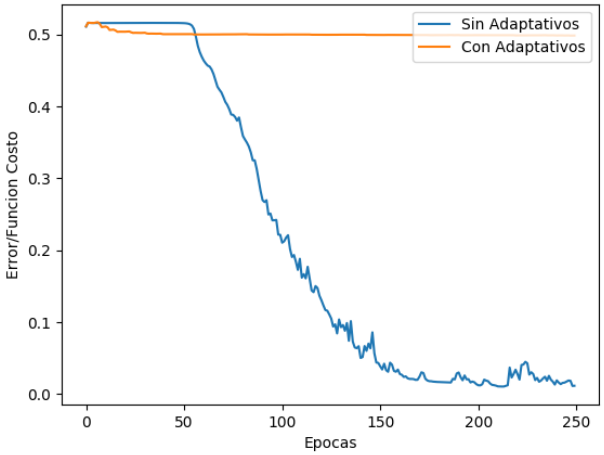
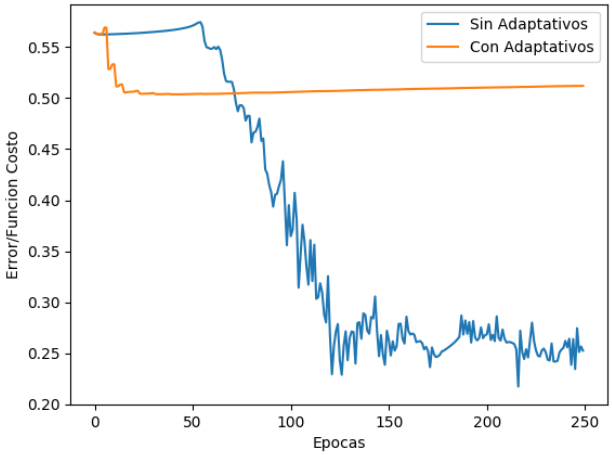



**Figura 10:** Evolución del error de validación promedio para todos los valores de momentum

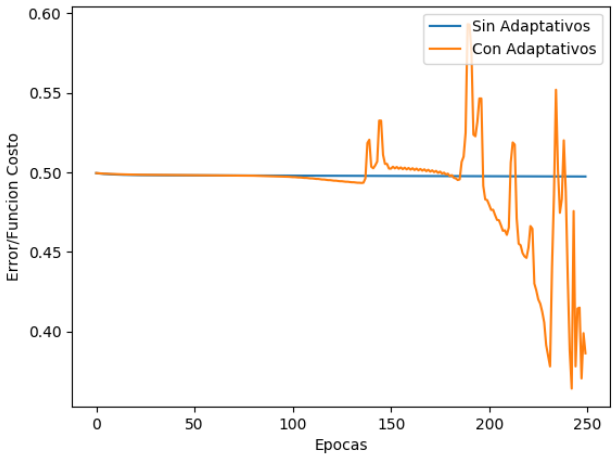



**(a)** Comparación del error en el conjunto de entrenamiento para estocástico (250 épocas)

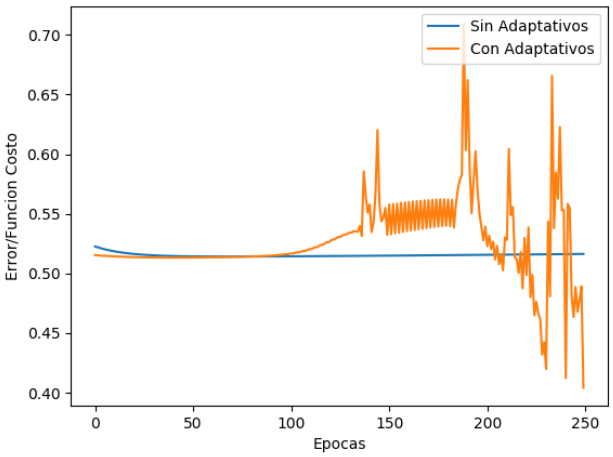



**(b)** Comparación del error en el conjunto de validación para estocástico (250 épocas)

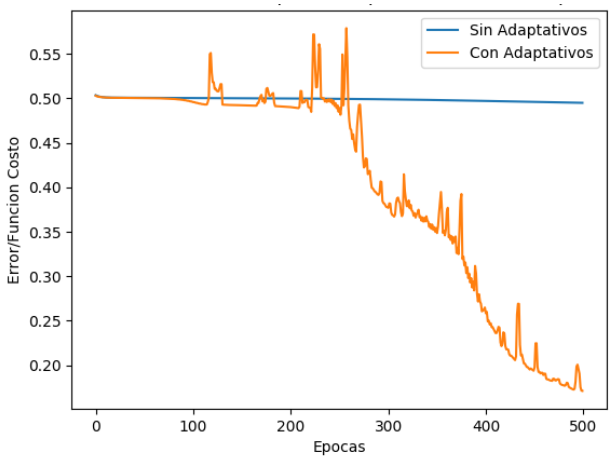
Con respecto a batch, nos dimos cuenta que los parámetros adaptativos permiten que se obtengan muy buenos resultados. Incluso se puede ver que el error de validación llega a ser menor que en el entrenamiento estocástico. Sin embargo, tuvimos que utilizar el doble de épocas para conseguir los mismos.



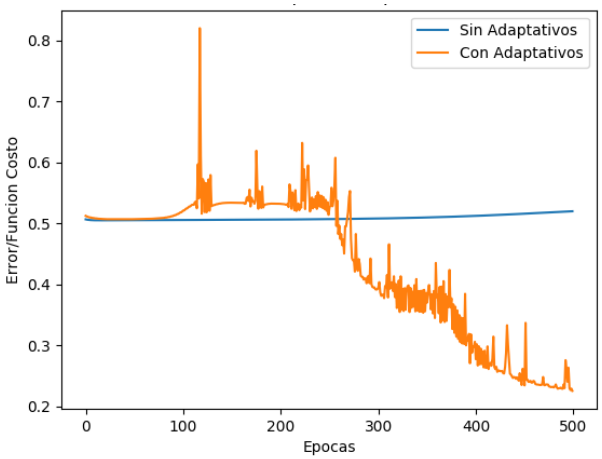

**(a)** Comparación del error en el conjunto de entrenamiento para batch (250 épocas)




**(b)** Comparación del error en el conjunto de validación para batch (250 épocas)




**(a)** Comparación del error en el conjunto de entrenamiento para batch (500 épocas)




**(b)** Comparación del error en el conjunto de validación para batch (500 épocas)

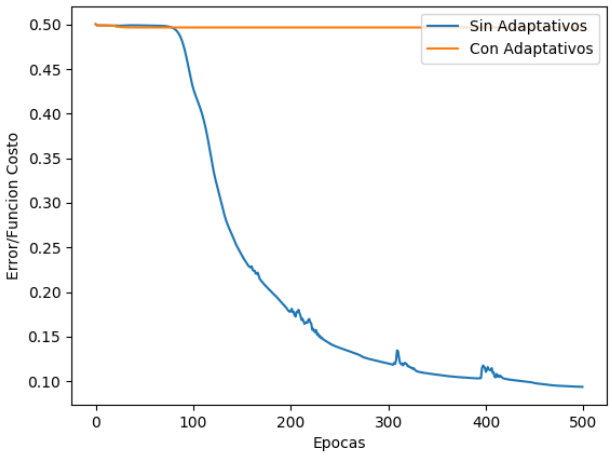
Para el entrenamiento mini batch, confirma la idea de que el learning rate adaptativo es útil solo en batch y no para estocástico. Realizamos un par de pruebas y vimos que mientras más cerca de 1 (estocástico) estuviese el tamaño del batch, peor resultados da el adaptivo, mientras que si más cerca está del tamaño total del dataset de entrenamiento, mejor funciona el learning rate adaptativo.

Nuestra conclusión para este experimento es que si bien se llegan a alcanzar buenos resultados con el entrenamiento batch, es necesario utilizar el doble de épocas lo cual hace que tarde mucho más el entrenamiento. Online learning en la mitad de las épocas y sin learning rate adaptativo obtiene resultados similares.

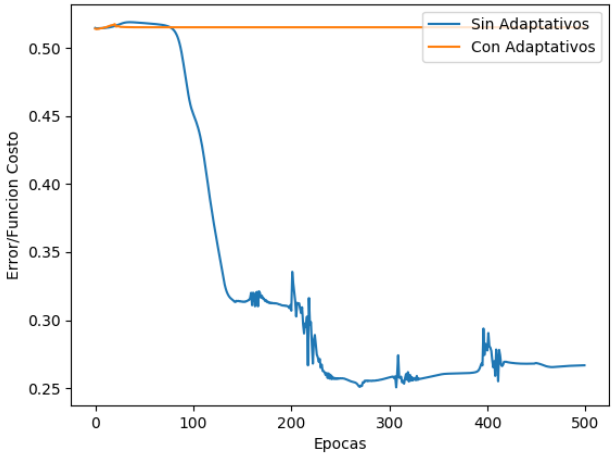
### 3.1.5. Performance de la red, variando simultáneamente el factor de aprendizaje $\mu$, y el parámetro $\alpha$ del momentum.

### 3.1.6. Performance de la red, variando técnicas de entrenamiento (estocástico, batch y mini batch) y de inicialización de pesos

En este experimento decidimos probar dos variables en simultáneo. Probamos la performance de las redes utilizando distintas técnicas de entrenamiento: estocástico, batch y mini batch con distintos tamaños. Al mismo tiempo, variamos la distribución utilizada para la inicialización de los pesos de la red. Utilizamos distribución normal y uniforme.




**(a)** Comparación del error en el conjunto de entrenamiento para mini batch de 10, cercano a ser estocástico (500 épocas)




**(b)** Comparación del error en el conjunto de validación para batch de 10, cercano a ser estocástico (500 épocas)

A la hora de elegir tamaños de batch para las ejecuciones mini batch, elegimos los siguientes tamaños: 5,10,100, mitad del dataset y dataset completo (batch). La idea es probar con batchs pequeños como 5 y 10, cercanos en teoría al online learning y también con algunos tamaños más grandes. Probamos también utilizando el dataset completo.

Al igual que en los primeros dos experimentos, juzgaremos las modalidades midiendo la performance final de error sobre dataset de training, error sobre el de validación y efectividad de aciertos prediciendo los patrones de testing.
Observemos los resultados:

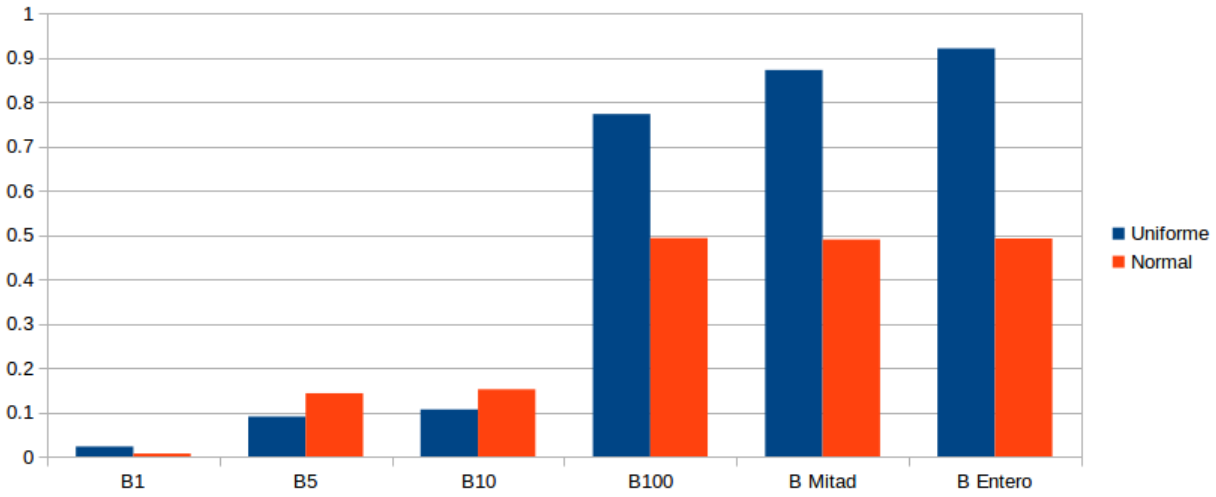



**Figura 15:** Error final (función de costo) promedio para cada cantidad de neuronas utilizando el dataset de entrenamiento

En la primera figura de este experimento mostramos la función de costo obtenida con las diferentes técnicas.
Notación:

- $B1$ = Entrenamiento estocástico.
- $BX$ = Mini batch de tamaño $X$.
- $BMitad$ = Mini batch de tamaño equivalente a la mitad del dataset.
- $BEntero$ = Batch común.

A su vez, mostramos en color azul los resultados obtenidos utilizando distribución de pesos uniforme y en naranja, distribución normal.
Podemos ver que, tanto para aprendizaje estocástico como para los mini batchs más pequeños, se obtuvieron mejores resultados utilizando la distribución normal.

Para el error final en los datos de validación, obtuvimos los siguientes resultados:

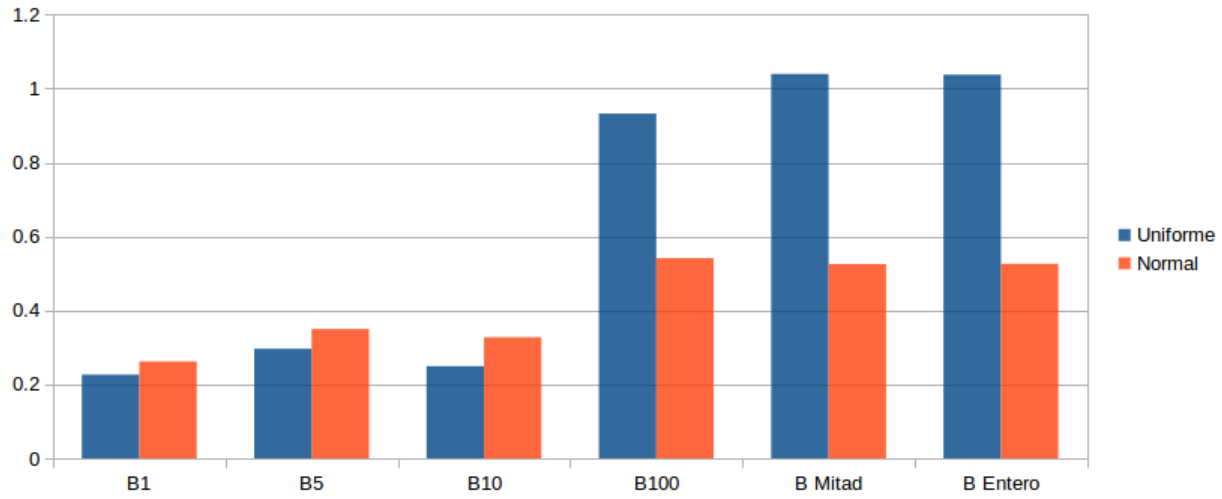



**Figura 16:** Error (función de costo) promedio para cada cantidad de neuronas utilizando el dataset de validación

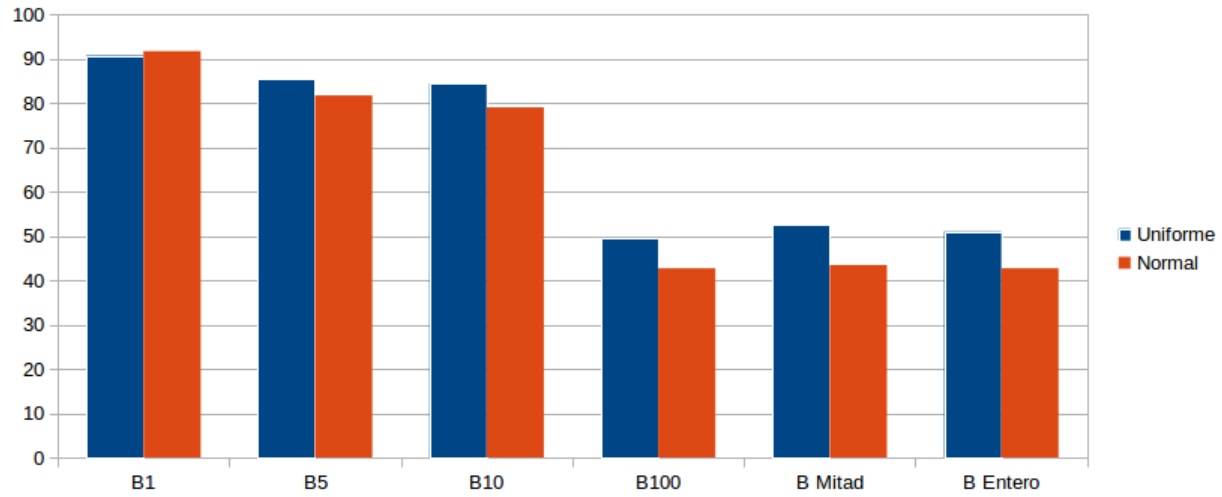



**Figura 17:** Tasa de predicciones correctas, en porcentaje, para el dataset de testing

#### 3.1.7. Performance de la red, sin y con preprocesamiento de los patrones.

#### 3.1.8. Performance de la red, sin y con early-stopping.

#### 3.1.9. Performance de la red, variando las funciones de activación y/o sus parámetros.

### 3.2. Descripción, justificación y performance, de la solución óptima propuesta.

### 3.3. Conclusiones.

# 4. Apéndice - Código

# 5. Bibliografía

## Referencias

[1] ANSI C Yacc grammar - Jeff Lee
`https://www.lysator.liu.se/c/ANSI-C-grammar-y.html`