

---

## Trabajo Práctico N° 3

### Machine Learning

#### EJERCICIOS

**Ejercicio 1. Familiarizarse con el código de la red neuronal feedforward fully connected de 1 capa oculta explicada en clase.**

Vimos el video subido en Aula Abierta que explica el funcionamiento del código y lo ejecutamos en Modo Debug para ver las variables y entender el funcionamiento.

**Ejercicio 2. Modificar el programa para que**

- Mida la precisión de clasificación (accuracy) además del valor de Loss**
- Utilice un conjunto de test independiente para realizar dicha medición (en lugar de utilizar los mismos datos de entrenamiento). Este punto requiere generar más ejemplos.**

En el caso de este algoritmo de clasificación, la salida del algoritmo es a qué clase pertenece un dato de entrada. Por lo que la precisión del algoritmo (**accuracy**) se mide como:

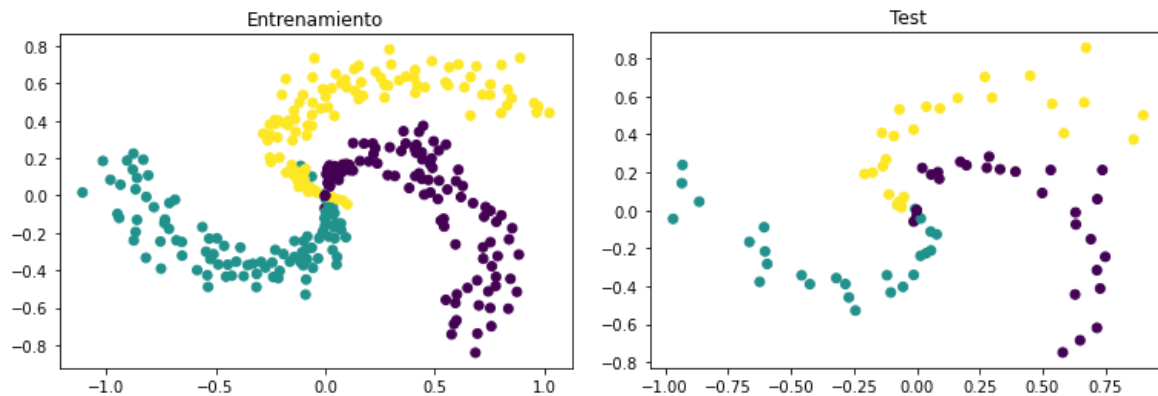
$$accuracy = \frac{\text{cant ejemplos clasificados correctamente}}{\text{cantidad total de ejemplos clasificados}}$$

en el código esto se implementó como se ve a continuación:

```
# Testing =====  
mm = np.size(x_test, 0)  
resultados = clasificar(x_test, pesos)  
  
accuracy_test = 0  
  
for j in range(mm):  
    if resultados[j]==t_test[j]: # si fue correcta la clasificacion  
        accuracy_test+=1  
accuracy_test/=mm  
  
print(" * Testing Accuracy epoch ",i,":",accuracy_test)
```

Donde se midió la precisión para un conjunto de validación que nunca ha sido visto durante el entrenamiento. Además al final del algoritmo se vuelve a calcular una precisión con los datos de test, que son otro conjunto de datos (que tampoco se han visto durante el entrenamiento) que se usa para testear el modelo una vez ya entrenado y validado.

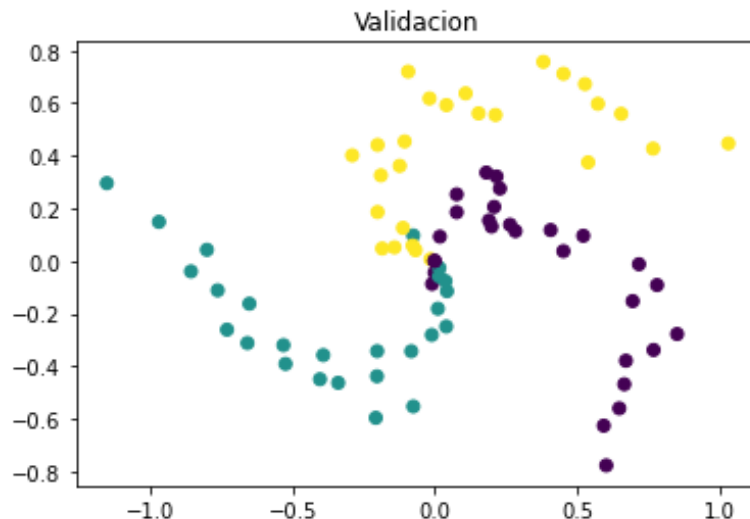
Estos conjuntos se pueden ver a continuación:



**Ejercicio 3. Agregar parada temprana, utilizando un conjunto de validación, distinto del conjunto de entrenamiento y de test (este punto requiere generar más ejemplos).**

**Esto es: verificar el valor de loss o de accuracy cada N epochs (donde N es un parámetro de configuración) utilizando el conjunto de validación, y detener el entrenamiento en caso de que estos valores hayan empeorado (puede incluirse una tolerancia para evitar cortar el entrenamiento por alguna oscilación propia del proceso de entrenamiento).**

Para empezar con este punto primero generamos más datos de forma aleatoria como se puede apreciar en la siguiente imagen (además de los conjuntos de Train y Test anteriores):



Para la detención temprana, creamos 2 hiperparametros más, que son:

[illegible]

Uno es cada cuantas **epochs** validamos para ver si hay que **detener** el entrenamiento, y el otro es una **tolerancia** de que cuando empeora la precisión más de un 1 % se **detiene**.

En el código esto se implementó de la siguiente forma:

```
# Validation =====

mm = np.size(x_validation, 0)
resultados = clasificar(x_validation, pesos)

accuracy_validation = 0

for j in range(mm):
    if resultados[j]==t_validation[j]: # si fue correcta la clasificacion
        accuracy_validation+=1
accuracy_validation/=mm

print("Validation Accuracy epoch ",i,":",accuracy_validation)

if prevAccuracy==None:
    prevAccuracy = accuracy_validation

elif accuracy_validation < (prevAccuracy*(1-tolerancia)):
    # empezo a empeorar
    print("Empeoro!!")
    break

else:
    # print(accuracy_validation,"<=",prevAccuracy,"margen",prevAccuracy*(1-tolerancia))
    prevAccuracy = accuracy_validation
```

Cada cierta cantidad de epochs (según el hiper parámetro), calculamos la precisión con el conjunto de validación. Si este va mejorando continuamos, pero si este empeora más de una tolerancia fijada entonces detenemos el entrenamiento antes de tiempo.

Un ejemplo de los resultados obtenidos con los siguientes hiperparametros:

```
# Inicializa pesos de la red
NEURONAS_CAPA_OCULTA = 100
NEURONAS_ENTRADA = 2
```

```
# Entrena
LEARNING_RATE=1
EPOCHS=10000

DETENCION_TEMPRANA = 500 # se vali
TOL_DETENCION_TEMPRANA = 1/100 # s
```

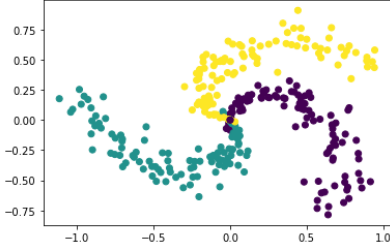
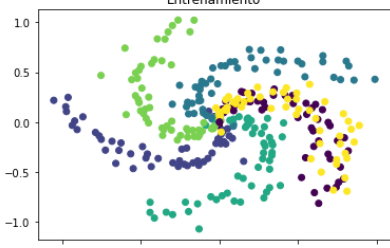
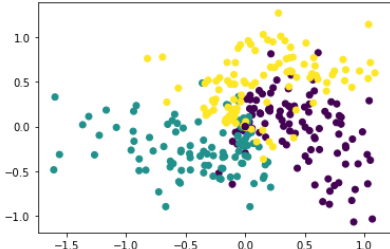
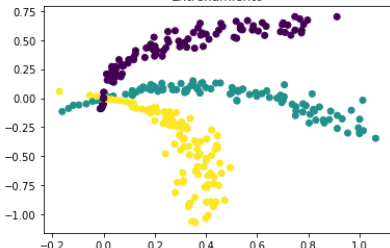
Es el siguiente:

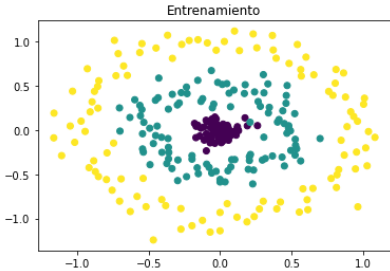
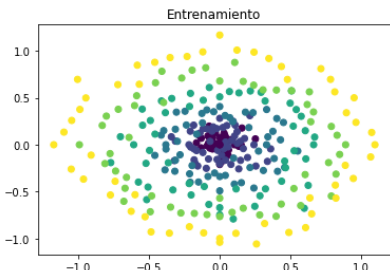
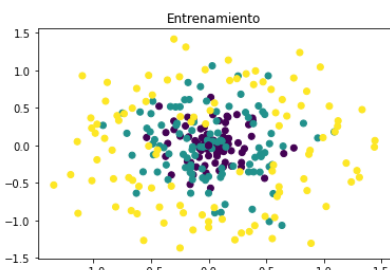
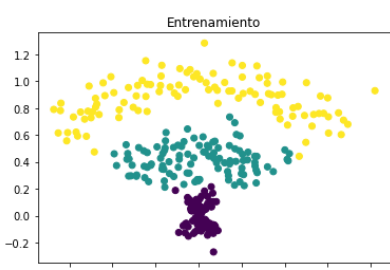
```
Training Loss epoch 4000 : 0.13490689999288116
Training Accuracy epoch 4000 : 0.953125
Validation Accuracy epoch 4000 : 0.95
Empeoro!!

* Termino el entrenamiento en 4000 epochs
* Testing Accuracy epoch 4000 : 0.9375
```

Logrando una **precisión de Test** del **93,75 %**, al cabo de **4000 epochs** solamente.

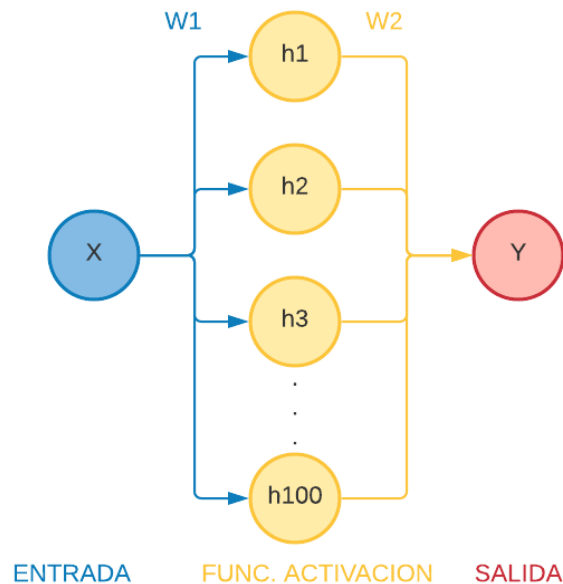
**Ejercicio 4. Experimentar con distintos parámetros de configuración del generador de datos para generar sets de datos más complejos (con clases más solapadas, o con más clases). Alternativamente, experimentar con otro generador de datos distinto (desarrollado por usted). Evaluar el comportamiento de la red ante estos cambios.**

Patron Datos	Cant Class	Amp	Factor Angulo	Ejemplo	Resultados
Espiral	3	0.1	0.79		EPOCHS: 8500  Testing Accuracy: 93,75 %
Espiral	6	0.1	0.79		EPOCHS: 3500  Testing Accuracy: 77,50 %
Espiral	3	0.3	0.79		EPOCHS: 3000  Testing Accuracy: 72,50 %
Espiral	3	0.1	0.3		EPOCHS: 2500  Testing Accuracy: 97,50 %

Círculo	3	0.1	1		<p>EPOCHS: 9500</p> <p>Testing Accuracy: 98,75 %</p>
Círculo	6	0.1	1		<p>EPOCHS: 1000</p> <p>Testing Accuracy: 72,50 %</p>
Círculo	3	0.3	1		<p>EPOCHS: 1500</p> <p>Testing Accuracy: 76,25 %</p>
Círculo	3	0.1	0.3		<p>EPOCHS: 9500</p> <p>Testing Accuracy: 98,75 %</p>

### Ejercicio 5. Modificar el programa para que funcione para resolver problemas de regresión:

Para la solución de un problema de regresión se implementó una red neuronal estructurada como sigue:



- Una única neurona de entrada “X”.
- 100 neuronas en una capa media.
- Una única neurona de salida “Y”.

El problema de regresión resuelto mediante redes neuronales se basa en que, para este caso particular, la neurona de salida pueda predecir el comportamiento de un conjunto de datos numéricos por medio de extrapolación. Dicha técnica puede implementarse gracias a que previamente la red fue entrenada con ejemplos que mostraron un comportamiento similar, de forma que la red conoce parcialmente la distribución de los datos.

**a. Debe modificarse la función de pérdida y sus derivadas, utilizando por ejemplo MSE.**

Para la solución del problema de regresión mediante redes neuronales, se aplicó la siguiente función de pérdida:

$$L_i(W) = (t_i - y_i)^2$$

$$L(W) = \frac{1}{m} \sum_i L_i(W)$$

función MSE (Mean Square Error), calculada como sigue:

```
# LOSS --> Cálculo de MSE
L = np.power((t_train - y), 2)
loss = (1 / m) * np.sum(L)
```

donde t representa el “target” o salida deseada, e y la salida calculada por la red neuronal.

Al modificar la función “loss” del algoritmo, también se modifica su derivada, la cual resulta:

$$\frac{\partial L}{\partial Y} = -\frac{2(T - Y)}{m} = \frac{2(Y - T)}{m} \quad (m \times q)$$

Y es útil para el cálculo de los pesos sinápticos W1 y W2; y para el cálculo de los sesgos B1 y B2, requeridos para mejorar el rendimiento de la red durante el entrenamiento.

Se introdujo además una parada temprana por medio de la función antes explicada. Se implementa una parada cada 500 epochs, se calcula la función “loss” utilizando los datos de validación, y por comparación con el valor anterior calculado más cierta aleatoriedad, se establece un criterio de detención.

```
resultados = clasificar(x_validation, pesos, sig)
L_validation = np.power((t_validation - resultados), 2)
loss_validation = (1 / mm) * np.sum(L_validation)
print("Validation Loss epoch", i, ":", loss_validation)

if prevLoss==None:
    prevLoss = loss_validation
elif loss_validation<=prevLoss:
    prob = randint(0, 100)/100
    # hacemos una parada temprana en base a una
probabilidad
    if prob<(i/epochs):
        plt.scatter(x_validation[:, 0], resultados)
        plt.title("Aproximacion con datos de validacion")
        plt.show()
        break
    else:
        prevLoss = loss_validation
else:
    prevLoss = loss_validation
```

- b. Debe crearse un generador de datos nuevo para que genere datos continuos (pueden mantenerse igualmente 2 entradas; en caso de usar más entradas puede requerirse más capas en la red neuronal).**

Para generar los datos de entrada, como se explicó anteriormente se implementó una única entrada “X”, variable continua que representa las abscisas correspondientes a dos funciones:

1. Lineal
2. Cuadrática

### Regresión 1: Función lineal

La salida deseada o “target” está dada por una distribución lineal  $f(x) = mx + b = t$ , de pendiente m elegida por el usuario, y de ordenada b generada aleatoriamente como sigue:

```

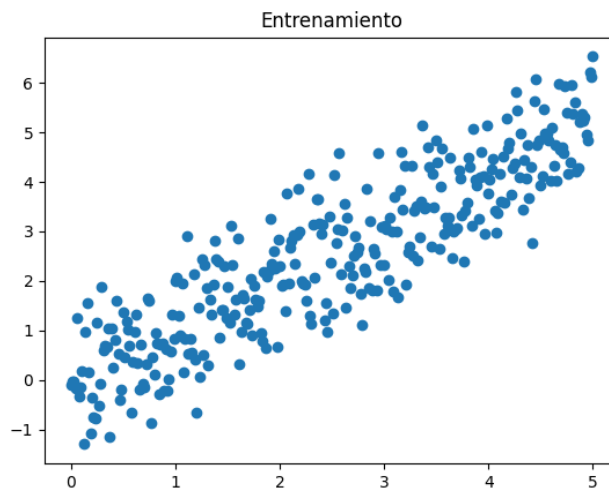
AMPLITUD_ALEATORIEDAD = 0.8
# Entradas: 1 col x1
x = np.zeros((cantidad_ejemplos, 1))
t = np.zeros((cantidad_ejemplos, 1))
randomgen = np.random.default_rng()

x1 = np.linspace(0, 5, cantidad_ejemplos)
x2=x1+AMPLITUD_ALEATORIEDAD*randomgen.standard_normal(size=cantidad_ejemplos)

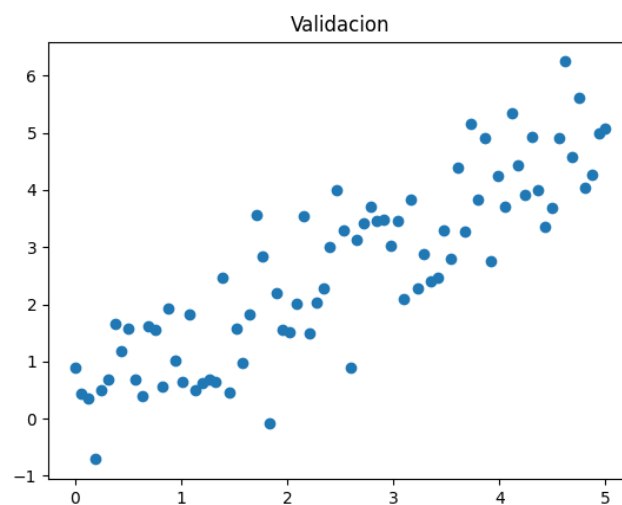
# Generamos un rango con los subindices de cada punto generado.
indices = range(0, cantidad_ejemplos)
#Creamos la matriz de ejemplos de regresion
x1.sort() #Ordenamos en orden creciente las abscisas
x[indices] = np.c_[x1]
t[indices] = np.c_[x2]

```

De manera que en una gráfica de “x vs t” para m=1 se obtiene:

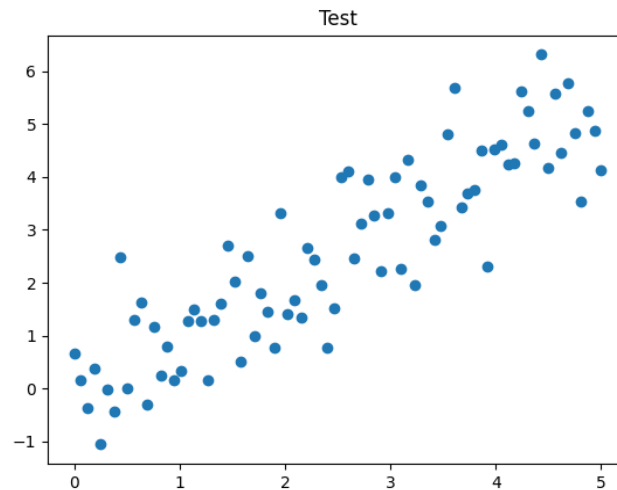


**Conjunto de datos de entrenamiento (320 puntos)**



**Conjunto de datos de validación (80 puntos)**





**Conjunto de datos de test (80 puntos)**

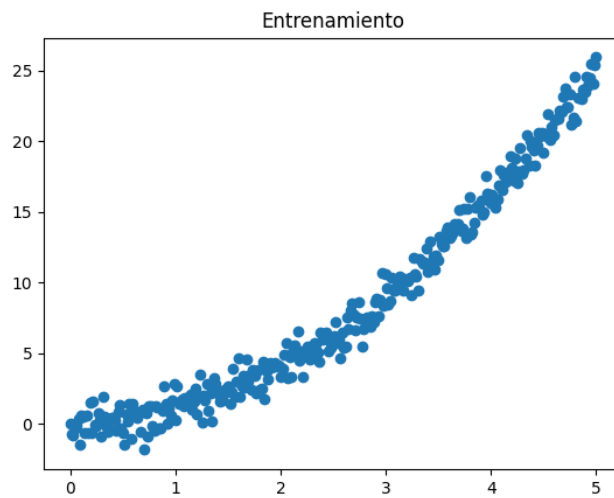
### Regresión 2: Función cuadrática

La salida deseada o “target” está dada por una distribución cuadrática  $f(x) = x^2 + b = t$ , de ordenada b generada aleatoriamente como se muestra a continuación:

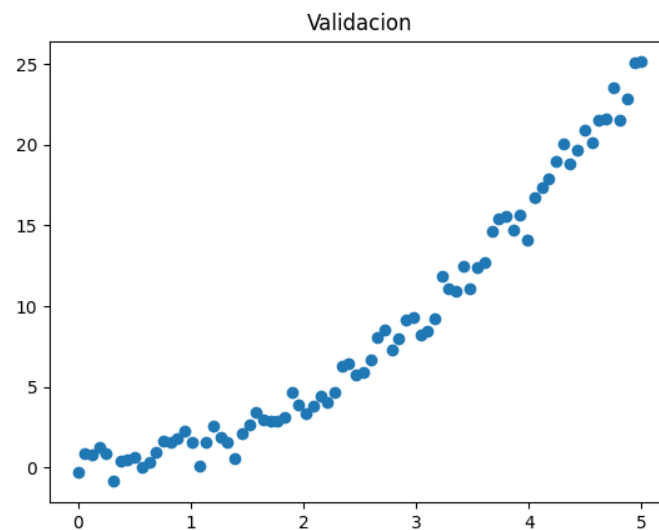
```
AMPLITUD_ALEATORIEDAD = 0.8
# Entradas: 1 col x1
x = np.zeros((cantidad_ejemplos, 1))
t = np.zeros((cantidad_ejemplos, 1))
randomgen = np.random.default_rng()

x1 = np.linspace(0, 5, cantidad_ejemplos)
x2 = np.power(x1, 2) + AMPLITUD_ALEATORIEDAD * randomgen.standard_normal(size=cantidad_ejemplos)
# Generamos un rango con los subindices de cada punto generado.
indices = range(0, cantidad_ejemplos)
# Creamos la matriz de ejemplos de regresion
x1.sort() # Ordenamos en orden creciente las abscisas
x[indices] = np.c_[x1]
t[indices] = np.c_[x2]
```

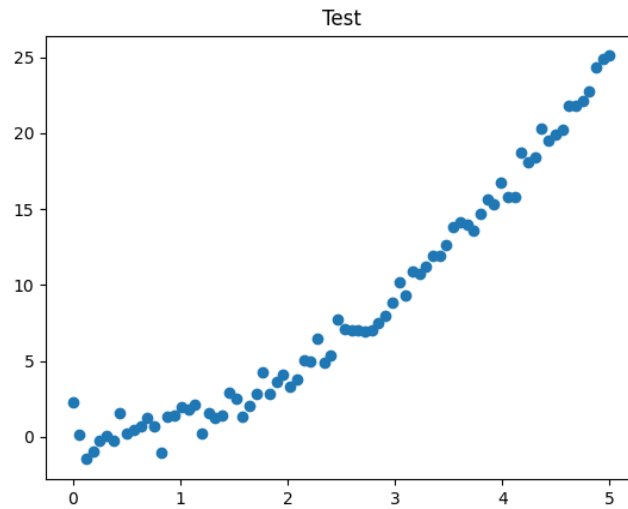
De manera que en una gráfica de “x vs t” se obtiene:



**Conjunto de datos de entrenamiento (320 puntos)**



**Conjunto de datos de validación (80 puntos)**

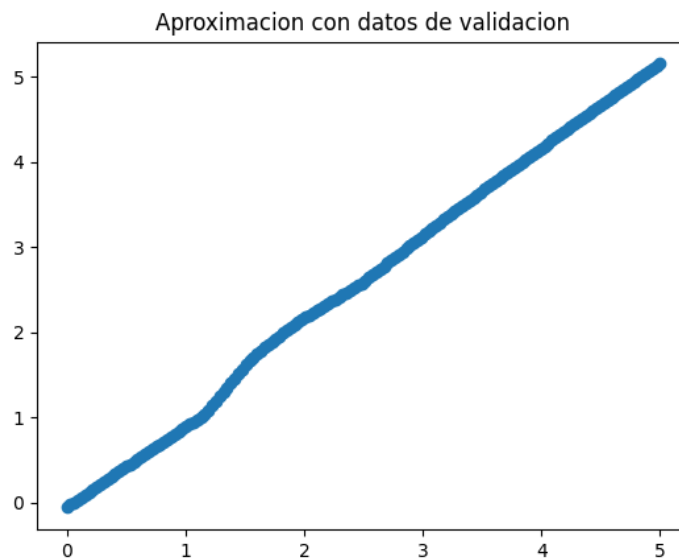


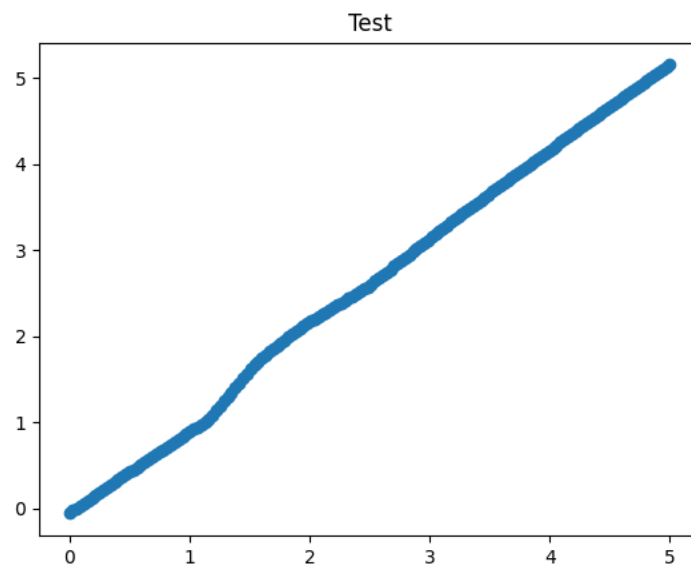
**Conjunto de datos de test (80 puntos)**

**Resultados:**

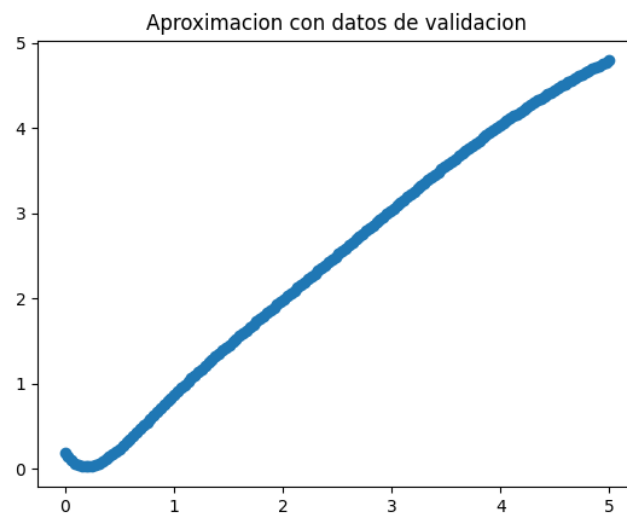
**1. Función lineal:**

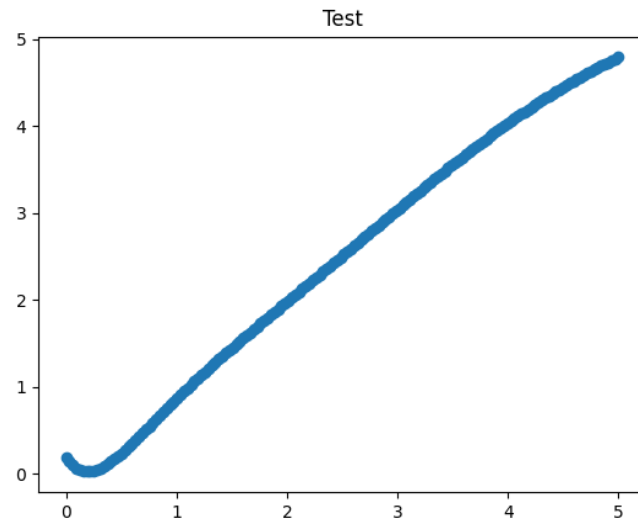
- Función de activación: ReLU
- Cantidad de ejemplos de entrenamiento: 640
- Cantidad de ejemplos de validación: 160
- Cantidad de ejemplos de test: 160





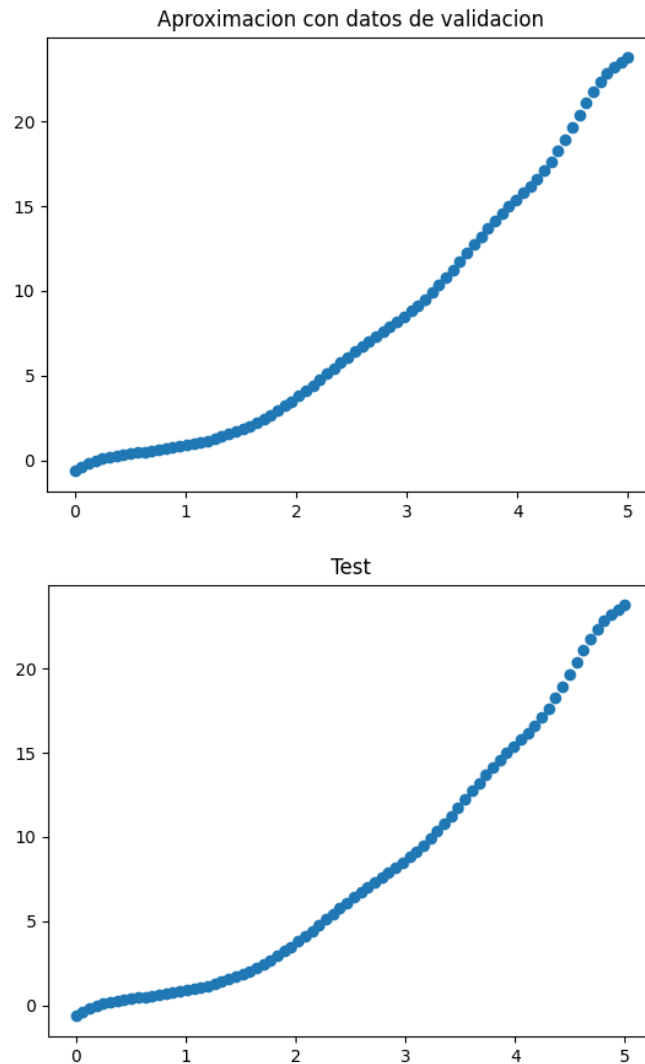
- Aproximación utilizando como función de activación Sigmoide con igual cantidad de ejemplos





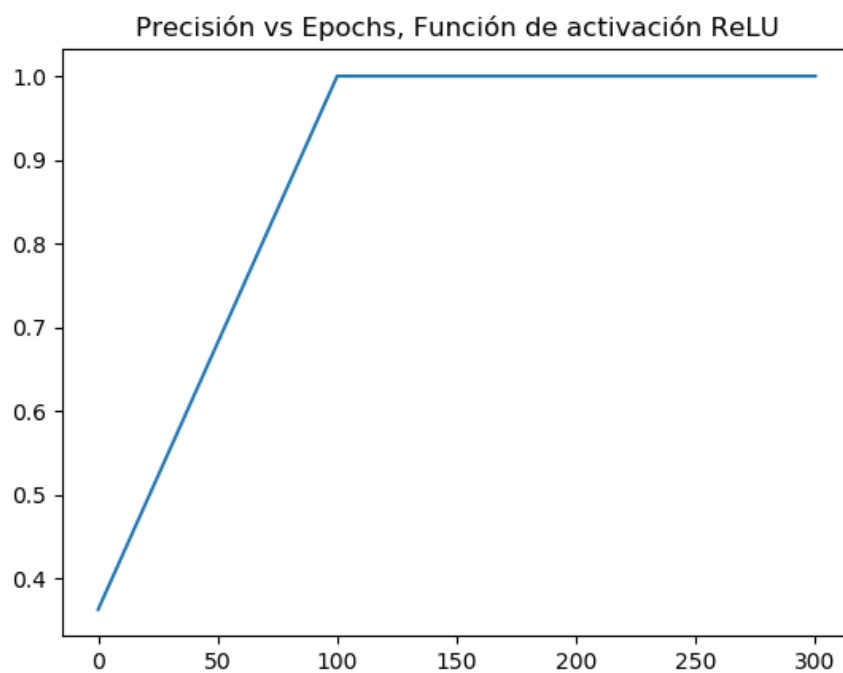
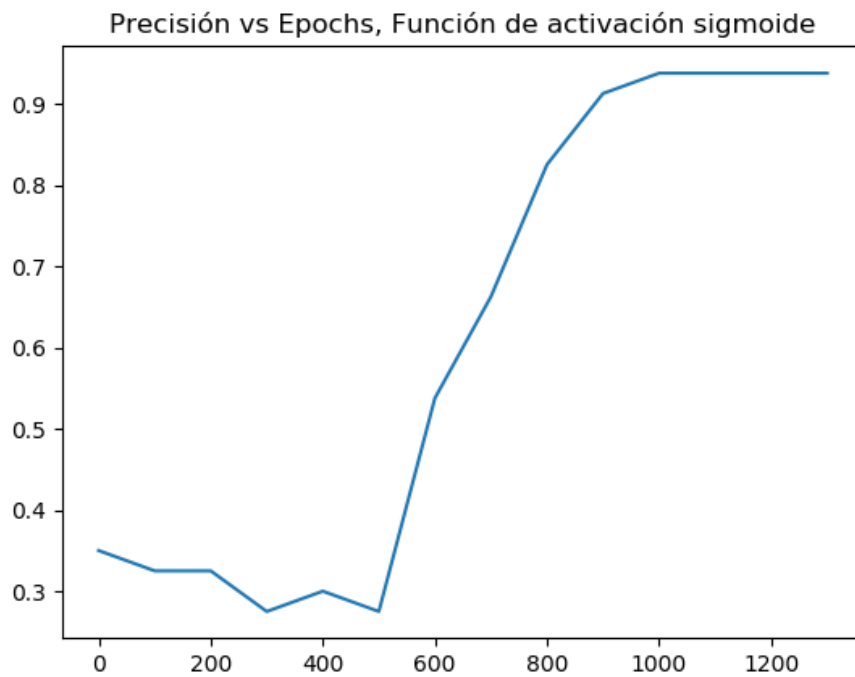
## 2. *Función cuadrática:*

- Función de activación: Sigmoide
- Cantidad de ejemplos de entrenamiento: 320
- Cantidad de ejemplos de validación: 80
- Cantidad de ejemplos de test: 80

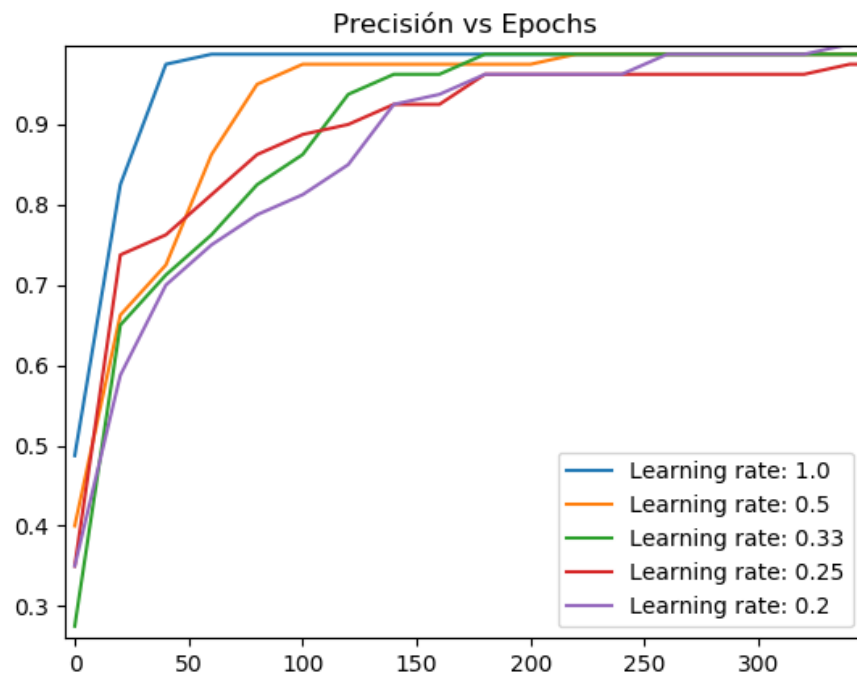


**Ejercicio 6. Realizar un barrido de parámetros (learning rate, cantidad de neuronas en la capa oculta, comparación de ReLU con Sigmoide).**

En primer lugar se experimentó con distintas funciones de activación, comparando entre ReLU y sigmoide. En la primera imagen podemos observar la gráfica de precisión vs Epochs para una función de activación sigmoide, y en la segunda para función de activación ReLU. Como vemos la función ReLU no solo llega a una mejor precisión en menos Epochs, sino que también es menos costosa a nivel computacional, y esto es debido a que no se satura su gradiente en los extremos, como sucede con la función sigmoide.



En la siguiente figura se observa como varía la curva Precisión vs Epochs para distintos Learning Rate, como se observa a medida que se disminuye este parámetro, la pendiente de la curva disminuye y también la rapidez con la que se llega a un resultado óptimo.



Por último, se graficó el cambio de esta curva ante la variación de la cantidad de neuronas, como se aprecia, la precisión aumenta con la cantidad de neuronas.

