UNCUYO
UNIVERSIDAD NACIONAL DE CUYO

FACULTAD DE INGENIERÍA

# Trabajo Práctico N°3:

Machine Learning

## Inteligencia Artificial II

**Profesor:**

Dr. Ing. Martin Marchetta

**Grupo N°1**


**Estudiantes:**

Juan Ignacio Luna

Mario Fernando Bustillo

Rodrigo Aleo

Facultad de Ingeniería, Universidad Nacional de Cuyo


Ciclo Lectivo 2023

UNCUYO
UNIVERSIDAD NACIONAL DE CUYO

FACULTAD DE INGENIERÍA

# Tabla de contenidos

# CONSIGNAS

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

2. Modificar el programa para que:

   a. Mida la precisión de clasificación (accuracy) además del valor de Loss.

   b. 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.

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).

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.

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

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

   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).

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

# RESOLUCIÓN

## 1) Definición de Parámetros

Una vez consolidado el código de la red neuronal feedforward fully connected, se establecieron de forma arbitraria los siguientes parámetros:

- Total de Epochs: 10000
- Cantidad de clases: 3 clases
- Cantidad de ejemplos: 300 ejemplos
- Neuronas de Entrada: 2 neuronas
- Neuronas de Salida: 3 neurona

## 2) Medición de precisión.

Se establecieron de forma arbitraria los siguientes parámetros para calcular la precisión del algoritmo:

- Cada 1000 epoch se calcula la precisión de los pesos calculados obtenidos hasta el momento.

Inicialmente se clasifica los datos de entrada utilizando la red neuronal y se obtiene la clase predicha para cada ejemplo. Luego, se compara cada uno de los datos clasificados con la salida correspondiente(t). En caso de ser iguales, se aumenta la cantidad de aciertos de precisión en 1. Luego se calcula la precisión como la cantidad total de aciertos dividido la cantidad de ejemplos. Una mayor cantidad de aciertos entre la salida esperada y la correspondiente significa una mayor precisión.

En código:

```
def Precision(x, t, pesos, estado, mostrar=True): # Ejercicio 2.a
    cantidad_ejemplos=np.size(x, 0)
    resultados=clasificar(x, pesos)
    precision=0
    for j in range(cantidad_ejemplos):
        if resultados[j]==t[j]:
            precision+=1
    precision=precision/cantidad_ejemplos
    if mostrar:
        text = "Precision " + estado + " " + str(precision)
        print(text)
    return precision
```

**Resultados**

Para un cierto conjunto de datos, cada 1000 epochs calculamos la precisión:

```
Loss epoch 0 : 1.1028058503268876
Precision entrenamiento 0.3333333333333333
Loss epoch 1000 : 0.16736195004350352
Precision entrenamiento 0.9416666666666667
Loss epoch 2000 : 0.1544306123069124
Precision entrenamiento 0.9416666666666667
Loss epoch 3000 : 0.14801073968830594
Precision entrenamiento 0.9416666666666667
Loss epoch 4000 : 0.14354489486238897
Precision entrenamiento 0.9416666666666667
Loss epoch 5000 : 0.1383367982995415
Precision entrenamiento 0.9458333333333333
Loss epoch 6000 : 0.13127132537146213
Precision entrenamiento 0.9458333333333333
Loss epoch 7000 : 0.1291423018693285
Precision entrenamiento 0.9541666666666667
Loss epoch 8000 : 0.12410973635310907
Precision entrenamiento 0.9458333333333333
Loss epoch 9000 : 0.12221763693654894
Precision entrenamiento 0.9541666666666667
```

Luego para evaluar la precisión del entrenamiento con los pesos obtenidos, se generó un conjunto de valores test y se midió la precisión.

```
Precision test 0.9666666666666667
```

## 3) Parada temprana

Para evaluar una parada temprana, se genera una muestra de valores de validación para evaluar las bases calculadas iterativamente. El rendimiento se evalúa cada 500 epoch, para ciertos valores de peso se calcula la precisión (ejercicio 2) y se observa si hay una mejora respecto al último mejor rendimiento multiplicado por un factor de corrección de 0,985 (en caso de que se presenten oscilaciones de mejoras). En caso de no observarse una mejora, se termina el ciclo y se da por finalizado el entrenamiento. Caso contrario el programa se continuará ejecutando hasta obtener un mejor rendimiento o alcanzar el límite total de epochs.

En código:

```
# Ejercicio 3
if i%cant_test == 0:
    precision_validacion = Precision(x_validacion, t_validacion, pesos, "validacion", False)

    if(precision_validacion>precision_val_anterior*0.98):
        precision_val_anterior = precision_validacion
    else:
        print("Fin del entrenamiento")
        break
```

**Resultados**

Para un cierto conjunto de datos, cada 500 epochs calculamos la precisión:

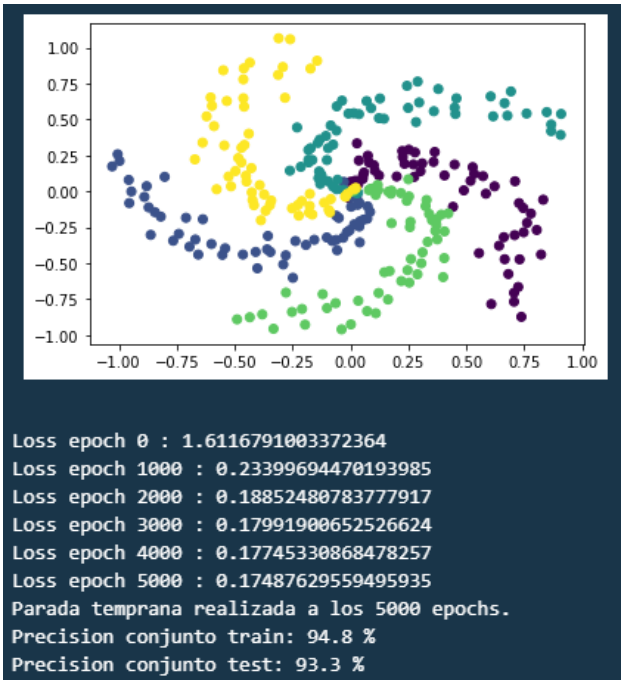
```
Loss epoch 0 : 1.0956475605547733
Precision entrenamiento 0.2916666666666667
Loss epoch 1000 : 0.11561886679839659
Precision entrenamiento 0.9625
Loss epoch 2000 : 0.10507475548715887
Precision entrenamiento 0.9708333333333333
Loss epoch 3000 : 0.09605928677707022
Precision entrenamiento 0.975
Loss epoch 4000 : 0.08191952764284134
Precision entrenamiento 0.975
Loss epoch 5000 : 0.07385189232056548
Precision entrenamiento 0.9833333333333333
Loss epoch 6000 : 0.06713993073880892
Precision entrenamiento 0.9833333333333333
Fin del entrenamiento


Precision test 0.9666666666666667
```

## 4) Experimentar

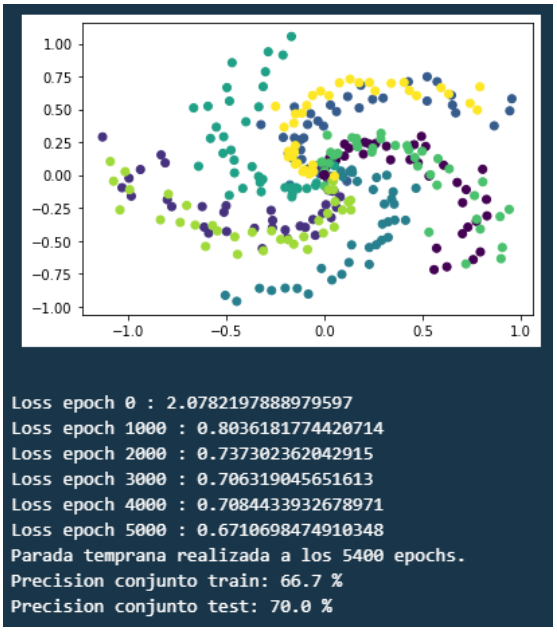
**Aumentando cantidad de clases**

Este ejercicio consiste en variar el código de los ejercicios anteriores generando nuevas clases.

Con 5 clases:



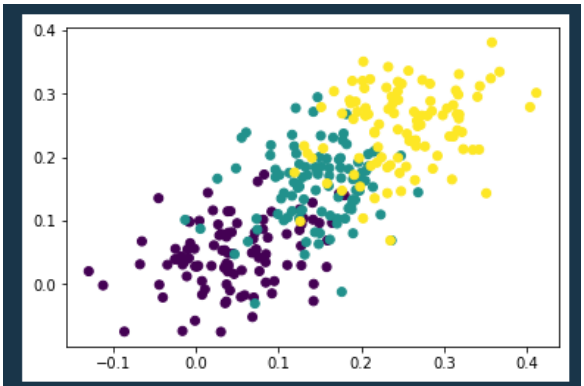

Vemos que la precisión se mantiene relativamente alta, pero mientras más clases generamos, a la red le cuesta más hacer el entrenamiento y la precisión disminuye. Ejemplo con 8 clases:




Para aumentar la precisión habría que probar aumentando la cantidad de neuronas en la capa oculta e incluso aumentar la cantidad de capas. Además sería necesario aumentar la cantidad de puntos de prueba.

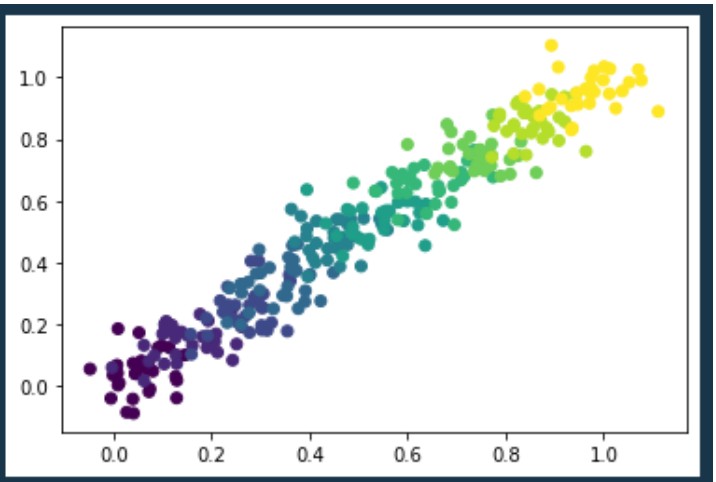
**Generador de datos alternativo**

Se pone a prueba la red neuronal usando otro generador de datos, que crea puntos esparcidos sobre una diagonal y les asigna una clase.
Con 3 clases:



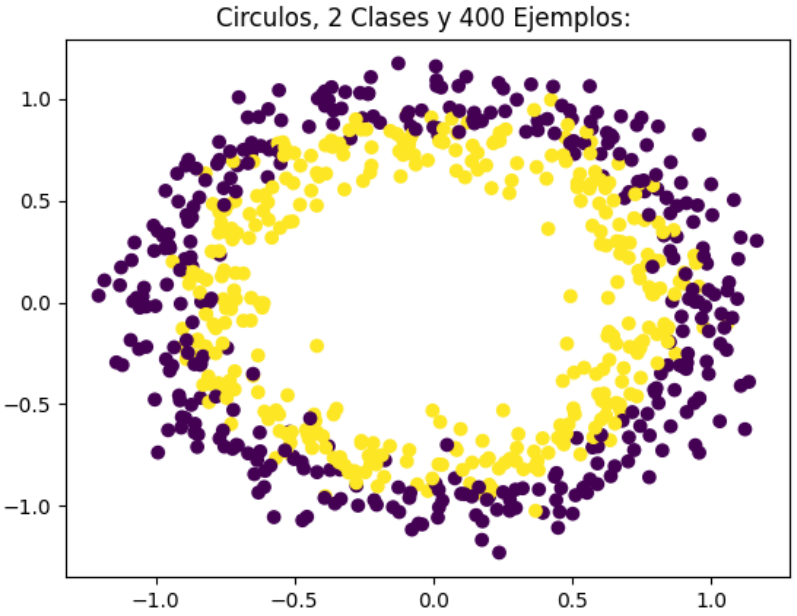

```
Loss epoch 0 : 1.0999634665699431
Loss epoch 1000 : 0.40622207229959933
Loss epoch 2000 : 0.40231246742833054
Loss epoch 3000 : 0.3979811110605633
Parada temprana realizada a los 3700 epochs.
Precision conjunto train: 83.7 %
Precision conjunto test: 76.7 %
```

Con 10 clases:




```
Loss epoch 0 : 2.2938795247657278
Loss epoch 1000 : 0.8886665810995447
Loss epoch 2000 : 0.6543340949397141
Loss epoch 3000 : 0.5930032260152406
Loss epoch 4000 : 0.559106175419126
Loss epoch 5000 : 0.5616367434073141
Parada temprana realizada a los 5900 epochs.
Precision conjunto train: 76.3 %
Precision conjunto test: 70.0 %
```

Por otro lado, se pone a prueba la red neuronal con un conjunto de datos aleatorios alrededor de una circunferencia., con 2 clases y un total de 400 ejemplos.




Para este conjunto de valores se obtuvieron los siguientes resultados:

```
Circulos, 2 Clases y 400 Ejemplos:
Loss epoch 0 : 0.6974860246149719
Precision entrenamiento 0.484375


Loss epoch 1000 : 0.31469937154757616
Precision entrenamiento 0.85625


Fin del entrenamiento
Precision test 0.475
```

Para este generador de puntos, el rendimiento no varía significativamente al aumentar la cantidad de clases. Cabe aclarar que también aumenta el rango de puntos, por lo que en este caso, que haya más clases no indica mayor superposición de datos.

## 5) Modificar para resolver problemas de regresión

La red neuronal hasta ahora trabajada es utilizada para resolver problemas de clasificación donde la salida de la red neuronal es categórica. En un problema de regresión, los valores de salida son contínuos en su dominio.

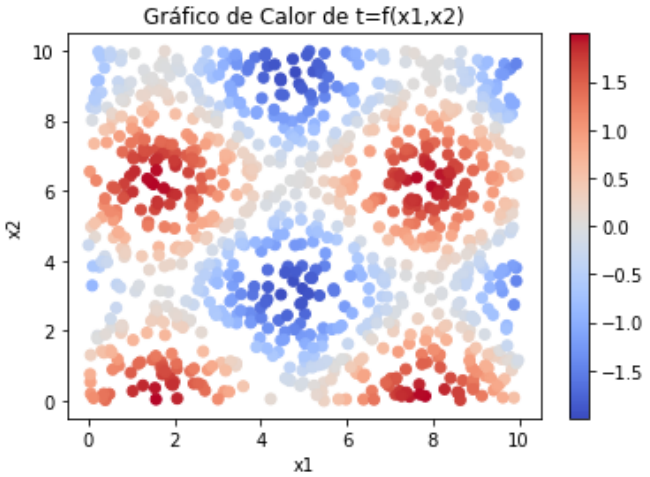
Para realizar esta modificación fue necesario realizar los siguientes pasos:

**Generación de datos continuos**

A pesar de que las entradas no deben ser necesariamente continuas, se decidió que sea así para poder trabajar con una función:

$$t = f(x1, x2) = sin(x1) + cos(x2)$$

Se generaron 1000 puntos en los que x1 y x2 toman un valor al azar entre 0 y 10. Se presenta una gráfica representativa de la muestra:




El objetivo de la nueva red neuronal es que dados un valor de x1 y x2 en este rango, se pueda aproximar el valor que tendría t si se aplicara la ecuación antes mencionada.

**Modificación de función de pérdida**

La función de pérdida usada en la red neuronal de clasificación es la softmax:

$$L = \frac{1}{m} \sum_{m} -log \left( \frac{e^{y_c}{}^{m}}{\sum_j e^{y_j}} \right)$$

Que tiene en consideración si la elección de la clase correcta, mientras que en regresión, en este caso se utilizará el error cuadrático medio (MSE):

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

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

A continuación puede verse su aplicación en el código:

```
loss = (1 / m) * np.sum( np.square( t-y ))
```

Donde t es el valor target e y es la salida de la red neuronal.

**Modificación de las derivadas parciales en Backpropagation**

A pesar de que las derivadas parciales a realizar no cambian, sí cambia la forma de realizar el cálculo analístico de algunas de ellas.

En el caso de $\partial L/\partial Y$, en clasificación se tiene en cuenta la probabilidad de pertenecer a cierto grupo:

$$\frac{\partial L}{\partial Y} = \begin{cases} \frac{1}{m}(P_i - 1) & ; \text{si } y_i \text{ es la clase correcta en el ejemplo } m \\ \frac{1}{m}(P_i) & ; \text{si } y_i \text{ no es la clase correcta en el ejemplo } m \end{cases}$$

En código:

```
dL_dy = p                  # Para todas las salidas, L' = p (la probabilidad)...
dL_dy[range(m), t] -= 1    # ... excepto para la clase correcta
dL_dy /= m
```

Mientras que en regresión, donde se usa el error cuadrático medio:

$$\partial L/\partial Y \; = \; 2/m \; * \; (y - t),$$

Donde m es la cantidad de pares de entradas x1 y x2. Se muestra acontinuación la aplicación en código.

```
dL_dy = 2/m*(y-t)
```

Realizando este cambio, el resto podemos dejarlo igual que antes, que numpy se encargará de realizar las operaciones matriciales correspondientes. Se muestra una imágen con la implementación en código del resto de derivadas parciales y la actualización de los pesos:

```
'''--- EJERCICIO 5: MODIFICACION DE DERIVADAS ---'''
# Ajustamos los pesos: Backpropagation

dL_dy = 2/m*(y-t)


dL_dw2 = h.T.dot(dL_dy)                         # Ajuste para w2
dL_db2 = np.sum(dL_dy, axis=0, keepdims=True)   # Ajuste para b2

dL_dh = dL_dy.dot(w2.T)

dL_dz = dL_dh       # El calculo dL/dz = dL/dh * dh/dz. La funcion "h" es la funcion de activacion de la capa oculta,
dL_dz[z <= 0] = 0   # para la que usamos ReLU. La derivada de la funcion ReLU: 1(z > 0) (0 en otro caso)

dL_dw1 = x.T.dot(dL_dz)                         # Ajuste para w1
dL_db1 = np.sum(dL_dz, axis=0, keepdims=True)   # Ajuste para b1
'''--- END EJERCICIO 5: MODIFICACION DE DERIVADAS ---'''

# Aplicamos el ajuste a los pesos
w1 += -learning_rate * dL_dw1
b1 += -learning_rate * dL_db1
w2 += -learning_rate * dL_dw2
b2 += -learning_rate * dL_db2
```

**Métrica utilizada**

Para poder darle un puntaje al modelado, se utilizó la métrica del coeficiente de determinación (R^2), que es una medida estadística utilizada en para evaluar la bondad de ajuste de un modelo de regresión lineal. Indica que tan bien se ajustan los valores observados a los valores predichos por el modelo. Si el modelo se ajusta de forma perfecta a los datos, tendría un valor máximo de 1. Este valor disminuye hasta -∞ al empeorar el ajuste.

**Resultados**

Realizados estos cambios ya se puede aprovechar la red neuronal para crear resultados. Se ve a continuación como la función Loss va disminuyendo con el transcurso de los 10000 epochs y el r^2 obtenido tanto para el conjunto de train como de test:

```
Loss epoch 0 : 1.1433450140286832
Loss epoch 1000 : 0.7800933288167431
Loss epoch 2000 : 0.680053689171948
Loss epoch 3000 : 0.5935962230216678
Loss epoch 4000 : 0.5447100112958609
Loss epoch 5000 : 0.482015935708015
Loss epoch 6000 : 0.4078559975988334
Loss epoch 7000 : 0.35448944485884926
Loss epoch 8000 : 0.3207038089214781
Loss epoch 9000 : 0.2900231013921453
El r^2 de train es 0.7323385061595704
El r^2 de test es 0.7215499830539074
```

Podemos ver que el loss sigue en descenso, por lo que se probó aumentar aumentar la cantidad de epochs a 100000 para ver qué nuevos resultados se obtenían:

```
Loss epoch 98000 : 0.02236541236093595
Loss epoch 99000 : 0.019701771370444057
El r^2 de train es 0.9791313791492093
El r^2 de test es 0.9768925947005265
```

Ahora el r^2 en ambos conjuntos es de aproximadamente 1 mientras que el loss es próximo a 0. Lo cual indica que la red neural es muy eficaz a la hora de aproximar la función t.

Cabe destacar que el tiempo de entrenamiento aumento de 14 s para 10000 epochs, a 2 minutos para 100000 epochs.

## 6) Realizar barrido de hiper parámetros

**HIper parámetros a analizar**

Los hiper parámetros a encontrar para este ejercicio son:

- Learning Rate.
- Cantidad de neuronas en la capa oculta.
- Cantidad de epochs.
- Función de activación Relu o Sigmoide.

**Creación de vectores y posibles combinaciones**

Para realizar esta tarea, se procedió a realizar vectores que contengan valores en un rango definido para cada variable. Se muestra a continuación un ejemplo de la definicion del valor mínimo, valor máximo y paso para cada vector:

```python
rango_learning_rate = (0.01, 0.1, 0.01)

rango_cant_neuronas = (100, 500, 100)

rango_epochs = (10000, 20000, 5000)

funciones_activacion = ('relu', 'sigmoide')
```

Se realizó tanto un barrido matricial, donde los vectores se crean desde el valor mínimo hasta el máximo utilizando el paso proporcionado; como también se utilizó el barrido aleatorio, donde se crea cierta cantidad de puntos aleatorios entre los valores mínimos y máximo.

```
if barrido=='matricial':
    vect_lr = np.arange(min_lr,
                        max_lr+paso_lr,
                        paso_lr)

    vect_cant_neuronas = np.arange(min_cn,
                                   max_cn+paso_cn,
                                   paso_cn)

    vect_epochs = np.arange(min_e,
                            max_e+paso_e,
                            paso_e)

elif barrido=='random':
    n = int ((max_lr-min_lr)/paso_lr)
    vect_lr = np.random.uniform(min_lr,
                                max_lr,
                                n)

    n = int ((max_cn-min_cn)/paso_cn)
    vect_cant_neuronas = np.random.uniform(min_cn,
                                           max_cn,
                                           n)

    n = int ((max_e-min_e)/paso_e)
    vect_epochs = np.random.uniform(min_e,
                                    max_e,
                                    n)
```

Una vez creado los vectores, se realiza la combinatoria entre ellos, logrando una matriz, donde cada fila es una combinación de los diferentes hiperparámetros.

Por ejemplo, usando la penúltima imágen y un barrido matricial, Learning Rate tendrá 10 elementos, cantidad de neuronas tendrá 5 elementos y cantidad de epochs tendrá 3, por lo tanto se tendrán 3x10x5 = 150 combinaciones como puede confirmarse a continuación:

```
Cantidad de combinaciones: 150
```

**K-fold Cross Validation**

A la hora de probar cada una de estas combinaciones, se realiza la técnica de K-fold Cross Validation, que implica separar al conjunto de train en varios subconjuntos o *folds*. En este caso se optó por realizar 10 divisiones. Con 9 de ellos se realiza el entrenamiento de la red neuronal y con el sobrante, se realiza un test de validación. Posteriormente, se realiza el mismo proceso pero rotando cual será el subconjunto sobre el cuál se realizará la validación.

De esta forma podemos obtener una muestra representativa de qué tan bien funcionan la combinación de hiperparámetros y evitamos ser engañados por casos aislados donde la red neuronal se entrena excepcionalmente bien (o mal).


En este caso el conjunto de test no es utilizado aún.

**Evaluación**

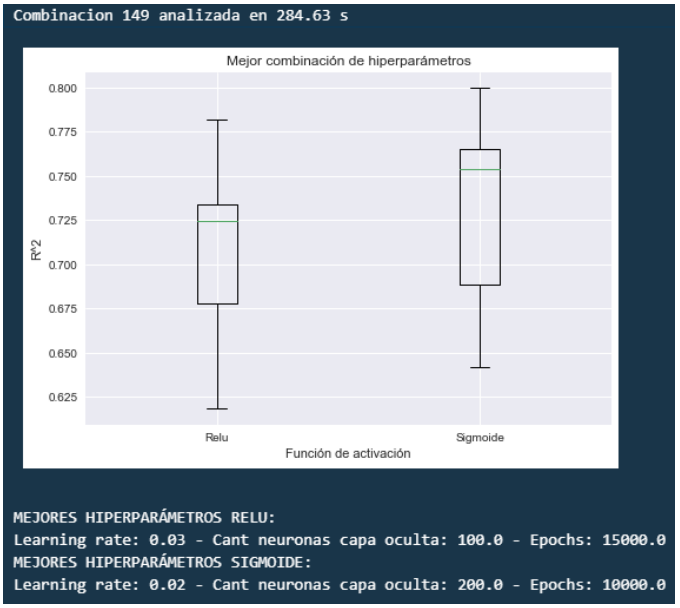
Como método para puntuar al modelo se usa el R^2, como se realiza 10 veces el entrenamiento para cada conjunto de hiper parámetros, para darle una puntuación final se utiliza el promedio del R^2 obtenido entre los 10. Se elije para analizar el conjunto que tenga el promedio más alto. El análisis se realizó por separado para la función sigmoide y relu para poder comparar luego.

**Resultados**

Realizando el análisis sobre los siguientes rangos y pasos:

```
#(valor_min, valor_max, paso)

rango_learning_rate = (0.01, 0.1, 0.01)

rango_cant_neuronas = (100, 500, 100)

rango_epochs = (10000, 20000, 5000)
```

Con barrido matricial, y luego de más de 11 horas de ejecución, se obtuvieron los siguientes resultados:




Al utilizar el modelo sobre el conjunto de test, para la función Relu, se obtienen los siguientes resultados:

```
El r^2 de train es 0.7818585481061727
El r^2 de test es 0.8133553604457808
```

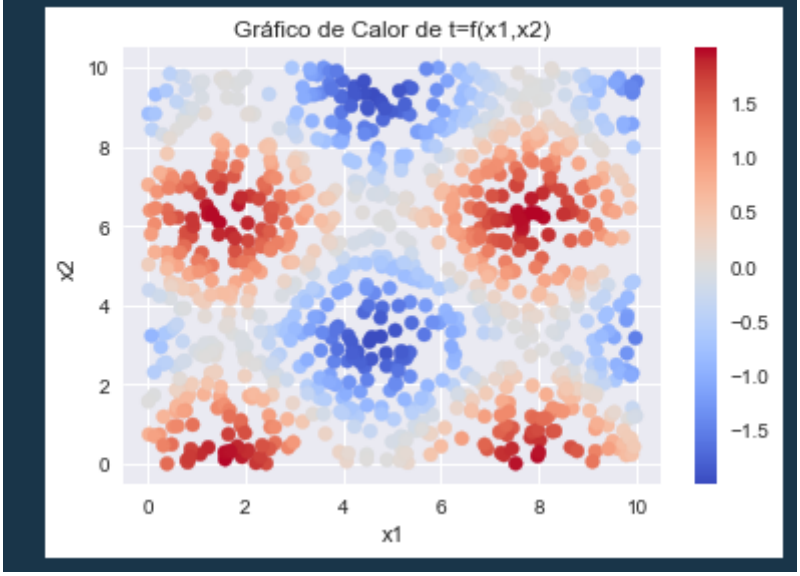
Que es próximo a los valores obtenidos. Es importante que en la gráfico de cajas la varianza no sea mucha, para asegurar de que al entrenar una red neuronal, los resultados obtenidos luego no tengan mucha variación.

Lo ideal es probar un rango más amplio de hiperparámetros con un paso menor. No se realizó dicha tarea en este trabajo porque es un proceso muy lento que requiere mucho tiempo de computación.

## 11) Implementar una red neuronal Feedforward con Keras [OPCIONAL]

Para ello se utilizó la librería Keras de Tensorflow. Primero se procedió a realizar un conjunto de datos similar al de regresión de los ejercicios anteriores:

```
✓ x, t = generar_datos_regresion(1000) ···
```




Se realiza la separación del conjunto train y test usando librerías de SkLearn, usando una proporción de 90-10, respectivamente:

```
x_train, x_test, t_train, t_test = train_test_split(x, t, test_size=0.1)
```

Luego se define la arquitectura que tendrá nuestra red neuronal:

```
model = keras.Sequential([
    keras.layers.Dense(100, input_shape=(2,), activation='relu'),  # Capa oculta con 100 neuronas
    keras.layers.Dense(1)  # Capa de salida con 1 unidad
])
```

Además, se compila el modelo y se define la función de loss y el optimizador que utilizará.

```
model.compile(loss='mean_squared_error', optimizer='adam')
```

Adam (Adaptive Moment Estimation), es un algoritmo de optimización que calcula las tasas de aprendizaje individuales de cada parámetro en función de su historial de gradientes y momentos anteriores, lo que le permite ajustar automáticamente las tasas de aprendizaje según la magnitud y la dirección de los gradientes. Esto genera una convergencia más rápida y eficiente del modelo.

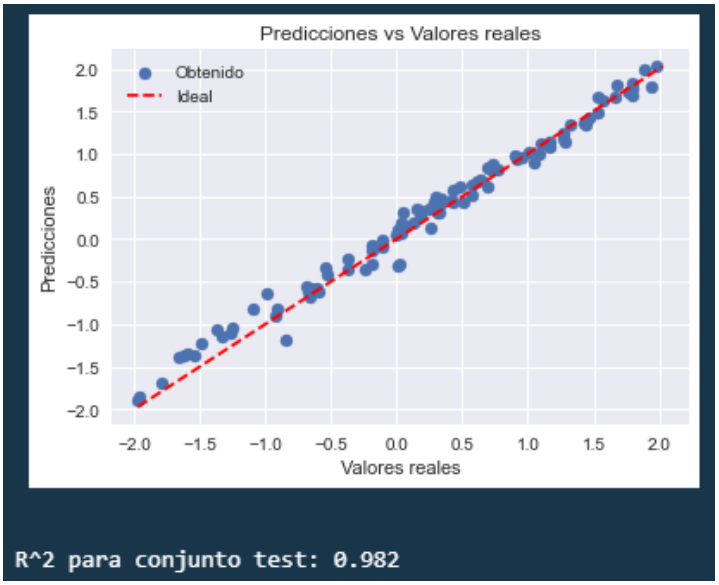
Se procede a realizar el entrenamiento del modelo usando el conjunto de test, lo cual demora 53 s para los hiperparámetros escogidos :

```
1  model.fit(x_train, t_train, epochs=100, batch_size=1)
✓ 53.4s
```

El loss obtenido en el último epoch es:

```
Epoch 100/100
900/900 [==============================] - 0s 492us/step - loss: 0.0411
```

Es un valor bajo que indica un buen entrenamiento del modelo. Calculando el R^2 del modelo para el conjunto de train y graficando los valores obtenidos contra los esperados, se tienen los siguientes resultados:




Si el modelo fuera perfecto, todos los puntos se ubicarían sobre la línea diagonal. El R^2 es próximo a 1 lo que indica que la red neuronal ha sido bien entrenada.

**Comparación resultados red Keras vs red casera**

Con las dos redes neuronales implementadas se pueden obtener buenos resultados, sin embargo se puede apreciar cómo la de Keras está mucho mejor optimizada ya que conseguir dichos resultados demoró tan sólo 53 s, mientras que obteniendo resultados similares con la casera (al usar 100000 epochs), se demoró unos 2 minutos. Y esta diferencia aumentaría al tratar con conjuntos de datos más grandes.