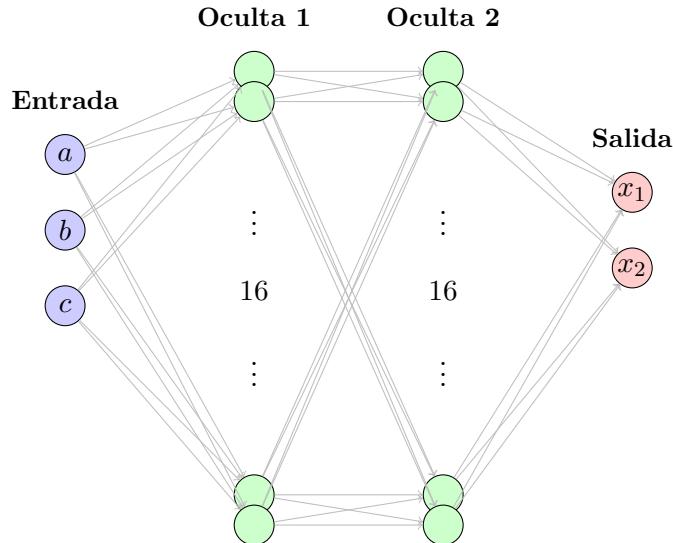


## Práctica: Ecuaciones de segundo grado

En esta práctica trataremos de crear un modelo para calcular ecuaciones de segundo grado. Tras crearlo lo analizaremos y realizaremos modificaciones. Para las Capas ocultas necesitamos profundidad. Usaremos dos capas de 16 neuronas de ancho. Más adelante probaremos a modificar este diseño y analizaremos los resultados. Por ahora la estructura será:



La generación de los datos es el primer desafío. Si generamos  $a,b,c$  al azar, muchas ecuaciones tendrán soluciones complejas ( $b^2 - 4ac < 0$ ). PyTorch estándar no maneja números complejos de forma nativa para entrenamiento simple. Para evitar este problema lo haremos al revés, fijaremos las soluciones de la ecuación  $x_1$  y  $x_2$  y calcularemos  $a,b,c$  a partir de estas. Para ello podemos usar las fórmulas de Vieta. De donde extraemos:

$$a = 1 \quad b = -(x_1 + x_2) \quad c = x_1 \cdot x_2$$

También acotaremos el rango de nuestras salidas (en este caso entre -10 y 10) y por último ordenaremos estas. Si para la entrada (1,-5,6) a veces le dices que la respuesta es (2,3) y otras veces (3,2), la red se confundirá y el error (Loss) nunca bajará de cierto punto. Para evitar esto siempre ordenamos las salidas ( $y$ ). Por ejemplo, haz que la salida 1 sea siempre la raíz más pequeña y la salida 2 la más grande. Esto facilita enormemente el aprendizaje porque la red detecta un patrón claro.

```
def generar_datos(n_muestras=500):
    x_reales = np.random.uniform(-10, 10, (n_muestras, 2))
    inputs = []
    targets = []
    for i in range(n_muestras):
        x1, x2 = x_reales[i]
        a = 1.0
        b = -(x1 + x2)
        c = x1 * x2
        inputs.append([a, b, c])
        targets.append(sorted([x1, x2]))

    return torch.tensor(inputs, dtype=torch.float32),
            torch.tensor(targets, dtype=torch.float32)
```

En relación al cálculo de error, como es un problema de predecir valores numéricos exactos (regresión), la función de pérdida ideal es el MSELoss (Mean Squared Error). Tenemos que tener cuidado

con la tasa de aprendizaje, sobre todo si empezamos con un 0.5 como en casos anteriores. Un 0.5 es extremadamente alto para este tipo de problema. Imagina que el optimizador está intentando bajar a un pozo (el mínimo error), pero da saltos tan gigantes que se pasa de largo y termina en la montaña de enfrente. Terminaremos obteniendo infinitos o `nan` como resultado del error. Por lo tanto, la tasa de aprendizaje debe ser baja de inicio, por ahora la hemos dejado en 0,01. Aquí mostramos el algoritmo inicial y mostramos la salida ofrecida.

```

import torch
import numpy as np
import torch.nn as nn
import torch.optim as optim

class RedEcuaciones(nn.Module):
    def __init__(self):
        super(RedEcuaciones, self).__init__()

        self.hidden1 = nn.Linear(3, 16) # Capa de entrada (3) a primera capa oculta (16)
        self.hidden2 = nn.Linear(16, 16) # Segunda capa oculta (16 a 16)
        self.output = nn.Linear(16, 2) # Capa de salida (16 a 2 soluciones)
        self.relu = nn.ReLU()         # Func. de activacion

    def forward(self, x):
        x = self.relu(self.hidden1(x))
        x = self.relu(self.hidden2(x))
        x = self.output(x)
        return x

X, y = generar_datos(500)

model = RedEcuaciones()
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

epochs=8000
print("Entrenando ...")

for epoch in range(epochs):
    outputs = model(X)
    loss = criterion(outputs, y) # Error

    optimizer.zero_grad()
    loss.backward()             # Backpropagation
    optimizer.step()            # Ajuste pesos automatico (de eso se encarga el optimizador)

    if (epoch + 1) % 500 == 0:
        print(f"Epoca {epoch+1}/{epochs} - Error: {loss.item():.4f}")

```

```

Epoca 500/8000 - Error: 0.0718
Epoca 1000/8000 - Error: 0.0477
....
Epoca 8000/8000 - Error: 0.0172

```

El error nunca bajará al 0 como lo haría un cálculo matemático. La fórmula de la ecuación de segundo grado tiene una raíz cuadrada  $\sqrt{(b - 4ac)}$ . Las funciones de activación como ReLU son "pedazos" de líneas rectas. Imagina que intentas dibujar un círculo perfecto usando solo reglas cortas. Por más reglas que uses, si te acercas mucho, verás pequeños ángulos. Pues eso mismo está haciendo la

red, aproximando una curva compleja con pequeños segmentos lineales.

En teoria cualquier red neuronal pude aproximar cualquier función pero no es fácil. Con 16 neuronas, tienes un número limitado de «reglas» para dibujar esa curva. Si subieras a 64 o 128 neuronas, verías que el error baja a 0.0050, pero el modelo sería más pesado. Además de todo esto tenemos un sesgo de los datos. Si tus valores de  $x$  van de -10 a 10, pero la mayoría de tus ejemplos en el dataset de 500 muestras están cerca del 0, la red será experta en números pequeños y torpe en los grandes. A esto se le llama distribución de los datos y en este caso está muy sesgada para ese marco de resultados de  $x_1$  y  $x_2$ .

Hemos implementado tan solo uno de los cambios mencionados arriba, incrementando la anchura de las capas ocultas de 16 neuronas a 128 y los resultados han sido una disminución del error considerable. El tiempo de entrenamiento para la misma cantidad de muestras ha pasado de 5 segundos a 23 segundos, sin embargo<sup>1</sup>.

```
Entrenando ...
Epoca 500/8000 - Error: 0.0483
Epoca 1000/8000 - Error: 0.0285
....
....
Epoca 7500/8000 - Error: 0.0065
Epoca 8000/8000 - Error: 0.0045
```

---

<sup>1</sup>Probado en Google Colab