Universidad Central de Chile
Facultad de Ingeniería y Arquitectura
Escuela de Ingeniería

Creación desde cero

de Neurona artificial para

compuertas lógicas AND y OR

Nombre de los estudiantes:

- Felipe Mancilla Yañez

- Simón Henríquez Lind

Profesor: Hernán Olmi Reyes

Fecha: 12-06-2023

Introducción

Creación de una neurona que se comporte como una compuerta lógica binaria “AND” u “OR”, es decir, esta neurona podrá recibir 2 entradas (además del “bias”) y dependiendo de los valores esperados otorgados en su entrenamiento, funcionara como una o la otra. Por otro lado, el entrenamiento de la neurona será realizado con el método “BackPropagation”, método que modifica los pesos en la dirección en que el error entre el valor obtenido y el deseado disminuya.

Para la creación de la neurona se utilizó el lenguaje de programación “Python” junto a la librería “random” y su funcionalidad “uniform” con la cuál se logra inicializar la neurona con pesos aleatorios. Esta neurona se apoya de la clase “BackPropagation” para poder entrenarse. La función de activación de la neurona es de tipo escalón y debe tenerse claro que al final esta será; 1 si “suma_ponderada” > “bias” * -1.

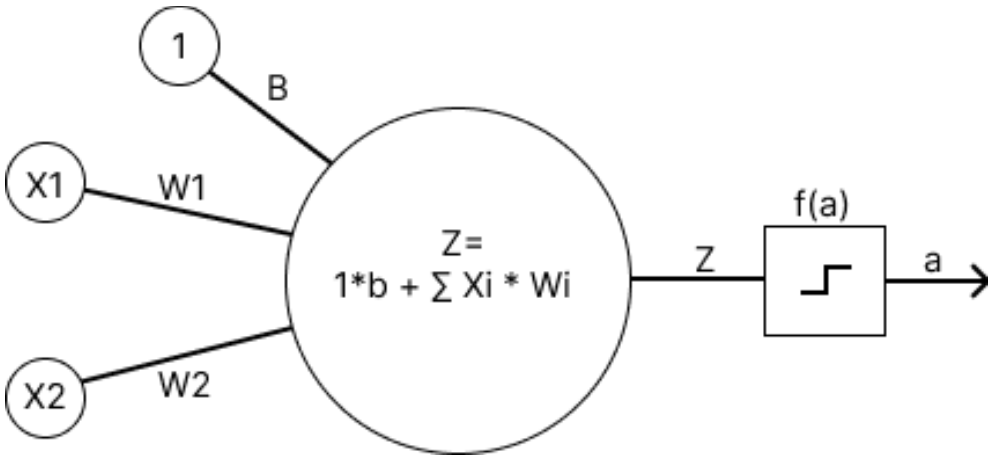

Figura 1. Visualización de la Neurona, imagen usada en la interfaz

Enlace al repositorio de la aplicación, descargar código y foto, guardar en una misma carpeta y ejecutar.
https://github.com/Simon-Henriquez/Neurona-BackPropagation

Desarrollo

```python
class Neurona:
    """Simple Neurona"""
    def __init__(self, nombre: str = "", rango_aleatorio: tuple[int] = (-1, 1), cantidad_entradas: int = 2,
                 cte_aprendizaje: float = 1/10):
        self.nombre = nombre
        self.cantidad_entradas = cantidad_entradas
        self.estado = "Aprendizaje no terminado"
        self.bias = round(uniform(rango_aleatorio[0], rango_aleatorio[1]), 3)
        self.pesos = [round(uniform(rango_aleatorio[0], rango_aleatorio[1]), 3) for _ in range(cantidad_entradas)]
        self.cte_aprendizaje = cte_aprendizaje

    def suma_ponderada(self, *entradas: int) -> float:
        """Retorna la suma de cada entrada multiplicada por su peso.\n
        Entrada del bias considerada cte = 1"""
        z = 0
        for i, val in enumerate(entradas):
            z += self.pesos[i] * val
        return round(z + self.bias, 3)

    def activacion(self, z: float) -> int:
        """Evalúa el valor de la suma ponderada en una función de activación 'escalon'.
        Activación dinámica dependiendo del valor del bias."""
        if z >= -self.bias:
            return 1
        return 0

    def calcular_error(self, valor_deseado: int, valor_obtenido: int) -> int:
        """Resta entre el valor deseado con el valor obtenido."""
        return valor_deseado - valor_obtenido

    def actualizar_pesos(self, *entradas: int, error: float) -> None:
        """Actualiza los pesos y el bias cuando el error es distinto de cero."""
        if error == 0:
            return 0
        for i, val in enumerate(entradas):
            self.pesos[i] = round(self.pesos[i] + (self.cte_aprendizaje * error * val), 3)
        self.bias = round(self.cte_aprendizaje * error, 3)
        return 1

    def resultado(self, entradas: List[int]) -> List[int]:
        """Retorna los resultados para cierta entrada de valores usando el bias y pesos actuales."""
        result = []
        for i in range(0, len(entradas), 2):
            x1 = entradas[i]
            x2 = entradas[i+1]
            z = self.suma_ponderada(x1, x2)
            a = self.activacion(z)
            result.append(a)
            print(f"Resultado para [{x1}, {x2}]: {a}")
        return result
```

Figura 2. Clase Neurona

```python
class BackPropagation:
    """Encargado de generar epocas hasta optimizar los pesos."""
    @staticmethod
    def descenso_gradiente(neurona: Neurona, entradas: List[int]) -> int:
        """Itera sobre la lista de entradas con sus respectivos\n
        valores esperados, indefinidamente hasta optimizar pesos."""
        cont = 0
        for i in range(0, len(entradas), neurona.cantidad_entradas+1):
            x1 = entradas[i]
            x2 =  entradas[i+1]
            z = neurona.suma_ponderada(x1, x2)
            a = neurona.activacion(z)
            y = entradas[i+2]
            e = neurona.calcular_error(y, a)
            if cont == 0:
                cont = neurona.actualizar_pesos(x1, x2, error=e)
        return cont

    @staticmethod
    def optimizar(neurona: Neurona, entradas: List[int], queue: Queue = False) -> None:
        """Calcula nuevos pesos hasta obtener los optimos.\n
        La queue es para fines visuales con la interfaz"""
        exitoso = 1
        while exitoso != 0:
            exitoso = BackPropagation.descenso_gradiente(neurona, entradas)
            if queue:
                queue.put((neurona.pesos[0], neurona.pesos[1], neurona.bias))
            sleep(0.2)
        neurona.estado = "Aprendizaje exitoso"
        if queue:
            queue.put(None)
        print("Optimizacion finalizada")
```

Figura 3. Clase BackPropagation

```python
def main():
    neurona_and = Neurona(cte_aprendizaje=1/10)
    datos_and = [0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1]
    print(neurona_and)
    BackPropagation.optimizar(neurona_and, datos_and)
    print(neurona_and)
    sleep(2)
    prueba_and = [0, 0, 0, 1, 1, 0, 1, 1]
    neurona_and.resultado(prueba_and)
    print("\n")
    neurona_or = Neurona(cte_aprendizaje=1/10)
    datos_or = [0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1]
    print(neurona_and)
    BackPropagation.optimizar(neurona_or, datos_or)
    print(neurona_or)
    sleep(4)
    prueba_or = [0, 0, 0, 1, 1, 0, 1, 1]
    neurona_or.resultado(prueba_or)
```

Figura 4. Ejecución del programa

```
<__main__.Neurona object at 0x00000212B810AD60>
W1: 0.128  W2: -0.058  B: -0.1
W1: 0.228  W2: 0.042  B: 0.1
W1: 0.228  W2: 0.042  B: -0.1
W1: 0.128  W2: 0.042  B: -0.1
W1: 0.228  W2: 0.142  B: 0.1
W1: 0.228  W2: 0.142  B: -0.1
W1: 0.128  W2: 0.142  B: -0.1
W1: 0.128  W2: 0.142  B: -0.1
Optimizacion finalizada
<__main__.Neurona object at 0x00000212B810AD60>
Resultado para [0, 0]: 0
Resultado para [0, 1]: 0
Resultado para [1, 0]: 0
Resultado para [1, 1]: 1
```

Figura 5. Neurona AND funcionando

```
<__main__.Neurona object at 0x00000212B810AD60>
W1: 0.165  W2: -0.827  B: 0.1
W1: 0.165  W2: -0.827  B: -0.1
W1: 0.165  W2: -0.727  B: 0.1
W1: 0.165  W2: -0.727  B: -0.1
W1: 0.165  W2: -0.627  B: 0.1
W1: 0.165  W2: -0.627  B: -0.1
W1: 0.165  W2: -0.527  B: 0.1
W1: 0.165  W2: -0.527  B: -0.1
W1: 0.165  W2: -0.427  B: 0.1
W1: 0.165  W2: -0.427  B: -0.1
W1: 0.165  W2: -0.327  B: 0.1
W1: 0.165  W2: -0.327  B: -0.1
W1: 0.165  W2: -0.227  B: 0.1
W1: 0.165  W2: -0.227  B: -0.1
W1: 0.165  W2: -0.127  B: 0.1
W1: 0.165  W2: -0.127  B: -0.1
W1: 0.165  W2: -0.027  B: 0.1
W1: 0.165  W2: -0.027  B: -0.1
W1: 0.165  W2: 0.073  B: 0.1
W1: 0.165  W2: 0.073  B: -0.1
W1: 0.165  W2: 0.173  B: 0.1
W1: 0.165  W2: 0.173  B: -0.1
W1: 0.165  W2: 0.273  B: 0.1
W1: 0.165  W2: 0.273  B: -0.1
W1: 0.265  W2: 0.273  B: 0.1
W1: 0.265  W2: 0.273  B: -0.1
W1: 0.265  W2: 0.273  B: -0.1
Optimizacion finalizada
```

Figura 6. Neurona OR funcionando