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: 14-06-2023

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 cual 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.

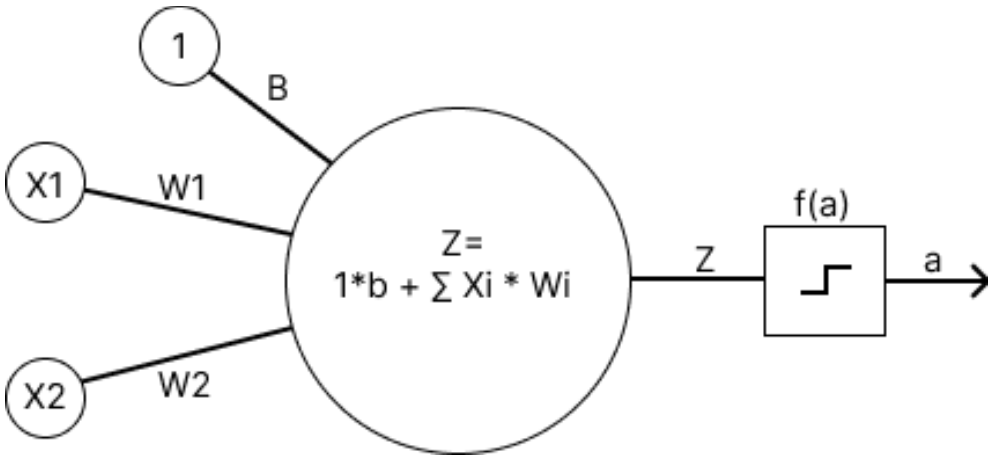


Figura 1. Visualización de la Neurona, imagen usada en la interfaz

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

Para la creación y entrenamiento de la neurona se utilizó el algoritmo de retro-propagación, por lo cual se crearon las clases Neurona y BackPropagation con los siguientes atributos y métodos:

Neurona

- *nombre*
- *cantidad_entradas*
- *estado*
- *bias*
- *pesos*
- *cte_aprendizaje*

Métodos de Neurona

- *suma_ponderada*: Recibe una lista de entradas que serán multiplicadas con los pesos correspondientes de la neurona, con estos resultados se obtiene una suma ponderada y posteriormente se le suma el valor del peso del bias.

$$z = w1 * x1 + w2 * x2 + b$$

- *activacion:* A partir del valor entregado por suma_ponderada, se calcula la función de activación de escalón unitario dado por

$$f(z) = \begin{cases} 1, & z \geq -bias \\ 0, & C.O.C \end{cases}$$

- *actualizar_pesos*: Modifica los pesos de las entradas de la neurona (peso actual + peso actualizador).

$$w_{i+1} = w_i + \delta w_i$$

- *front_propagation*: Propaga los datos por la neurona obteniendo el valor de salida.
- Resultado: Retorna los resultados para cierta entrada de valores usando el bias y pesos actuales.

Métodos de BackPropagation

- *descenso_gradiente*: Objetivo de actualizar los pesos de la neurona en la dirección en que el error disminuya, si no existió error retorna 0.
- *nuevos_pesos*: Obtiene los pesos actualizadores para la neurona.

$$\delta w_i = \eta * e * x_i$$

Donde:

$w_i = peso\ actual\ de\ cada\ entrada$

$\delta w_i = peso\ actualizador\ de\ cada\ entrada$

$w_{i+1} = peso\ posterior\ a\ la\ actualización\ de\ cada\ entrada$

$\eta = tasa\ de\ aprendizaje$

$e = error\ entre\ el\ valor\ deseado\ y\ el\ valor\ de\ salida\ de\ la\ función\ de\ activación$

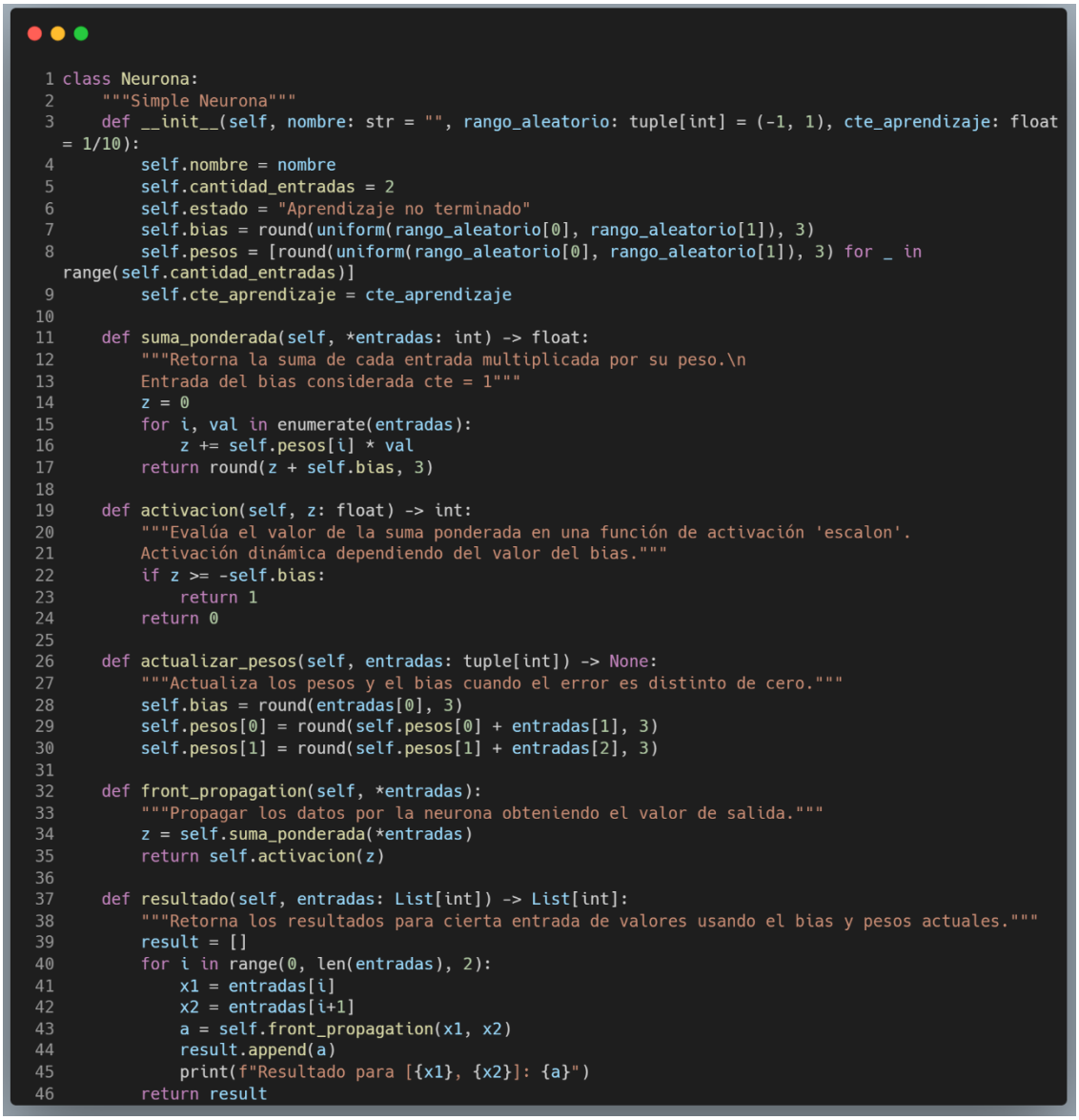
$x_i = valor\ de\ entrada\ de\ cada\ entrada$

Mientras que para el sesgo se utiliza la siguiente formula:

$$b_{i+1} = 1 * b_i + \eta * e$$

- *calcular_error*: Resta entre el valor deseado con el valor obtenido
- *epocar_hasta_optimizar*: Realiza un descenso gradiente una y otra vez hasta lograr obtener los pesos adecuados, es decir, es el corazón de la retro-propagación hacia atrás.

Para probar el código se ejecuta la función "*main*", esta crea una neurona AND y otra OR, inicialmente solo contienen el nombre, pero luego se les entrena con las mismas entradas binarias, pero distintos valores deseados, utilizando el método "*epocar_hacia_atras*" hasta convertirlas en unas verdaderas compuertas lógicas AND y OR. Finalmente, para comprobar si el entrenamiento realmente fue exitoso se llama al método "resultado" para que nos devuelva el valor de salida para las entradas binarias.



```python
class Neurona:
    """Simple Neurona"""
    def __init__(self, nombre: str = "", rango_aleatorio: tuple[int] = (-1, 1), cte_aprendizaje: float = 1/10):
        self.nombre = nombre
        self.cantidad_entradas = 2
        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(self.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 actualizar_pesos(self, entradas: tuple[int]) -> None:
        """Actualiza los pesos y el bias cuando el error es distinto de cero."""
        self.bias = round(entradas[0], 3)
        self.pesos[0] = round(self.pesos[0] + entradas[1], 3)
        self.pesos[1] = round(self.pesos[1] + entradas[2], 3)

    def front_propagation(self, *entradas):
        """Propagar los datos por la neurona obteniendo el valor de salida."""
        z = self.suma_ponderada(*entradas)
        return self.activacion(z)

    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]
            a = self.front_propagation(x1, x2)
            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."""
    @classmethod
    def descenso_gradiente(cls, 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]
            a = neurona.front_propagation(x1, x2)
            y = entradas[i+2]
            e = cls.calcular_error(y, a)
            if e != 0:
                pesos_nuevos = BackPropagation.nuevos_pesos(x1, x2, error=e, neurona=neurona)
                neurona.actualizar_pesos(pesos_nuevos)
                cont = 1
        return cont

    @staticmethod
    def nuevos_pesos(*valores, error: float, neurona: Neurona) -> tuple[int]:
        b_nuevo = 1 * neurona.cte_aprendizaje * error
        w1_nuevo = valores[0] * neurona.cte_aprendizaje * error
        w2_nuevo = valores[1] * neurona.cte_aprendizaje * error
        return (b_nuevo, w1_nuevo, w2_nuevo)

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

    @classmethod
    def epocar_hasta_optimizar(cls, neurona: Neurona, entradas: List[int], queue: Queue = False) ->
None:
        """Ciclo de generar epocas hasta encontrar los valores adecuados para los pesos.
        La queue es para almacenar los pesos de cada epoca y poder mostrarlos en la interfaz."""
        exitoso = 1
        while exitoso != 0:
            exitoso = cls.descenso_gradiente(neurona, entradas)
            sleep(0.2)
            if queue:
                queue.put((neurona.pesos[0], neurona.pesos[1], neurona.bias))
            else:
                print(f"W1: {neurona.pesos[0]}  W2: {neurona.pesos[1]}  B: {neurona.bias}")
        neurona.estado = "Aprendizaje exitoso"
        if queue:
            queue.put(None)
        print("Optimizacion finalizada")
```

Figura 3. Clase BackPropagation

```python
    def front_propagation(self, *entradas):
        """Propagar los datos por la neurona obteniendo el valor de
salida."ż"= self.suma_ponderada(*entradas)
        return self.activacion(z)
```

Figura 4. Método de propagación hacia delante

```python
    def descenso_gradiente(cls, 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]
            a = neurona.front_propagation(x1, x2)
            y = entradas[i+2]
            e = cls.calcular_error(y, a)
            if e != 0:
                pesos_nuevos = BackPropagation.nuevos_pesos(x1, x2, error=e,
neurona=neurona)neurona.actualizar_pesos(pesos_nuevos)
                cont = 1
        return cont
```

Figura 5. Método de descenso gradiente

```python
def main():
    neurona_and = Neurona(rango_aleatorio=(-1, 1), cte_aprendizaje=1/10)
    datos_and = [0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1]
    print("Neurona AND")
    BackPropagation.epocar_hasta_optimizar(neurona=neurona_and, entradas=datos_and)
    prueba_and = [0, 0, 0, 1, 1, 0, 1, 1]
    neurona_and.resultado(entradas=prueba_and)
    print("\n")
    neurona_or = Neurona(rango_aleatorio=(-1, 1), cte_aprendizaje=1/10)
    datos_or = [0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1]
    print("Neurona OR")
    BackPropagation.epocar_hasta_optimizar(neurona=neurona_or, entradas=datos_or)
    prueba_or = [0, 0, 0, 1, 1, 0, 1, 1]
    neurona_or.resultado(entradas=prueba_or)
```

Figura 6. Ejecución del programa

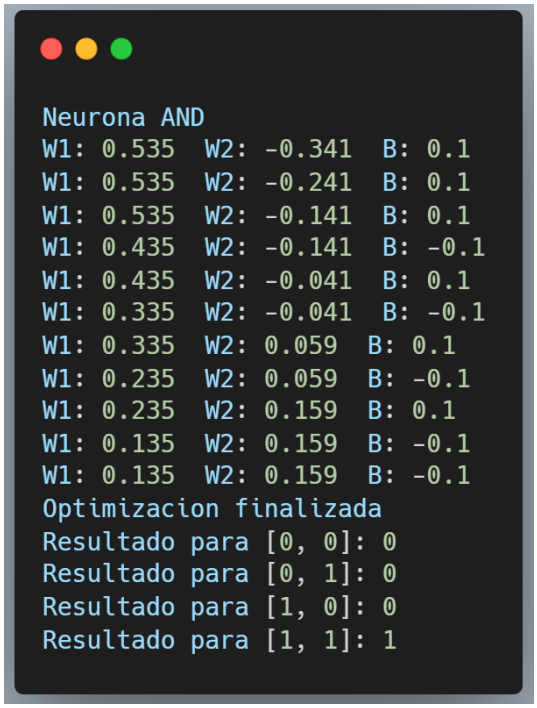


Figura 7. Ejecución en consola de neurona AND

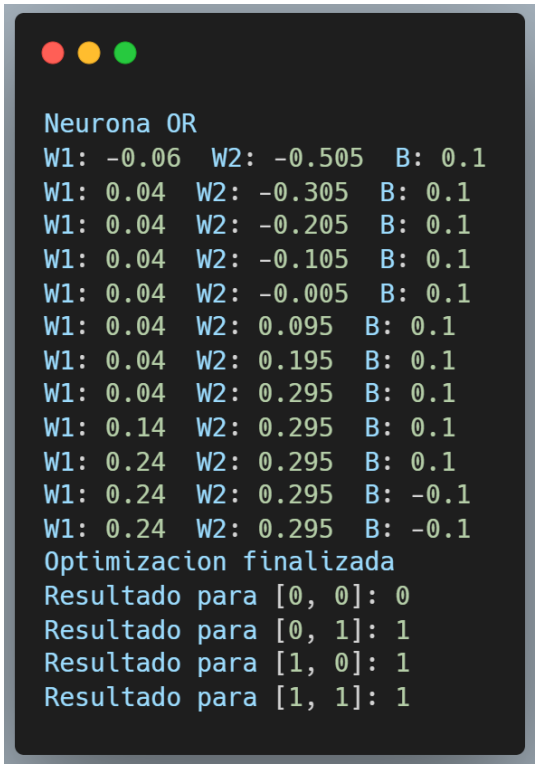


Figura 8. Ejecución en consola de neurona OR