Introducción a Redes Neuronales Artificiales (ANNs)

Una **red neuronal artificial (Artificial Neural Network)** es un modelo computacional inspirado en la forma en que las neuronas del cerebro procesan información. Su objetivo principal es **aprender patrones a partir de datos** para resolver tareas como clasificación, regresión o generación.

Conceptos Fundamentales

1. Neurona Artificial

La unidad básica. Cada neurona realiza esta operación:

$$z = \sum(x_i \cdot w_i) + b \quad \text{y luego} \quad a = \sigma(z)$$

- x_i: entrada
- w_i: peso
- b: sesgo (bias)
- $\sigma(z)$: función de activación (sigmoid, ReLU, etc.)
- a: salida

---

2. Capas

- **Entrada**: recibe los datos originales.
- **Ocultas**: transforman los datos internamente.
- **Salida**: produce la predicción final.

Cada capa tiene varias neuronas conectadas a la capa anterior.

---

3. Funciones de Activación

Añaden **no linealidad**, lo que permite a la red aprender relaciones complejas.

Función	Fórmula	Características
Sigmoid	$\frac{1}{1+e^{-x}}$	Salida entre 0 y 1, buena para probabilidades

Función	Fórmula	Características
Tanh	$\tanh(x)$	Salida entre -1 y 1
ReLU	$\max(0, x)$	Rápida, evita desvanecimiento del gradiente
Softmax	~~$exi \sum exj$~~ $\frac{e^{x_i}}{\sum e^{x_j}}$	Para clasificación multiclase

Cambiar formula

4. Función de pérdida (Loss Function)

Mide qué tan lejos está la predicción del valor real.

Tipo de problema	Función de pérdida común
Clasificación binaria	`binary_crossentropy`
Clasificación multiclase	`categorical_crossentropy`
Regresión	`mean_squared_error (MSE)`

5. Propagación hacia adelante (Forward Propagation)

Se calculan las salidas de cada neurona **desde la entrada hasta la salida** final.

6. Retropropagación (Backpropagation)

Algoritmo que calcula cómo deben ajustarse los pesos para reducir el error. Se basa en el **gradiente del error con respecto a los pesos** y aplica el **descenso del gradiente**.

7. Optimización

Actualiza pesos y sesgos. Algoritmos comunes:

- `SGD` (Stochastic Gradient Descent)
- `Adam` (ajusta automáticamente la tasa de aprendizaje)
- `RMSProp`

🛠️ Conceptos Clave para Implementar en Python

🔢 Preprocesamiento

- **Normalizar/estandarizar** los datos para que los valores estén en rangos similares.

- **Codificar** etiquetas si es clasificación (One-hot o label encoding).

🧱 Inicialización de Pesos

- Crucial para que la red converja (ej: He, Xavier).

🧠 Arquitectura

- ¿Cuántas capas ocultas?
- ¿Cuántas neuronas por capa?
- ¿Qué función de activación?

📉 Evitar Overfitting

El modelo está muy ajustado a reconocet un caso.

- **Regularización**: L2/L1
- **Dropout**: apaga neuronas aleatoriamente durante el entrenamiento
- **Early stopping**: detener si la validación deja de mejorar

🖨️ Batch size y Epochs

- `batch_size`: cuántas muestras se usan por paso de entrenamiento.
- `epochs`: cuántas veces pasa todo el dataset por la red.

🧪 Ejemplo de Red Neuronal Desde Cero (Python + NumPy)

```
import numpy as np

# Sigmoid y derivada
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_deriv(x):
    return x * (1 - x)

# Datos (XOR)
X = np.array([[0,0], [0,1], [1,0], [1,1]])
y = np.array([[0], [1], [1], [0]])
```

```
# Pesos aleatorios
np.random.seed(42)
w0 = 2 * np.random.random((2, 4)) - 1  # capa oculta (2 entradas, 4 neuron
as)
w1 = 2 * np.random.random((4, 1)) - 1  # salida (4 entradas, 1 salida)

# Entrenamiento
for epoch in range(10000):
    # Forward
    l0 = X
    l1 = sigmoid(np.dot(l0, w0))
    l2 = sigmoid(np.dot(l1, w1))

    # Error
    error = y - l2
    if epoch % 1000 == 0:
        print(f"Error: {np.mean(np.abs(error))}")

    # Backpropagation
    l2_delta = error * sigmoid_deriv(l2)
    l1_delta = l2_delta.dot(w1.T) * sigmoid_deriv(l1)

    # Actualizar pesos
    w1 += l1.T.dot(l2_delta)
    w0 += l0.T.dot(l1_delta)

# Predicciones
print("Predicción final:")
print(l2)
```

🤖 Ejemplo con Keras (Alto Nivel)

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import numpy as np

# XOR
```

```
X = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([[0],[1],[1],[0]])

# Modelo
model = Sequential()
model.add(Dense(4, input_dim=2, activation='relu'))  # capa oculta
model.add(Dense(1, activation='sigmoid'))            # capa salida

# Compilar y entrenar
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['ac
curacy'])
model.fit(X, y, epochs=1000, verbose=0)

# Predicciones
print("Predicciones:")
print(model.predict(X))
```

🧠 Recomendaciones Finales

- Empieza con pocas capas y aumenta la complejidad solo si es necesario.
- Siempre separa tus datos en **entrenamiento y validación**.
- Usa `sklearn` para preparar datos y evaluar modelos (confusion matrix, accuracy, etc.).
- Visualiza la evolución del error con `matplotlib`.