

## SESIÓN DISEÑO DE REDES NEURONALES EN PYTHON

### CONTENIDOS:

- El entorno Python para redes neuronales (Keras o PyTorch).
- Qué es un tensor.
- Diseño de una RN para resolver un problema de regresión.
- Diseño de una RN para resolver un problema de clasificación.
- Funciones de activación.

### EL ENTORNO PYTHON PARA REDES NEURONALES (Keras o PyTorch)

Python es uno de los lenguajes más utilizados para trabajar con redes neuronales, principalmente por su simplicidad y las poderosas librerías que ofrece. Las dos más populares son **Keras** y **PyTorch**.

- **Keras**: Es una API de alto nivel escrita en Python que se utiliza para desarrollar y entrenar modelos de redes neuronales. Está construida sobre **TensorFlow** y es conocida por su simplicidad y facilidad de uso. Keras abstrae muchas de las complejidades de TensorFlow y permite a los usuarios crear redes neuronales rápidamente.
- **PyTorch**: Es una librería de Python que proporciona una base para redes neuronales profundas. A diferencia de Keras, PyTorch es más flexible y permite un control más granular sobre los modelos y los procesos de entrenamiento. Es ampliamente usado por investigadores y en la academia debido a su flexibilidad y su capacidad para trabajar con redes neuronales dinámicas.

Ambas son herramientas poderosas, pero la elección entre Keras y PyTorch depende de la experiencia del usuario, las necesidades específicas y la preferencia personal.

### QUÉ ES UN TENSOR

En el contexto de redes neuronales, un **tensor** es una estructura de datos que puede almacenar información en múltiples dimensiones. Es similar a un array de **NumPy**, pero más flexible y eficiente para operaciones en hardware como GPUs.

- Un **tensor de orden 0** es un **escalar** (un número único).

- Un **tensor de orden 1** es un **vector** (una lista de números).
- Un **tensor de orden 2** es una **matriz** (una tabla bidimensional de números).
- Un **tensor de orden 3 o superior** puede tener más dimensiones, por ejemplo, un tensor de orden 3 podría representar una imagen a color con dimensiones de altura, ancho y canales de color.

## DISEÑO DE UNA RN PARA RESOLVER UN PROBLEMA DE REGRESIÓN

Un problema de **regresión** busca predecir valores continuos, por ejemplo, la temperatura o el precio de una casa. Para resolverlo con una red neuronal:

- **Capa de entrada:** La cantidad de neuronas debe coincidir con las características del conjunto de datos (por ejemplo, número de variables independientes).
- **Capas ocultas:** Pueden tener varias capas densas (fully connected), con una cantidad adecuada de neuronas. El número de capas y neuronas depende de la complejidad del problema.
- **Capa de salida:** En un problema de regresión, la capa de salida suele tener una sola neurona, ya que se predice un solo valor continuo.
- **Función de activación:** Para un problema de regresión, no se suele usar una función de activación en la capa de salida, ya que se desea que el valor de salida sea continuo.
- **Función de pérdida:** La función de pérdida comúnmente usada para problemas de regresión es **Mean Squared Error (MSE)**.

## Llevándolo a la práctica

En esta actividad práctica guiada resolveremos un problema de regresión usando una red neuronal. Utilizaremos **Keras** y trabajaremos con un conjunto de datos sencillos provenientes de la librería **scikit-learn**.

### Contexto:

Imaginemos que estamos construyendo una red neuronal para predecir el **precio de casas** en función de características como el número de habitaciones, la ubicación, el tamaño, etc. Este es un problema de regresión donde el objetivo es predecir un valor continuo (el precio de la casa).

### Herramientas a utilizar:

- **Python:** Lenguaje de programación.
- **Keras/TensorFlow:** Framework para crear y entrenar redes neuronales.
- **Scikit-learn:** Librería para cargar datasets de ejemplo, preprocesar datos y dividirlos en conjuntos de entrenamiento y prueba.
- **Matplotlib:** Para visualizar los resultados.

### Pasos para la actividad práctica:

#### 1. Instalar las librerías necesarias

Antes de comenzar, asegúrate de tener las siguientes librerías instaladas. Si no las tienes, puedes instalarlas usando pip:

```
pip install tensorflow scikit-learn matplotlib
```

#### 2. Cargar y preparar los datos

Para este ejemplo, utilizaremos el **dataset de Boston Housing** de scikit-learn. Este dataset contiene información sobre casas en Boston, con variables como el número de habitaciones, el índice de criminalidad, la proximidad a los centros de empleo, entre otras. Vamos a usar estas características para predecir el precio de la casa.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Cargar el dataset de Boston Housing
data = load_boston()
X = data.data # Características (por ejemplo, número de habitaciones)
y = data.target # Precio de las casas (variable continua)

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Normalizar los datos (esto mejora la eficiencia del entrenamiento)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

### 3. Definir la red neuronal

Ahora vamos a crear la red neuronal utilizando **Keras**. Definimos una red simple con una capa de entrada, una capa oculta y una capa de salida.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Crear el modelo de la red neuronal
model = Sequential()

# Capa de entrada (tiene 13 características de entrada)
model.add(Dense(64, input_dim=X_train.shape[1], activation='relu'))

# Capa oculta
model.add(Dense(32, activation='relu'))

# Capa de salida (una neurona, ya que es un problema de regresión)
model.add(Dense(1))

# Compilar el modelo
model.compile(optimizer='adam', loss='mean_squared_error')
```

### 4. Entrenar el modelo

Entrenamos el modelo usando el conjunto de entrenamiento. Podemos configurar el número de épocas y el tamaño del batch.

```
# Entrenar el modelo
history = model.fit(X_train, y_train, epochs=100, batch_size=32, validation_split=0.2)
```

## 6. Visualizar el rendimiento del modelo

Finalmente, visualizamos cómo ha evolucionado la pérdida durante el entrenamiento para asegurarnos de que el modelo ha aprendido correctamente.

```
# Graficar la evolución de la pérdida durante el entrenamiento
plt.plot(history.history['loss'], label='Entrenamiento')
plt.plot(history.history['val_loss'], label='Validación')
plt.xlabel('Épocas')
plt.ylabel('Pérdida')
plt.legend()
plt.show()
```

## 7. Realizar predicciones

Una vez entrenado el modelo, podemos hacer predicciones sobre nuevos datos.

```
# Realizar predicciones sobre el conjunto de prueba
predictions = model.predict(X_test)

# Comparar las predicciones con los valores reales
for i in range(10):
    print(f"Real: {y_test[i]}, Predicción: {predictions[i]}")
```

## DISEÑO DE UNA RN PARA RESOLVER UN PROBLEMA DE CLASIFICACIÓN

En un problema de **clasificación**, el objetivo es asignar una categoría o clase a una entrada. Para este caso:

- **Capa de entrada:** La cantidad de neuronas coincide con las características del conjunto de datos.
- **Capas ocultas:** Se pueden agregar múltiples capas ocultas, con activaciones como **ReLU**.
- **Capa de salida:** El número de neuronas en la capa de salida corresponde al número de clases. En una clasificación binaria, sería una sola neurona; en una clasificación multiclase, el número de neuronas será igual al número de clases.
- **Función de activación:** Para problemas de clasificación, la capa de salida generalmente usa la función de activación **Softmax** (para clasificación multiclase) o **Sigmoid** (para clasificación binaria).
- **Función de pérdida:** Para clasificación, las funciones de pérdida más comunes son **Categorical Crossentropy** (para clasificación multiclase) y **Binary Crossentropy** (para clasificación binaria).

### Llevándolo a la práctica

A continuación, llevaremos a cabo una actividad práctica guiada para resolver un problema de clasificación utilizando una red neuronal. Usaremos Keras y trabajaremos con un conjunto de datos clásico de clasificación: el dataset de Iris que se encuentra en scikit-learn. El objetivo será predecir la especie de una flor (Setosa, Versicolor o Virginica) en función de las características de la flor.

#### Contexto:

Imaginemos que estamos construyendo una red neuronal para predecir la especie de una flor del conjunto de datos Iris, utilizando características como el largo y el ancho del sépalo y el pétalo. Este es un problema de clasificación multiclase.

## Herramientas a utilizar:

- **Python:** Lenguaje de programación.
- **Keras/TensorFlow:** Framework para crear y entrenar redes neuronales.
- **Scikit-learn:** Librería para cargar datasets de ejemplo, preprocesar datos y dividirlos en conjuntos de entrenamiento y prueba.
- **Matplotlib:** Para visualizar los resultados.

## Pasos para la actividad práctica:

### 1. Instalar las librerías necesarias

Si aún no tienes las librerías necesarias, instálalas como vimos en la actividad práctica anterior.

### 2. Cargar y preparar los datos

Vamos a cargar el **dataset de Iris** desde scikit-learn. Este dataset contiene 150 muestras de flores con 4 características: largo y ancho del sépalo y del pétalo. Hay tres clases posibles: Setosa, Versicolor y Virginica.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.utils import to_categorical

# Cargar el dataset Iris
data = load_iris()
X = data.data # Características (largo y ancho de los sépalos y pétalos)
y = data.target # Clases (Setosa, Versicolor, Virginica)

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Normalizar los datos
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Convertir las etiquetas a formato "one-hot" (para clasificación multiclase)
y_train = to_categorical(y_train, 3)
y_test = to_categorical(y_test, 3)
```

### 3. Definir la red neuronal

Ahora vamos a crear una red neuronal para clasificar las flores. Tendremos una capa de entrada (con 4 neuronas, una por cada característica), una o más capas ocultas y una capa de salida con 3 neuronas, ya que tenemos 3 clases (Setosa, Versicolor, Virginica).

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Crear el modelo de la red neuronal
model = Sequential()

# Capa de entrada (4 características de entrada)
model.add(Dense(64, input_dim=X_train.shape[1], activation='relu'))

# Capa oculta
model.add(Dense(32, activation='relu'))

# Capa de salida (3 neuronas, una para cada clase)
model.add(Dense(3, activation='softmax'))

# Compilar el modelo
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

### 4. Entrenar el modelo

Entrenamos el modelo con el conjunto de entrenamiento y verificamos su rendimiento en el conjunto de validación.

```
# Entrenar el modelo
history = model.fit(X_train, y_train, epochs=100, batch_size=32, validation_split=0.2)
```

### 5. Evaluar el modelo

Una vez entrenado el modelo, evaluamos su desempeño en el conjunto de prueba.

```
# Evaluar el modelo
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Precisión en el conjunto de prueba: {accuracy*100:.2f}%")
```



## 6. Visualizar el rendimiento del modelo

Para ver cómo ha mejorado el modelo durante el entrenamiento, graficamos la precisión y la pérdida.

```
# Graficar la precisión y la pérdida durante el entrenamiento
plt.plot(history.history['accuracy'], label='Precisión en entrenamiento')
plt.plot(history.history['val_accuracy'], label='Precisión en validación')
plt.xlabel('Épocas')
plt.ylabel('Precisión')
plt.legend()
plt.show()

plt.plot(history.history['loss'], label='Pérdida en entrenamiento')
plt.plot(history.history['val_loss'], label='Pérdida en validación')
plt.xlabel('Épocas')
plt.ylabel('Pérdida')
plt.legend()
plt.show()
```

## 7. Realizar predicciones


Finalmente, realizamos predicciones con el conjunto de prueba y comparamos las clases predichas con las clases reales.

```
# Realizar predicciones sobre el conjunto de prueba
predictions = model.predict(x_test)

# Mostrar las clases predichas y las clases reales
for i in range(10):
    print(f"Real: {np.argmax(y_test[i])}, Predicción: {np.argmax(predictions[i])}")
```

## FUNCIONES DE ACTIVACIÓN

Las funciones de activación controlan cómo se activan las neuronas en una red neuronal, introduciendo no linealidades que permiten a la red aprender patrones complejos.

- 
- **ReLU (Rectified Linear Unit):** Es la más común. Es simple y eficaz. La función devuelve el valor de entrada si es positivo y 0 si es negativo. Ayuda a resolver el problema de gradientes vanishing, lo que permite entrenar redes neuronales profundas.
  - **Sigmoid:** Devuelve un valor entre 0 y 1. Se usa en redes neuronales para problemas de clasificación binaria. Sin embargo, puede causar el problema de gradientes vanishing en redes profundas.
  - **Tanh (Tangente Hiperbólica):** Similar a Sigmoid, pero su rango es entre -1 y 1. Es más útil que Sigmoid para ciertos problemas debido a que centra los valores en torno a 0.
  - **Softmax:** Se usa en la capa de salida para problemas de clasificación multiclase. Convierte los valores de salida en probabilidades, sumando 1 en todas las clases.