SESIÓN ENTRENAMIENTO Y OPTIMIZACIÓN DE REDES NEURONALES

CONTENIDOS:

- Entrenamiento de la Red Neuronal
    - función de pérdida.
    - Optimizadores.
    - Learning rate.
    - Epochs.
- Evaluación y optimización de la red
- Validación y regularización
    - El problema del sobreajuste en las rrnn.
    - Dropout.
- Realizando predicciones con el modelo neuronal.

ENTRENAMIENTO DE LA RED NEURONAL

El proceso de entrenamiento de una red neuronal implica ajustar los pesos de las conexiones entre las neuronas a través de la retropropagación del error. Esto se hace utilizando un conjunto de datos de entrenamiento, en el cual la red intenta aprender las relaciones entre las entradas y salidas. En cada iteración del entrenamiento, la red ajusta sus pesos para reducir la diferencia entre la salida predicha y la salida esperada (la verdadera).

Función de pérdida

La función de pérdida (o función de coste) mide qué tan bien o mal está realizando la red neuronal su tarea. El objetivo del entrenamiento es minimizar esta función. Existen diferentes tipos de funciones de pérdida, dependiendo del tipo de problema:

- **Regresión**: Una función común es el **error cuadrático medio (MSE)**, que calcula la diferencia cuadrada entre el valor predicho y el valor real.
- **Clasificación**: Se usa comúnmente la **entropía cruzada (cross-entropy loss)**, que mide la diferencia entre la probabilidad predicha y la real.

Optimizadores

Los optimizadores son algoritmos que ajustan los pesos de la red neuronal durante el entrenamiento. Usan la derivada de la función de pérdida con respecto a los pesos para decidir cómo ajustar los valores de los pesos. Algunos de los optimizadores más comunes son:

- **SGD (Stochastic Gradient Descent)**: Actualiza los pesos utilizando un solo ejemplo de entrenamiento a la vez. Es más ruidoso, pero puede ser más eficiente para grandes datasets.
- **Adam (Adaptive Moment Estimation)**: Combina las ventajas de **AdaGrad** y **RMSprop**, adaptando el learning rate para cada parámetro, y utilizando tanto el promedio de los gradientes como el promedio de los cuadrados de los gradientes.

Learning Rate (Tasa de Aprendizaje)

El learning rate es un hiperparámetro que controla cuánto se ajustan los pesos en cada iteración. Si el learning rate es muy pequeño, el proceso de aprendizaje será lento, y si es muy grande, puede causar que la red no converja correctamente. En general, el aprendizaje efectivo requiere un learning rate adecuado y puede ser ajustado durante el entrenamiento utilizando técnicas como el **learning rate decay** o **annealing**, donde el learning rate disminuye gradualmente a medida que el entrenamiento progresa.

Epochs

Una **epoch** es una iteración completa a través de todo el conjunto de datos de entrenamiento. Durante una epoch, la red neuronal ve cada ejemplo de entrenamiento una vez y ajusta sus pesos. Usualmente se necesitan varias epochs para que la red aprenda adecuadamente, aunque si el número de epochs es demasiado alto, la red puede sobreajustarse (overfitting), es decir, aprender demasiado bien los datos de entrenamiento, pero no generalizar bien a nuevos datos.

EVALUACIÓN Y OPTIMIZACIÓN DE LA RED

Después del entrenamiento, es fundamental evaluar y optimizar la red neuronal para asegurar que generalice bien a datos nuevos. Esto implica medir su rendimiento, detectar problemas como el sobreajuste y mejorar su desempeño mediante técnicas avanzadas.

Evaluación de la Red Neuronal

La evaluación de la red neuronal se realiza utilizando un conjunto de datos de prueba que no fue visto durante el entrenamiento. Se emplean métricas adecuadas según el tipo de problema:

Métricas para Problemas de Regresión

Para problemas en los que la red predice valores continuos, las métricas más comunes son:

- **Error Cuadrático Medio (MSE)**:

$$MSE = \frac{1}{n}\sum(y_{\text{real}} - y_{\text{predicho}})^2$$

Ilustración 1 MSE

Penaliza errores grandes y es útil cuando los errores extremos son indeseables.

- **Error Absoluto Medio (MAE)**:

$$MAE = \frac{1}{n}\sum|y_{\text{real}} - y_{\text{predicho}}|$$

Ilustración 2 MAE

Es más interpretable, ya que mide el error promedio en las mismas unidades de los datos.

Métricas para Problemas de Clasificación

Cuando la red neuronal clasifica datos en categorías, se utilizan métricas como:

- **Precisión (Accuracy)**:

$$\text{Accuracy} = \frac{\text{número de predicciones correctas}}{\text{total de predicciones}}$$

Ilustración 3 Precisión

Es útil si las clases están balanceadas.

- **Matriz de Confusión**: Muestra cómo se distribuyen las predicciones entre las clases.

- **Precisión, Recall y F1-Score**:
    - **Precisión**: Proporción de predicciones correctas dentro de cada clase.
    - **Recall**: Capacidad del modelo para encontrar todos los ejemplos de una clase dada.
    - **F1-Score**: Promedio armónico de precisión y recall, útil en datos desbalanceados.
- **AUC-ROC**: Mide la capacidad del modelo para distinguir entre clases positivas y negativas.

Optimización de la Red Neuronal

Una vez entrenada la red neuronal, es fundamental optimizar su rendimiento para mejorar la precisión y evitar problemas como el sobreajuste o el aprendizaje deficiente. La optimización implica ajustar hiperparámetros, aplicar técnicas de regularización y mejorar la eficiencia del entrenamiento para lograr una red que generalice bien a datos nuevos. A continuación, se presentan estrategias clave para optimizar una red neuronal.

Regularización para Evitar el Sobreajuste

El sobreajuste ocurre cuando la red aprende demasiado bien los datos de entrenamiento, pero falla con datos nuevos. Algunas soluciones son:

- **Dropout**: Desactiva aleatoriamente neuronas durante el entrenamiento para evitar dependencia excesiva de ciertas conexiones.
- **Regularización L1 y L2 (Lasso y Ridge)**: Agregan penalizaciones a los pesos para evitar valores demasiado grandes.
- **Aumento de Datos (Data Augmentation)**: En imágenes y texto, se pueden aplicar transformaciones para generar más ejemplos artificiales.

Ajuste de Hiperparámetros

Para mejorar el rendimiento de la red, se ajustan parámetros como:

- **Learning Rate**: Un valor demasiado alto impide la convergencia, mientras que uno muy bajo hace que el entrenamiento sea lento. Se pueden usar estrategias como el **learning rate scheduling**.
- **Número de Capas y Neuronas**: Más capas pueden mejorar la capacidad de aprendizaje, pero también aumentan el riesgo de sobreajuste.
- **Función de Activación**: ReLU suele ser la más utilizada en capas ocultas, pero en la última capa se usan funciones específicas según el problema (Softmax para clasificación, Sigmoide para binario, etc.).
- **Número de Epochs y Batch Size**: Un número de epochs muy alto puede llevar al sobreajuste, mientras que un tamaño de batch pequeño puede hacer que el entrenamiento sea más ruidoso pero eficiente.

Técnicas Avanzadas de Optimización

- **Batch Normalization**: Normaliza las activaciones en cada capa para mejorar la estabilidad del entrenamiento.
- **Transfer Learning**: Utiliza redes previamente entrenadas y las ajusta a una nueva tarea con menos datos.
- **Early Stopping**: Detiene el entrenamiento automáticamente cuando la validación comienza a empeorar.

VALIDACIÓN Y REGULARIZACIÓN

El entrenamiento de una red neuronal no solo debe enfocarse en lograr un buen desempeño en los datos de entrenamiento, sino también en garantizar que el modelo generalice bien a datos nuevos. Para esto, se emplean técnicas de validación y regularización, que ayudan a evitar el sobreajuste y mejorar la capacidad de predicción del modelo.

El problema del sobreajuste en las Redes Neuronales

El **sobreajuste (overfitting)** ocurre cuando una red neuronal aprende demasiado bien los detalles y ruido del conjunto de entrenamiento, en lugar de identificar patrones generales. Como resultado, la red tiene un rendimiento excelente en los datos de entrenamiento, pero falla al enfrentarse a datos nuevos.

Causas del sobreajuste

- Modelo demasiado complejo (demasiadas capas o neuronas).
- Cantidad insuficiente de datos de entrenamiento.
- Número de epochs demasiado alto.
- Falta de regularización o técnicas de prevención.

Cómo detectar el sobreajuste

- **Diferencia entre el error de entrenamiento y el error de validación**: Si el error de entrenamiento es bajo, pero el error en los datos de validación es alto, hay sobreajuste.
- **Curvas de pérdida divergentes**: Si la pérdida en el conjunto de entrenamiento sigue disminuyendo mientras la de validación comienza a aumentar, la red está memorizando en lugar de aprender.

Dropout

Una de las técnicas más efectivas para reducir el sobreajuste es el **Dropout**, que consiste en desactivar aleatoriamente un porcentaje de neuronas en cada iteración del entrenamiento.

¿Cómo funciona Dropout?

Durante el entrenamiento, cada neurona tiene una probabilidad ppp de ser eliminada temporalmente (es decir, no contribuir a la salida ni recibir gradientes). Esto obliga a la red a aprender representaciones más robustas y a no depender demasiado de ciertas conexiones.

Ventajas de Dropout

- ✓ Reduce la dependencia excesiva de neuronas individuales.
- ✓ Actúa como una forma de ensemble, entrenando múltiples sub-redes dentro del mismo modelo.
- ✓ Mejora la generalización del modelo.

ACTIVIDAD PRÁCTICA GUIADA: REALIZANDO PREDICCIONES CON EL MODELO NEURONAL

Imagina que trabajas en una empresa de salud que quiere predecir la probabilidad de que un paciente desarrolle diabetes basado en ciertas características médicas. Para ello, utilizaremos una **red neuronal** que primero entrenaremos y luego emplearemos para hacer predicciones en nuevos pacientes.

En esta actividad, usaremos el **dataset "Pima Indians Diabetes"**, disponible en la librería **sklearn.datasets**, que contiene información sobre distintos factores de riesgo y un indicador de si el paciente tiene diabetes o no.

Objetivo: Entrenar un modelo de red neuronal para predecir la probabilidad de diabetes en pacientes nuevos.

Paso 1: Instalación y carga de librerías

Primero, asegúrate de que tienes instaladas las librerías necesarias. Puedes ejecutar lo siguiente en tu entorno:

```
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import fetch_openml
```

Paso 2: Carga y exploración de los datos

Usaremos el dataset **Pima Indians Diabetes**, que se encuentra en OpenML y puede ser descargado con **fetch_openml**.

```
# Cargar el dataset desde OpenML
data = fetch_openml(name="diabetes", version=1, as_frame=True)
df = data.frame

# Ver las primeras filas del dataset
print(df.head())
```

Descripción de las columnas:

- **Pregnancies**: Número de embarazos
- **Glucose**: Nivel de glucosa en sangre
- **BloodPressure**: Presión arterial diastólica
- **SkinThickness**: Grosor del pliegue cutáneo
- **Insulin**: Nivel de insulina en sangre
- **BMI**: Índice de masa corporal
- **DiabetesPedigreeFunction**: Historial genético de diabetes
- **Age**: Edad del paciente
- **Outcome**: 1 = Tiene diabetes, 0 = No tiene diabetes (variable objetivo)

Paso 3: Preprocesamiento de los datos

Antes de entrenar la red neuronal, debemos dividir los datos en entrenamiento y prueba, además de escalar las variables para mejorar la convergencia del modelo.

```python
# Separar características (X) y variable objetivo (y)
X = df.drop(columns=["Outcome"])
y = df["Outcome"]

# Dividir en conjuntos de entrenamiento (80%) y prueba (20%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Escalar los datos para mejorar el rendimiento del modelo
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Paso 4: Construcción de la Red Neuronal

Crearemos un modelo de red neuronal con tres capas densas y una capa de salida con activación sigmoide para clasificación binaria.

```python
# Definir la estructura del modelo
modelo = Sequential([
    Dense(16, activation='relu', input_shape=(X_train.shape[1],)),
    Dropout(0.3),
    Dense(8, activation='relu'),
    Dense(1, activation='sigmoid')  # Capa de salida para clasificación binaria
])

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

# Ver resumen del modelo
modelo.summary()
```

Paso 5: Entrenamiento del Modelo

Entrenaremos el modelo usando el conjunto de entrenamiento y validaremos su desempeño en los datos de prueba.

```python
# Entrenar el modelo
historial = modelo.fit(X_train, y_train, epochs=50, batch_size=16, validation_data=(X_test, y_test))
```

Paso 6: Evaluación del Modelo

Ahora evaluaremos la precisión del modelo en los datos de prueba.

```
# Evaluar el modelo en el conjunto de prueba
loss, accuracy = modelo.evaluate(X_test, y_test)
print(f"Precisión del modelo en datos de prueba: {accuracy:.4f}")
```

Si la precisión es aceptable, podemos pasar a realizar predicciones.

Paso 7: Realizando Predicciones con el Modelo

Simularemos que llega un nuevo paciente y queremos predecir su probabilidad de tener diabetes.

```
# Crear datos de un nuevo paciente (simulación)
nuevo_paciente = np.array([[2, 120, 70, 32, 85, 25.5, 0.67, 45]])  # Datos ficticios
nuevo_paciente = scaler.transform(nuevo_paciente)  # Aplicar la misma normalización

# Hacer la predicción
probabilidad = modelo.predict(nuevo_paciente)[0][0]
print(f"Probabilidad de tener diabetes: {probabilidad:.2%}")
```

Si el valor es mayor a 0.5, es más probable que el paciente tenga diabetes, de lo contrario, es menos probable.

Paso 8: Visualización de Resultados

Para analizar el desempeño del modelo, graficamos la evolución de la pérdida y precisión durante el entrenamiento.

```
import matplotlib.pyplot as plt

# Gráfica de la pérdida
plt.plot(historial.history['loss'], label='Pérdida en entrenamiento')
plt.plot(historial.history['val_loss'], label='Pérdida en validación')
plt.xlabel('Épocas')
plt.ylabel('Pérdida')
plt.legend()
plt.show()

# Gráfica de la precisión
plt.plot(historial.history['accuracy'], label='Precisión en entrenamiento')
plt.plot(historial.history['val_accuracy'], label='Precisión en validación')
plt.xlabel('Épocas')
plt.ylabel('Precisión')
plt.legend()
plt.show()
```