Implementación de un motor de Red Neuronal

Joel Clemente López , Adonai Ojeda , Daniel Medina

November 4, 2024

Abstract

Este proyecto consiste en la implementación de una red neuronal desde cero en Python, sin utilizar librerías de aprendizaje profundo como TensorFlow o PyTorch. La red neuronal se ha entrenado y evaluado en tres conjuntos de datos clásicos: **Iris**, **Digits** y **Wine** Este documento explica las clases y funciones diseñadas, los modelos desarrollados, así como el análisis de rendimiento mediante curvas ROC y matrices de confusión. El objetivo es demostrar el proceso de diseño, entrenamiento y evaluación de una red neuronal simple para tareas de clasificación multiclase.

1 Introducción

Las redes neuronales artificiales son modelos computacionales inspirados en el cerebro humano, ampliamente utilizados para el reconocimiento de patrones y clasificación de datos. Este proyecto implementa desde cero una red neuronal, abordando todas las etapas del diseño de una red neuronal: definición de capas, funciones de activación, retropropagación y optimización de los pesos. Para poder poner a prueba nuestra Red Neuronal, hemos seleccionado los conjuntos de datos **Iris**, **Digits** y **Wine** para entrenar y evaluar distintos modelos de red, lo que nos permite analizar cómo diferentes arquitecturas y funciones de activación afectan el rendimiento.

2 Estructura del Proyecto

2.1 Funciones de Activación - activations.py

En `activations.py`, se definen funciones esenciales para introducir no linealidad en la red neuronal, permitiendo al modelo aprender representaciones complejas de los datos. Las funciones implementadas son:

- **ReLU** (*Rectified Linear Unit*): Devuelve 0 para valores negativos y el valor de entrada para valores positivos. Su derivada es 1 para valores positivos y 0 para negativos.

```python
def relu(x):
    """ReLU activation function."""
    return np.maximum(0, x)
```

Figure 1: ReLU function

- **Softmax**: Normaliza las salidas en probabilidades, útil en la capa de salida para clasificación multiclase. Su derivada calcula el gradiente necesario para la retropropagación.

```python
def softmax(x):
    """Softmax activation function."""
    exp_x = np.exp(x - np.max(x))
    return exp_x / np.sum(exp_x, axis=0)
```

Figure 2: SoftMax Function

- **Sigmoid**: Transforma la entrada a un rango entre 0 y 1. Es utilizada en capas ocultas y su derivada es útil en retropropagación.

```python
def sigmoid(x):
    """Sigmoid activation function."""
    return 1 / (1 + np.exp(-x))
```

Figure 3: Sigmoid Function

- **Tanh**: Rango entre -1 y 1, que suele converger más rápido que *sigmoid*. La derivada es útil para retropropagación.

```python
def tanh(x):
    """Tanh activation function."""
    return np.tanh(x)
```

Figure 4: Tanh Function

2.2 Clase de Capa - layer.py

El archivo `layer.py` define la clase `Layer`, que representa una capa individual de la red. Sus principales atributos y métodos son:

- **Atributos**: Dimensiones de la capa, función de activación y derivada.
- **Método forward**: Realiza la propagación hacia adelante en la capa. Calcula las activaciones para cada neurona aplicando la función de activación.

```python
def forward(self, A_prev):
    """
    Realiza la propagación hacia adelante.

    :param A_prev: Salida de la capa anterior.
    :return: Salida de esta capa después de aplicar la activación.
    """
    self.Z = self.compute_z(A_prev)
    return self.activation_func(self.Z)
```

Figure 5: Forward Method

La clase `Layer` permite diseñar capas personalizadas para las redes neuronales, facilitando la configuración de arquitecturas específicas según el conjunto de datos y los objetivos.

2.3 Red Neuronal - neural_network.py

La clase `NeuralNetwork`, en el archivo `neural_network.py`, maneja la estructura completa de la red. Esta clase permite añadir múltiples capas de la clase `Layer` y define los métodos fundamentales para el aprendizaje supervisado:

- **Método feedforward**: Propaga las entradas a través de todas las capas hasta la capa de salida.

```python
def feedforward(self, X, n=None):
    """
    Realiza la propagación hacia adelante.

    :param X: Datos de entrada.
    :param n: Número de capas a considerar en la propagación.
    :return: Salida de la red neuronal.
    """
    n = len(self.layers) if n is None else n

    for layer in self.layers[:n]:
        X = layer.forward(X)
    return X
```

Figure 6: FeedForward method

- **Método backpropagation**: Ajusta los pesos mediante la retropropagación, calculando gradientes y actualizando los pesos.

```python
def backpropagation(self, X, y):
    """
    Realiza la retropropagación para calcular los gradientes.

    :param X: Datos de entrada.
    :param y: Etiqueta verdadera.
    :return: Lista de gradientes para cada capa.
    """

    derivates = []

    A = self.feedforward(X)
    A = A.reshape(len(A),1)
    y = y.reshape(len(y),1)

    A_prev = self.feedforward(X, -1)
    A_prev = A_prev.reshape(len(A_prev),1)

    dz = self.layers[-1].activation_derivate(A, y)
    dw = dz.dot(A_prev.T)

    derivates.append((dz, dw))

    for k, layer in enumerate(self.layers[-2::-1]):
        A_prev = self.feedforward(X, -k-2)
        A_prev = A_prev.reshape(len(A_prev),1)

        derivate = layer.activation_derivate(layer.compute_z(A_prev))
        dz = derivate*self.layers[-k-1].weights.T.dot(dz)
        dw = dz.dot(A_prev.T)

        derivates.append((dz, dw))

    return derivates
```

Figure 7: BackPropagation Method

- **Método accuracy**: Calcula la precisión del modelo comparando etiquetas predichas y reales, proporcionando una medida de rendimiento.

```python
def accuracy(Y_pred, Y):
    """
    Calcula la precisión de las predicciones.

    :param Y_pred: Predicciones de la red.
    :param Y: Etiquetas verdaderas.
    :return: Precisión como porcentaje.
    """
    acc = 0
    for y_pred, y in zip(Y_pred, Y):
        if np.argmax(y_pred) == np.argmax(y):
            acc += 1

    return acc / len(Y_pred)
```

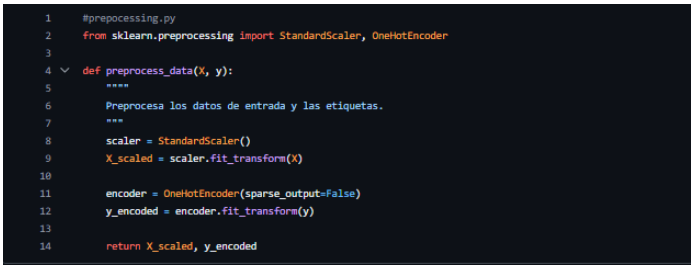
Figure 8: Accuracy method

2.4 Optimizador - optimizers.py

El archivo `optimizers.py` implementa la función `gradient_descent`, un optimizador que ajusta los pesos del modelo. Esta función utiliza el descenso de gradiente para minimizar la pérdida, mejorando la precisión de la red. Por otro lado, el optimizador *Adam* (Adaptive Moment Estimation) mejora la velocidad de convergencia al calcular promedios móviles de primer y segundo orden de los gradientes, utilizando parámetros de momento (beta1 y beta2). Esta estrategia permite ajustar dinámicamente la tasa de aprendizaje, haciendo a Adam más eficiente en problemas de optimización complicados y con grandes volúmenes de datos.

2.5 Preprocesamiento - preprocessing.py

El archivo `preprocessing.py` contiene la función `preprocess_data`, que transforma los datos de entrada mediante normalización y estandarización, optimizando así el rendimiento de la red.



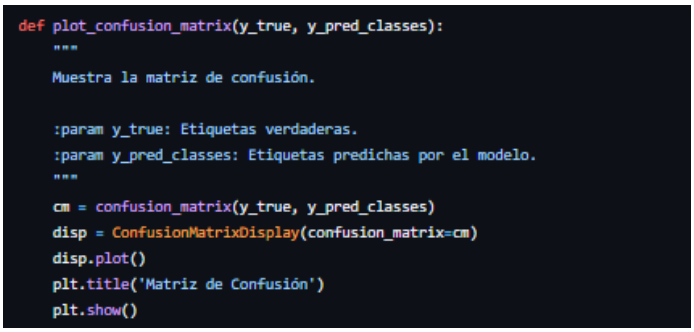
```python
#prepocessing.py
from sklearn.preprocessing import StandardScaler, OneHotEncoder

def preprocess_data(X, y):
    """
    Preprocesa los datos de entrada y las etiquetas.
    """
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)

    encoder = OneHotEncoder(sparse_output=False)
    y_encoded = encoder.fit_transform(y)

    return X_scaled, y_encoded
```

Figure 9: Preprocessing.py

2.6 Visualización de Resultados - visualizations.py

El archivo `visualizations.py` incluye funciones para evaluar el rendimiento del modelo visualmente:

- **plot_confusion_matrix**: Genera una matriz de confusión, permitiendo analizar las clasificaciones correctas e incorrectas.



```python
def plot_confusion_matrix(y_true, y_pred_classes):
    """
    Muestra la matriz de confusión.

    :param y_true: Etiquetas verdaderas.
    :param y_pred_classes: Etiquetas predichas por el modelo.
    """
    cm = confusion_matrix(y_true, y_pred_classes)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm)
    disp.plot()
    plt.title('Matriz de Confusión')
    plt.show()
```

Figure 10: Plot Confusion Matrix

- **roc_curve**: Muestra la curva ROC para analizar la tasa de verdaderos y falsos positivos en cada clase.

```python
def scatter_roc_curve(nn, X_test, y_test):
    """
    Función que grafica la curva ROC para un modelo de red neuronal.

    Parámetros:
    nn: instancia de la clase NeuralNetwork.
    X_test: matriz de características de prueba.
    y_test: matriz de etiquetas de prueba.
    """

    if len(y_test.shape) == 1:
        encoder = OneHotEncoder(sparse_output=False)
        y_test = encoder.fit_transform(y_test.reshape(-1, 1))

    y_probs = np.array([nn.feedforward(x) for x in X_test])

    plt.figure(figsize=(8, 6))

    for i in range(y_test.shape[1]):
        y_true_binary = y_test[:, i]
        y_probs_binary = y_probs[:, i]

        fpr, tpr, _ = roc_curve(y_true_binary, y_probs_binary)

        plt.scatter(fpr, tpr, label=f'Clase {i}', s=10)  # Ajusta el tamaño de los puntos según prefieras

    plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
    plt.xlabel('Tasa de Falsos Positivos')
    plt.ylabel('Tasa de Verdaderos Positivos')
    plt.title('Curva ROC - Gráfica de Dispersión')
    plt.legend()
    plt.show()
```

Figure 11: Roc Curve

2.7 Carga de Datos - load_data.py

El archivo `load_data.py` en la carpeta `data` se utiliza para cargar los conjuntos de datos Iris, Digits y Wine. Este archivo contiene funciones para obtener los datos y etiquetas necesarios para el entrenamiento.

```python
# load_data.py
from sklearn.datasets import load_iris, load_digits, load_wine

def load_iris_data():
    """
    Carga y devuelve el dataset de Iris.
    """
    iris = load_iris()
    X = iris['data']
    y = iris['target'].reshape(-1, 1)
    return X, y

def load_digits_data():
    """
    Carga y devuelve el dataset de dígitos.
    """
    digits = load_digits()
    X, y = digits.data, digits.target
    return X, y

def load_wine_data():
    """
    Carga y devuelve el dataset de vinos.
    """
    data = load_wine()
    X, y = data.data, data.target.reshape(-1, 1)
    return X, y
```

Figure 12: Datasets

3 Tests de entrenamiento

3.1 Conjunto de Datos Iris

El conjunto de datos Iris contiene 150 muestras de 3 especies de plantas, con 4 características cada una. Dos modelos de red se entrenaron en Iris:

- **Modelo 1**: Capa oculta de 5 neuronas ($ReLU$) y capa de salida de 3 neuronas ($softmax$).
- **Modelo 2**: Red con 3 capas ocultas de diferentes tamaños y *softmax* en la salida.
- **Modelo 3**: Red con 3 capas ocultas de diferentes tamaños usando la función de activación $ReLU$ y *Tanh* y *softmax* en la salida.

Se evaluó el rendimiento con matrices de confusión y precisión antes y después del entrenamiento.

3.2 Conjunto de Datos Digits

El conjunto de datos Digits tiene imágenes de dígitos (0-9) representados en matrices de 8x8. Dos arquitecturas de red se entrenaron en Digits:

- **Modelo 1**: Capa oculta de 5 neuronas ($tanh$) y capa de salida de 10 neuronas ($softmax$).
- **Modelo 2**: Red profunda con 3 capas ocultas de diferentes tamaños *sigmoid* y *softmax* en la salida.
- **Modelo 3**: Red con 3 capas ocultas de diferentes tamaños usando la función de activación $ReLU$ y *Tanh* y *softmax* en la salida.

Se utilizó la curva ROC para evaluar cada clase y visualizar la precisión del modelo.

3.3 Conjunto de Datos Wine

Para el conjunto de datos Wine (clasificación de 3 tipos de vino, con 13 características), también se entrenaron dos modelos:

- **Modelo 1**: Tres capas, con $ReLU$ y *softmax* en la capa de salida.
- **Modelo 2**: Cinco capas ocultas con *sigmoid* y una capa de salida de 3 neuronas con *softmax*.

Cada modelo se evaluó mediante precisión y curva ROC, observando los resultados de predicción de clases de vino.

4 Resultados y Visualización

Los resultados obtenidos muestran la precisión antes y después del entrenamiento, con una mejora significativa en cada modelo. Además, las matrices de confusión y las curvas ROC permiten analizar el rendimiento de los modelos en cada clase, resaltando fortalezas y áreas de mejora.

4.1 Test del Modelo Iris

En este análisis, se han implementado tres modelos de red neuronal con diferentes arquitecturas y configuraciones de capas para evaluar su capacidad de clasificación. El primer modelo, diseñado con una arquitectura simple,. El segundo modelo, que incrementa la complejidad al introducir múltiples capas ocultas. Finalmente, el tercer modelo, que continúa añadiendo capas, sugiriendo que la sobrecomplicación de la red puede llevar a un ligero descenso en la capacidad de generalización.

Modelo 1:

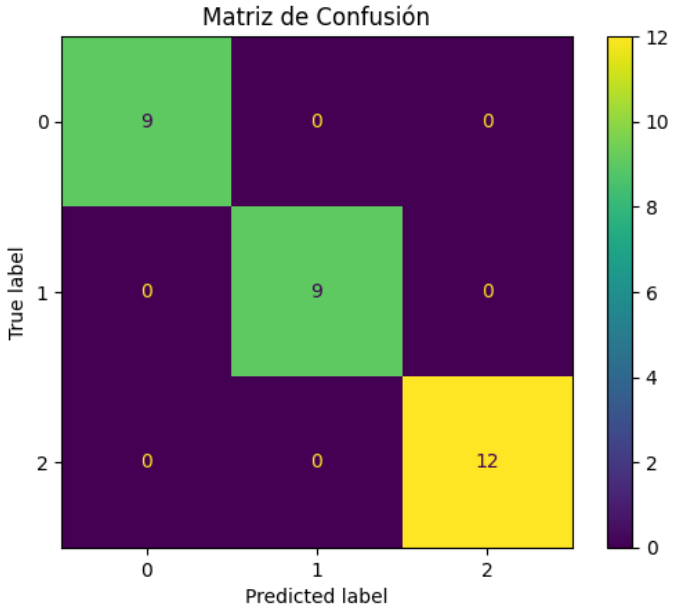


Figure 13: Model 1

Modelo 2:

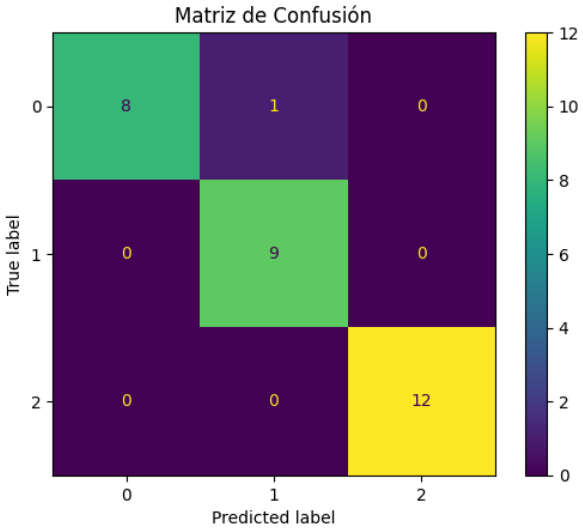


Figure 14: Model 2

Modelo 3:

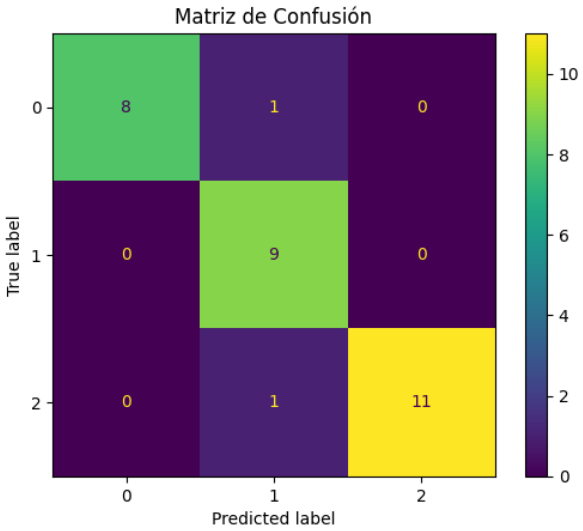


Figure 15: Model 3

Los resultados del entrenamiento de los modelos en el conjunto de datos Iris evidencian la relación entre la complejidad del modelo y su rendimiento en términos de precisión. El primer modelo, que presenta una arquitectura sencilla con solo una capa oculta de 5 neuronas y una función de activación ReLU, logró una precisión del 100%. Este resultado sugiere que incluso las configuraciones menos complejas pueden alcanzar un rendimiento óptimo en conjuntos de datos bien estructurados y menos propensos a la sobreajuste, como es el caso del conjunto de datos Iris.

En contraste, el segundo modelo, que incorpora una mayor complejidad al tener múltiples capas ocultas, obtuvo una precisión de 96%. Aunque esta cifra es aún alta, sugiere que agregar más capas y neuronas puede, en algunos casos, resultar en un leve descenso en el rendimiento, probablemente debido a la introducción de más parámetros y la posibilidad de sobreajuste.

El tercer modelo, que implementa una red aún más profunda al añadir más capas, alcanzó una precisión del 93%. Este ligero descenso en la precisión, en comparación con los modelos anteriores, refuerza la idea de que, a medida que aumentamos la complejidad de la red, el riesgo de sobreajuste puede incrementar, especialmente en conjuntos de datos más pequeños como Iris.

Estos resultados subrayan la importancia de encontrar un equilibrio entre la complejidad del modelo y la capacidad de generalización. Mientras que un modelo más simple puede capturar adecuadamente las relaciones subyacentes en los datos, modelos más complejos pueden no necesariamente traducirse en mejoras significativas en la precisión, y pueden requerir ajustes adicionales y técnicas de regularización para optimizar su rendimiento. En conclusión, se debe evaluar cuidadosamente la arquitectura de la red neuronal en función del conjunto de datos y del problema específico para lograr resultados óptimos.

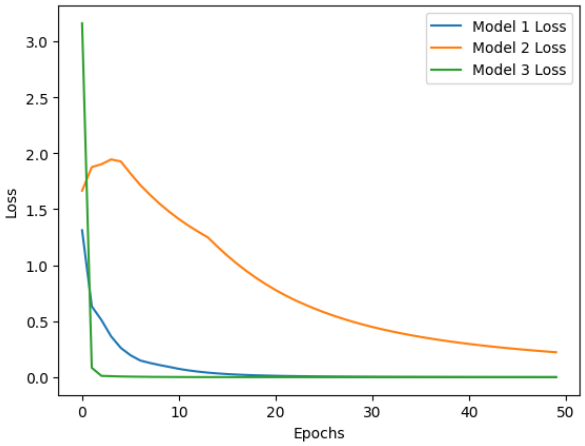


Figure 16: Loss

4.2 Test del Modelo Digits

En este análisis, se desarrollan y comparan tres modelos de red neuronal con diferentes arquitecturas para optimizar la precisión en la clasificación. El primer modelo, diseñado para ser compacto, logra una precisión aceptable, pero su simplicidad limita su capacidad para capturar la complejidad de los datos. El segundo modelo, con una estructura más profunda y múltiples capas ocultas, muestra una mejora significativa en la precisión, sugiriendo que ha aprendido patrones más complejos. Sin embargo, un tercer modelo con mayor complejidad no logra una mejora en la precisión esperada, lo que plantea la necesidad de un equilibrio entre la complejidad de la arquitectura y la capacidad de generalización.

Modelo 1:

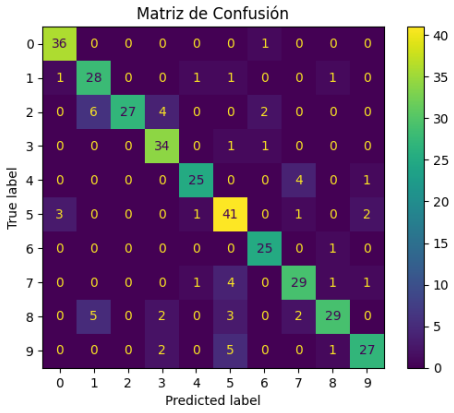


Figure 17: Model 1

Modelo 2:

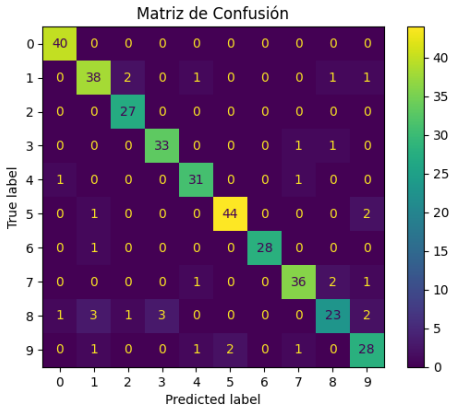


Figure 18: Model 2

En el análisis de los tres modelos de red neuronal aplicados al conjunto de datos Digits, se observó que la complejidad de la arquitectura no siempre se traduce en una mejor precisión. El primer modelo, aunque simple, alcanzó una precisión aceptable, mientras que el segundo modelo, con su diseño más complejo, mejoró notablemente la precisión, demostrando la capacidad de la red para aprender patrones más elaborados en los datos. Sin embargo, el tercer modelo, a pesar de su mayor complejidad, obtuvo muy poca precisión.

Estos resultados destacan la importancia de una adecuada selección de la arquitectura de la red y la necesidad de realizar pruebas y ajustes meticulosos para encontrar un equilibrio óptimo entre la complejidad del modelo y su capacidad de generalización. En última instancia, el éxito en la clasificación de dígitos manuscritos depende de una combinación de factores, incluida la arquitectura, las funciones de activación y el proceso de optimización, lo que requiere un enfoque experimentado para mejorar continuamente el rendimiento.

4.3 Test del Modelo Wine

En este estudio, se han implementado dos modelos de red neuronal para evaluar su capacidad de clasificación. El primer modelo logró una precisión sólida, indicando una adecuada arquitectura y elección de funciones de activación. A partir de estos resultados, se diseñó un segundo modelo más complejo con el objetivo de mejorar el rendimiento.

Modelo 1:

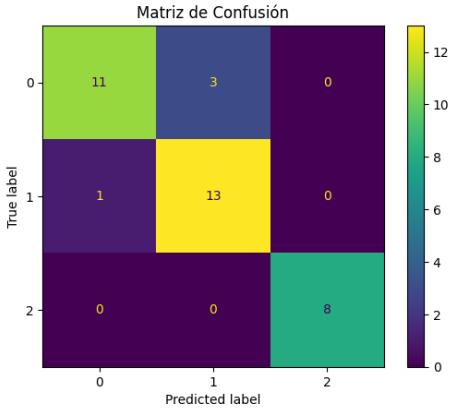


Figure 19: Model 1

Modelo 2:

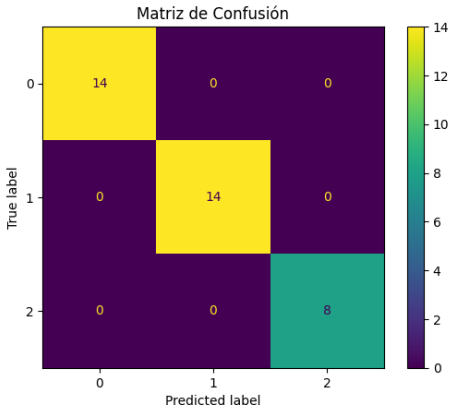


Figure 20: Model 2

Los resultados obtenidos de los modelos de red neuronal aplicados al conjunto de datos Wine destacan la efectividad de las configuraciones elegidas. El modelo nn alcanzó una precisión sólida en la clasificación, lo que sugiere que su arquitectura y funciones de activación fueron adecuadas para el problema. Sin embargo, el modelo nn2, con su arquitectura más compleja de cinco capas y el uso de la función de activación sigmoid, logró una impresionante precisión del 100 por ciento. Este resultado indica que el modelo pudo aprender eficazmente las complejas relaciones entre las características químicas de los vinos y sus clases correspondientes. A pesar del rendimiento sobresaliente del modelo nn2, aún hay margen para la mejora.

5 Conclusión

Este estudio se centró en la implementación y evaluación de redes neuronales para la clasificación de tres conjuntos de datos: Iris, Digits y Wine. A lo largo del trabajo, se desarrollaron diversos modelos con distintas arquitecturas y funciones de activación, analizando su rendimiento en términos de precisión y capacidad de generalización.

Los resultados revelaron que un equilibrio adecuado entre la complejidad del modelo y su rendimiento es esencial. En el caso del conjunto de datos Iris, modelos más simples lograron altas precisiones, destacando que una arquitectura menos compleja puede ser efectiva en conjuntos de datos bien estructurados. Sin embargo, la introducción de más capas y parámetros en los modelos más complejos no siempre se tradujo en mejoras en la precisión, lo que sugiere un riesgo de sobreajuste.

En el análisis del conjunto de datos Digits, se observó que la complejidad de la arquitectura puede permitir que los modelos aprendan patrones más sofisticados. No obstante, también se evidenció que una mayor complejidad no garantiza una mejora en la precisión, lo que pone de relieve la importancia de la selección y ajuste de hiperparámetros adecuados.

Finalmente, en el conjunto de datos Wine, los modelos demostraron una notable capacidad de clasificación, y el modelo más complejo alcanzó un rendimiento sobresaliente. Esto enfatiza el papel crucial de la arquitectura y las funciones de activación en el aprendizaje efectivo de relaciones complejas entre las características de los datos.

En conjunto, estos hallazgos subrayan la necesidad de una estrategia reflexiva en el diseño de modelos de redes neuronales, donde la arquitectura debe ser elegida en función de las características específicas del conjunto de datos y del problema a resolver. Este enfoque permite optimizar la precisión y la capacidad de generalización de los modelos, contribuyendo así a un avance significativo en el campo del aprendizaje automático.

article hyperref

References

[1] GitHub Repository: ModularNNEngine.

[2] OpenAI. "OpenAI: AI Research and Deployment." https://www.openai.com.

[3] Google DeepMind. "Google: Gemini AI Model." https://deepmind.google/technologies/gemini/.

[4] SciTools Iris Documentation. https://scitools-iris.readthedocs.io/en/stable/.

[5] Scikit-learn Wine Dataset Documentation. https://scikit-learn.org/1.5/modules/generated/sklearn.datasets/load$_{w}ine.html$.

[6] Scikit-learn Digits Dataset Documentation. https://scikit-learn.org/stable/modules/generated/sklearn.datasets/load$_{d}igits.html$.

[7] Stack Overflow. https://stackoverflow.com.