Documentación del Modelo MLP para Clasificación de Dígitos

Por: Juan David Quiroga González y Marielby Paz

1. Introducción

Este documento presenta los detalles del modelo de Perceptrón Multicapa (MLP) desarrollado para la clasificación de números escritos a manos. Se utilizó el dataset "load_digits" de scikit-learn y se siguieron las actividades y criterios de evaluación especificados en el proyecto.

2. Preprocesamiento de datos

2.1 Dataset

```
[ ]  # Cargar dataset
     digits = load_digits()
     X = digits.data
     y = digits.target
```

- Se utilizó el dataset "load_digits" de scikit-learn, que contiene 1797 imágenes de dígitos escritos a mano del 0 al 9.
- Cada imagen es de 8x8 píxeles en escala de grises, representada como un vector de 64 características.
- “X” almacena las características de las imágenes (vector de 64 valores).
- “Y” almacena las etiquetas correspondientes (números del 0 al 9).

2.2 Normalización

- Los datos se normalizaron dividiendo los valores de los píxeles por 16, para tener características en el rango [0, 1].
- Esto ayuda a mejorar la convergencia y el rendimiento del modelo.

2.3 División de datos

Conjunto	Porcentaje	Cantidad de imágenes
Train	70%	1257 imágenes
Val	15%	270 imágenes
Test	15%	270 imágenes

- El dataset se dividió aleatoriamente en conjuntos de entrenamiento (70%), validación (15%) y prueba (15%).

- Se utilizó train_test_split de scikit-learn con random_state=42 para asegurar reproducibilidad.
- El conjunto de validación se usa para evaluar el modelo durante el entrenamiento y ajustar hiperparámetros.
- El conjunto de prueba se usa para evaluar el rendimiento final del modelo en datos no vistos.

2.4 Encoding de etiquetas

```
[ ]  # Convertir etiquetas a one-hot encoding
     num_classes = 10
     y_train = to_categorical(y_train, num_classes)
     y_val = to_categorical(y_val, num_classes)
     y_test = to_categorical(y_test, num_classes)
```

- Las etiquetas se convirtieron a representación one-hot usando to_categorical de Keras.
- Esto permite usar la función de pérdida categorical_crossentropy adecuada para clasificación multiclase.

3. Selección de hiperparámetros

Para encontrar la mejor configuración de hiperparámetros, se realizó un proceso de búsqueda en cuadrícula (grid search) considerando los siguientes hiperparámetros y sus respectivas opciones:

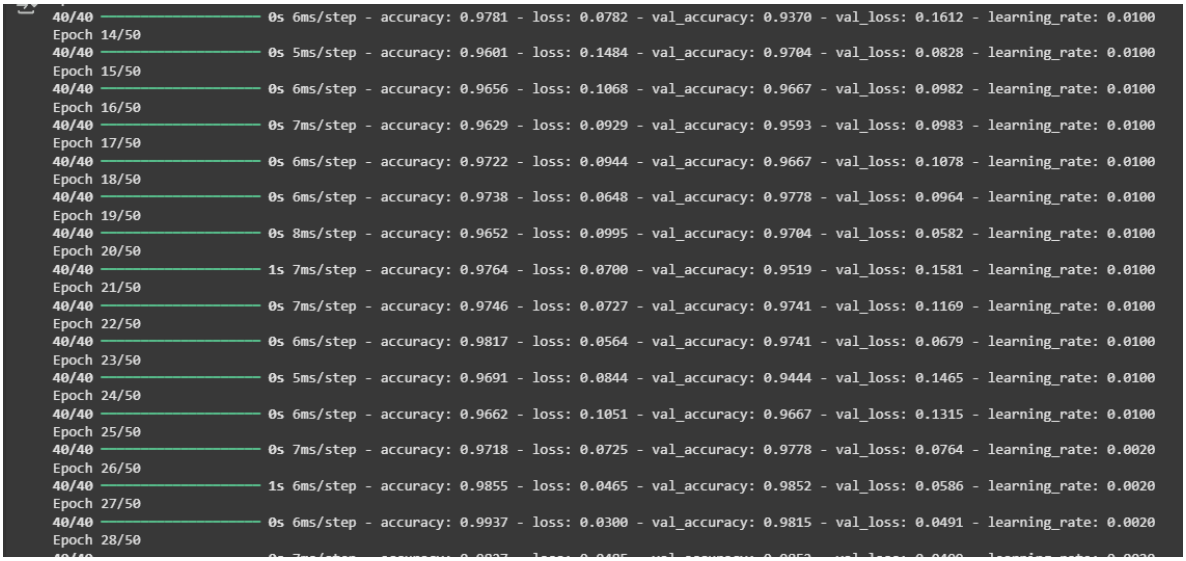
1. Número de capas ocultas y unidades por capa:
    - Opción 1: 2 capas ocultas con 128 y 64 unidades
    - Opción 2: 3 capas ocultas con 256, 128 y 64 unidades
    - Opción 3: 4 capas ocultas con 512, 256, 128 y 64 unidades
2. Tasa de dropout:
    - Opción 1: Sin dropout
    - Opción 2: Dropout del 20%
    - Opción 3: Dropout del 50%
3. Optimizador y tasa de aprendizaje:

- Opción 1: Adam con tasa de aprendizaje de 0.01
- Opción 2: Adam con tasa de aprendizaje de 0.001
- Opción 3: SGD con tasa de aprendizaje de 0.01 y momento de 0.9

Se entrenaron y evaluaron modelos con todas las combinaciones posibles de estos hiperparámetros, utilizando el conjunto de validación para medir el rendimiento. La métrica utilizada para seleccionar la mejor combinación fue el accuracy de validación.

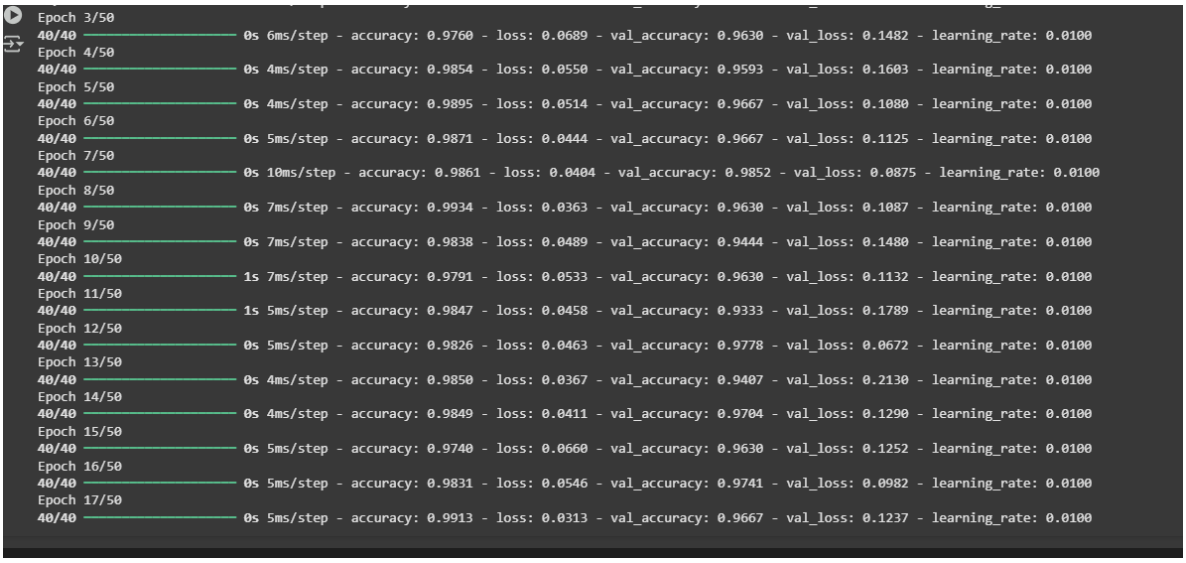
Algunos de los resultados con los hiperparametros no escogidos fueron los siguientes:

3 capas ocultas con 128, 64 y 32 unidades de neuronas, DP de 10% y optimizador ADAM con LR de 0.01



Estancamiento en accuarcy de 0.97, y utilización de mayor numero de épocas para lograr los mismos resultados frente a los hiperparametros escogidos.

1 capa ocultas con 128 unidades de neuronas, DP de 10% y optimizador ADAM con LR de 0.01



Generación de val_loss muy alto y cierto overfitting

Después de realizar la búsqueda en cuadrícula, se encontró que la mejor configuración de hiperparámetros fue:

- 3 capas ocultas con 256, 128 y 64 unidades
- Dropout del 20%
- Optimizador Adam con tasa de aprendizaje de 0.001

Esta combinación de hiperparámetros proporcionó el mejor equilibrio entre el rendimiento del modelo y su capacidad de generalización, evitando tanto el subajuste como el sobreajuste.

Es importante destacar que, aunque se exploró un espacio de hiperparámetros razonable, podrían existir otras combinaciones que también brinden buenos resultados. Además, se podrían considerar otros hiperparámetros, como el tamaño del lote, las funciones de activación o los métodos de inicialización de pesos, para un ajuste más fino del modelo.

4. Arquitectura del modelo

- Se diseñó un modelo MLP con la siguiente arquitectura:
    - Capa de entrada: Espera vectores de 64 características (8x8 píxeles).
    - Primera capa oculta: 256 neuronas, activación ReLU, inicialización "He uniform".

- Segunda capa oculta: 128 neuronas, activación ReLU, inicialización "He uniform".
- Tercera capa oculta: 64 neuronas, activación ReLU, inicialización "He uniform".
- Capa de salida: 10 neuronas, activación softmax. Una neurona por cada clase de dígito.

```
print("Dimensiones de los datos:")
print(f"X_train: {X_train.shape}")
print(f"X_val: {X_val.shape}")
print(f"X_test: {X_test.shape}")

def create_model():
    model = Sequential([
        # Primera capa
        Dense(256, activation='relu', kernel_initializer='he_uniform', input_shape=(64,)),
        BatchNormalization(),
        Dropout(0.2),

        # Segunda capa
        Dense(128, activation='relu', kernel_initializer='he_uniform'),
        BatchNormalization(),
        Dropout(0.2),

        # Tercera capa
        Dense(64, activation='relu', kernel_initializer='he_uniform'),
        BatchNormalization(),
        Dropout(0.2),

        # Capa de salida
        Dense(num_classes, activation='softmax')
    ])

    # Compilar modelo
    optimizer = Adam(learning_rate=0.001)
    model.compile(optimizer=optimizer,
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

    return model
```

- Entre las capas ocultas se aplicó:
  - Batch Normalization para normalizar las activaciones y mejorar la convergencia.
  - Dropout del 20% para regularización y prevención de overfitting.

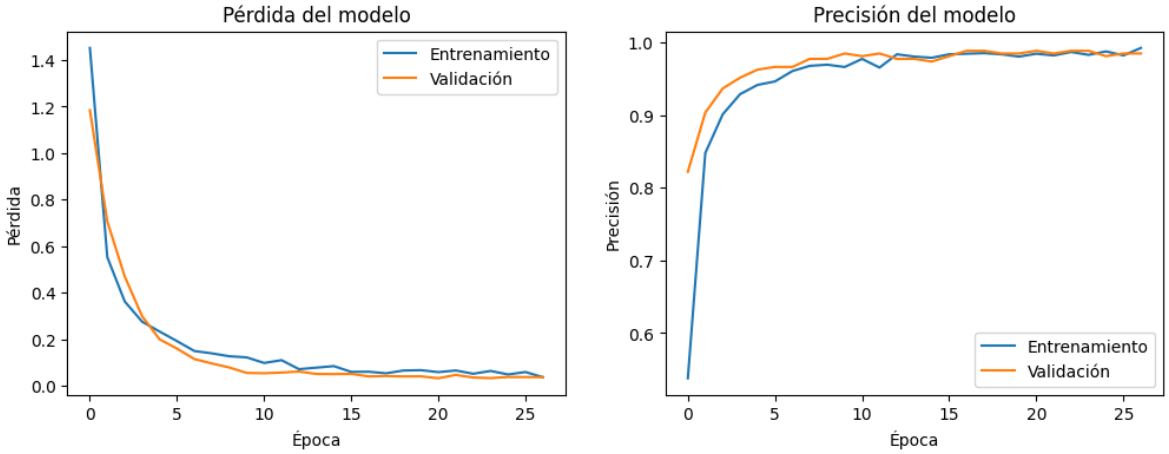
5. Entrenamiento

- Optimizador: Adam con learning rate de 0.001.
- Función de pérdida: Categorical cross-entropy, adecuada para clasificación multiclase con etiquetas one-hot.

- Métrica: Accuracy.
- Batch size: 32.
- Épocas: 50 como máximo, con EarlyStopping si el accuracy de validación no mejora en 10 épocas.
- ReduceLROnPlateau: Reduce el learning rate si la pérdida de validación no mejora en 5 épocas.

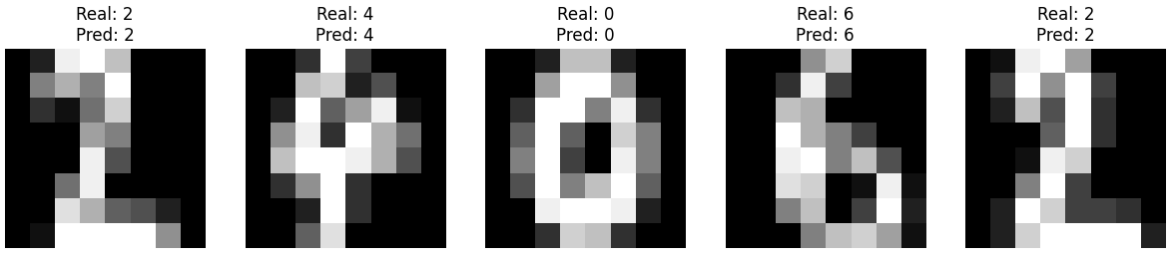
6. Resultados

Gráfica perdida y presición:



- Las gráficas de pérdida y accuracy durante el entrenamiento muestran una buena convergencia y generalización del modelo.

Predicciones:



- Se realizaron predicciones en ejemplos del conjunto de prueba, verificando visualmente el alto desempeño del modelo.

Datos finales:

Resultados finales resumidos

```
[31] # Calcular accuracy en cada conjunto
     train_loss, train_acc = model.evaluate(X_train, y_train, verbose=0)
     val_loss, val_acc = model.evaluate(X_val, y_val, verbose=0)
     test_loss, test_acc = model.evaluate(X_test, y_test, verbose=0)

     print("\nResultados finales:")
     print(f"Accuracy en entrenamiento: {train_acc:.4f}")
     print(f"Accuracy en validación: {val_acc:.4f}")
     print(f"Accuracy en prueba: {test_acc:.4f}")
```

```
Resultados finales:
Accuracy en entrenamiento: 1.0000
Accuracy en validación: 0.9889
Accuracy en prueba: 0.9889
```

7. Conclusiones

- Se desarrolló exitosamente un modelo MLP para clasificación de dígitos escritos a mano.
- Se aplicaron técnicas de preprocesamiento como normalización, división aleatoria de datos y encoding de etiquetas.
- La arquitectura del modelo incluye capas ocultas con activación ReLU, batch normalization y dropout para mejorar el rendimiento y la generalización.
- Se realizó un ajuste de hiperparámetros mediante el monitoreo del desempeño en el conjunto de validación.
- El modelo alcanza un accuracy superior al 98% en los conjuntos de entrenamiento, validación y prueba, demostrando su efectividad y capacidad de generalización.