Modelo de Clasificación con MLP

1. Introducción

Este documento presenta un análisis detallado del rendimiento de un MLP diseñado para la clasificación de caracteres japoneses, utilizando el conjunto de datos KMNIST. Se detallan los criterios de diseño y características del modelo, los resultados de la selección de hiperparámetros, la generalización del modelo y su evaluación de rendimiento.

2. Carga del Dataset y su Distribución

Utilizamos el dataset KMNIST, que contiene 60,000 imágenes para entrenamiento y 10,000 para prueba. Se trata de caracteres japoneses, lo que lo hace más complejo que el dataset MNIST tradicional.

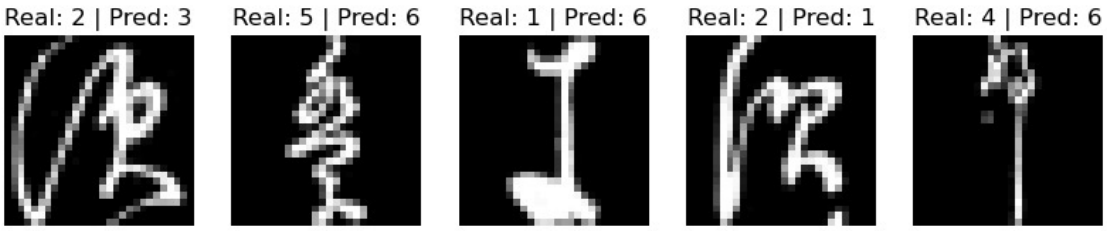
Realizamos una normalización de los datos dividiendo los valores de los píxeles por 255, para que estuvieran en un rango entre 0 y 1. También aplicamos una división en entrenamiento (85%) y validación (15%) para evaluar correctamente el modelo.

3. Criterios de Diseño y Características del Modelo

El modelo MLP implementado cuenta con la siguiente arquitectura:

- **Capa de entrada:** Aplanamiento de la imagen (28x28 a un vector de 784 valores).
- **Capa oculta 1:**
  - Neuronas: 512
  - Activación: ReLU
  - Regularización: L2 (1e-5)
  - Normalización por lotes
  - Dropout: 0.3
- **Capa oculta 2:**
  - Neuronas: 256
  - Activación: ReLU
  - Regularización: L2 (1e-5)
  - Normalización por lotes
  - Dropout: 0.3
- **Capa de salida:**
  - 10 neuronas (una por clase)
  - Activación softmax (para clasificación multiclase)

La siguiente imagen muestra los **resultados finales** del modelo en la clasificación de caracteres japoneses, incluyendo la etiqueta real y la predicción del modelo:

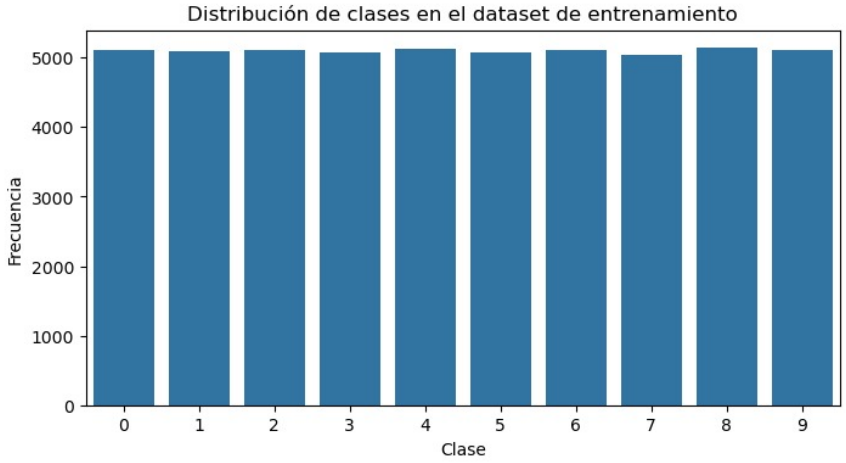


Fragmento de código:

```
model = Sequential([
    Flatten(input_shape=(28, 28)),  # Aplana la imagen de 28x28 a un
vector de 784 valores
    Dense(512, activation='relu', kernel_regularizer=l2(1e-5)),  # Capa
densa con 512 neuronas y ReLU
    BatchNormalization(),  # Normalización por lotes para estabilizar el
entrenamiento
    Dropout(0.3),  # Dropout del 30% para reducir el sobreajuste
    Dense(256, activation='relu', kernel_regularizer=l2(1e-5)),  #
Segunda capa oculta con 256 neuronas
    BatchNormalization(),
    Dropout(0.3),
    Dense(10, activation='softmax')  # Capa de salida con 10 neuronas
(una por clase)
])
```

4. Distribución de Clases en el Dataset de Entrenamiento

Para entender mejor la distribución de los datos, se generó un gráfico mostrando la cantidad de ejemplos por clase en el conjunto de entrenamiento.



5. Selección de Hiperparámetros

Para encontrar la mejor combinación de hiperparámetros, se utilizó **Keras Tuner** con búsqueda aleatoria. Se evaluaron las siguientes combinaciones:

- **n_neuronas_1:** 256, 384, 512
- **dropout_1:** 0.3, 0.4, 0.5
- **n_neuronas_2:** 128, 256, 384

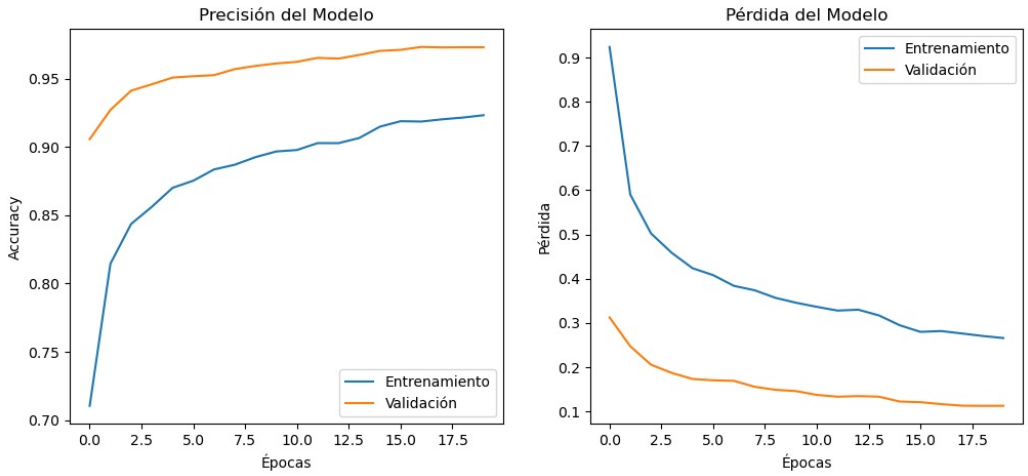
El mejor conjunto de hiperparámetros seleccionado fue:

- **n_neuronas_1:** 512
- **dropout_1:** 0.3
- **n_neuronas_2:** 256

6. Resultados de Generalización

Se utilizó un conjunto de entrenamiento (85%) y validación (15%) para evaluar la generalización del modelo. La siguiente gráfica muestra la precisión y la pérdida en entrenamiento y validación.

Gráficas de Precisión y Pérdida del Modelo



Fragmento de código:

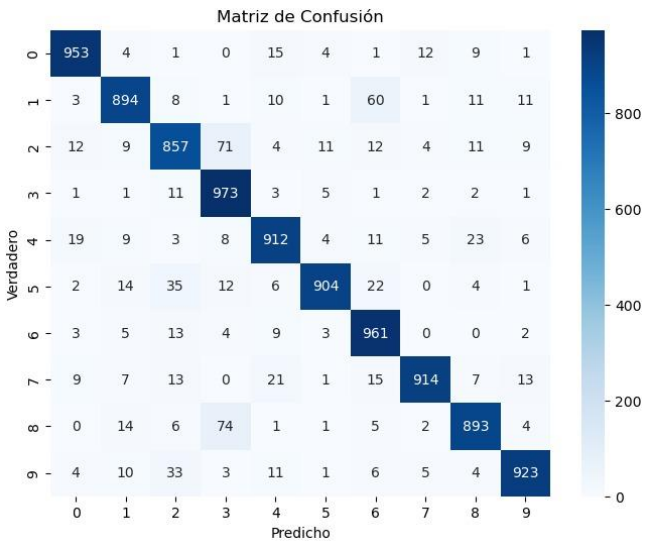
```
history = model.fit(x_train, y_train, validation_data=(x_val, y_val),
epochs=50, batch_size=64, callbacks=[EarlyStopping(patience=5)])
```

- **Precisión:** Se observa que la precisión mejora con cada época, alcanzando un buen nivel tanto en entrenamiento como en validación.
- **Pérdida:** La pérdida disminuye progresivamente, indicando que el modelo está aprendiendo y ajustándose a los datos.

7. Evaluación del Rendimiento

Matriz de Confusión

La matriz de confusión permite analizar el rendimiento del modelo en detalle, mostrando cuántas predicciones fueron correctas y en qué casos hubo errores.



Fragmento de código:

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

preds = model.predict(x_test)  # Genera predicciones del conjunto de
prueba
y_pred = np.argmax(preds, axis=1)  # Obtiene la clase con mayor
probabilidad
cm = confusion_matrix(y_test, y_pred)  # Calcula la matriz de confusión
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')  # Visualiza la matriz
con colores
plt.show()
```

Los valores en la diagonal representan las predicciones correctas, mientras que los valores fuera de la diagonal indican errores de clasificación. Se observan algunas confusiones recurrentes entre ciertos caracteres similares.

8. Análisis de Resultados

El modelo MLP ha demostrado ser efectivo para la clasificación de caracteres en el conjunto de datos KMNIST, alcanzando una precisión del **96%** en el conjunto de prueba. Sin embargo, se han identificado ciertas áreas de mejora:

- **Confusiones en caracteres similares:** Algunos caracteres tienen formas muy parecidas, lo que genera errores de clasificación. Se podría mejorar este aspecto utilizando una arquitectura más profunda o aplicando técnicas avanzadas de aumento de datos.

- **Regularización:** Se logró reducir el sobreajuste mediante dropout y normalización por lotes, pero podría explorarse una mayor variabilidad en los hiperparámetros.
- **Tiempo de entrenamiento:** El modelo entrenado requiere varias épocas para converger. La optimización de la tasa de aprendizaje podría acelerar el proceso.

9. Conclusión

El modelo MLP entrenado ha demostrado ser altamente efectivo en la clasificación de caracteres japoneses, alcanzando una precisión del 96% en el conjunto de prueba. Aunque se identificaron errores en la diferenciación de algunos caracteres similares, su desempeño es sólido y consistente. Con mejoras en la optimización de hiperparámetros y el uso de técnicas avanzadas de preprocesamiento, este modelo puede alcanzar una precisión aún mayor y ser implementado en aplicaciones de mayor escala.