Documentación Técnica de Mejoras al Sistema de Reconocimiento de Pokémon

Equipo de Desarrollo

9 de mayo de 2025

Índice

1. Introducción

Este documento detalla las mejoras implementadas en el sistema de reconocimiento de Pokémon, incluyendo evaluación avanzada de modelos, ajuste de hiperparámetros y visualizaciones comparativas. Cada funcionalidad se describe en tres aspectos: propósito, implementación técnica y uso práctico.

2. Evaluación Avanzada del Modelo

2.1. Propósito

Ampliar las métricas de evaluación para incluir:

- Análisis detallado por clase (precisión, recall, F1-score)
- Visualización interactiva de la matriz de confusión
- Diagnóstico del rendimiento en datos de prueba

2.2. Implementación Técnica

Se integraron las siguientes bibliotecas:

```
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns
```

El método de evaluación se modificó para:

1. Calcular predicciones en lote
2. Generar reporte de clasificación
3. Visualizar matriz de confusión con Seaborn

Código clave actualizado:

```
def evaluate_model(self, show_matrix=True):
    # [...] Carga de datos
    print(classification_report(all_labels, all_preds,
          target_names=list(self.idx_to_name.values())))

    if show_matrix:
        plt.figure(figsize=(15, 12))
        sns.heatmap(cm, annot=False, fmt='d',
                    xticklabels=self.idx_to_name.values(),
                    yticklabels=self.idx_to_name.values())
        plt.title('Matriz de Confusion Interactiva')
        plt.show()
```

2.3. Uso Práctico

- Presionar `m` durante la ejecución
- Genera dos salidas:
  1. Reporte textual con métricas por clase (Figura **??**)
  2. Matriz de confusión visual (Figura **??**)

3. Ajuste de Hiperparámetros

3.1. Arquitectura de Implementación

Se añadió un diccionario para gestión dinámica de parámetros:

```
self.hyperparams = {
    'lr': 0.001,         # Tasa de aprendizaje
    'batch_size': 8,      # Ejemplos por lote
    'epochs': 5           # Ciclos de entrenamiento
}
```

El método de entrenamiento fue modificado para soportar:

- Sobreescritura de parámetros
- Reconfiguración dinámica del DataLoader
- Actualización del optimizador

3.2. Flujo de Ajuste

1. Presionar `h` en el menú principal
2. Ingresar nuevos valores mediante consola:

```
    === AJUSTE DE HIPERPARAMETROS ===
    Nueva tasa de aprendizaje (ej: 0.001): 0.0005
    Nuevo tamano de batch (ej: 16): 32
    Nuevo numero de epocas (ej: 10): 15

```

3. Los cambios se aplican al próximo entrenamiento

4. Visualización Comparativa

4.1. Mecanismo de Comparación

Se implementó un sistema de historial que almacena:

$$Historial = \{loss : [valores], accuracy : [valores]\} \tag{1}$$

Método de comparación:

```python
def compare_models(self, history_list, labels):
    plt.figure(figsize=(12, 5))
    # Subgr fico para p rdida
    plt.subplot(1, 2, 1)
    for history, label in zip(history_list, labels):
        plt.plot(history['loss'], label=label)
    # Subgr fico para precisi n
    plt.subplot(1, 2, 2)
    for history, label in zip(history_list, labels):
        plt.plot(history['accuracy'], label=label)
    plt.show()
```

4.2. Escenarios de Uso

- Comparar diferentes tasas de aprendizaje
- Evaluar impacto del tamaño de lote
- Analizar estabilidad del entrenamiento

5. Mejoras Adicionales

5.1. Soporte Multiplataforma

- Detección automática de GPU:

```
    self.device = torch.device("cuda" if torch.cuda.
    is_available() else "cpu")
    self.model.to(self.device)

```

- Aceleración CUDA para entrenamientos grandes

5.2. Validación de Entradas

Input	Validación
Tasa de aprendizaje	Float positivo
Batch size	Entero mayor a 0
Épocas	Entero entre 1-100

6. Conclusión

Estas mejoras transforman el sistema en una herramienta profesional para experimentación con modelos de visión artificial. La integración de métricas estándar y ajustes dinámicos permite:

- Diagnóstico preciso de sobreajuste
- Optimización sistemática de hiperparámetros

- Comparación cuantitativa entre configuraciones