Tecnológico de Monterrey

Inteligencia artificial avanzada para la ciencia de datos II

Módulo 2 Implementación de un modelo de deep learning

Galo Alejandro Del Río Viggiano A01710791

ÍNDICE

Pokedex Gen1

1. Introducción

1.1 Problema

El problema principal de esta práctica consiste en desarrollar un modelo capaz de reconocer automáticamente a qué Pokémon pertenece una imagen, aun cuando estas varían en color, pose, estilo o calidad. Esto plantea el reto de que el modelo aprenda características visuales realmente representativas de cada especie y no dependa de detalles superficiales.

A través de esta práctica se busca comprender mejor cómo funcionan las redes convolucionales modernas, cómo procesan patrones visuales complejos y qué tan bien pueden generalizar cuando se les entrena con un conjunto de datos grande y diverso.

Además, permite explorar el flujo completo de un proyecto de visión computacional, desde la preparación del dataset hasta la evaluación del rendimiento de un modelo

1.2 Dataset

El dataset que se va usar es Pokemon Images, First Generation(17000 files) de Kaggle, este es especialmente adecuado para la práctica porque reúne una gran cantidad de imágenes de Pokémon de primera generación (3 Gigabytes), organizadas por clases de manera clara, lo que facilita el entrenamiento y la evaluación de un modelo de clasificación.

Otra cualidad importante es la diversidad de estilos y representaciones: hay imágenes con distintos fondos, poses y niveles de detalle e incluso estilos artísticas.

Lo que obliga al modelo a enfocarse en las características realmente distintivas de cada Pokémon y no solo en un contexto específico. Además, al estar centrado solo en la primera generación, el problema se mantiene acotado a un número razonable de clases (151).

2. Transformaciones

2.1 Aumentaciones

Un problema que tiene este dataset es que las imágenes tienen distintos tamaños. Por ello todas las imágenes se redimensionan a 160×160 píxeles con la función resize_160 y se convierten a RGB. Esto sirve para que todo el dataset tenga el mismo tamaño y número de canales (Los canales serán más importantes por el modelo con el que vamos a usar). Además, trabajar con 160×160 reduce el costo computacional respecto a imágenes grandes, pero sigue siendo suficiente para conservar los detalles visuales de los Pokémon.

Sobre la imagen base de 160×160 se aplican transformaciones aleatorias:

- Espejo horizontal (flip): Se invierte la imagen. Esto ayuda a que el modelo no dependa de si el Pokémon está mirando hacia la izquierda o la derecha.

- Rotación leve: Se rota la imagen entre -20 y 20 grados. Esto simula distintas orientaciones y hace al modelo más robusto ante imágenes ligeramente inclinadas.

- Cambios de brillo, contraste y saturación: Esto produce variaciones de iluminación y estilo, para que el modelo aprenda a enfocarse en la forma y no en condiciones específicas de luz o color.

- Desenfoque gaussiano: Se aplica un desenfoque suave. Esto simula imágenes ligeramente fuera de foco o con baja calidad y obliga al modelo a aprender características más robustas.

- Ruido gaussiano: Se añade ruido a los píxeles. Esto imita imágenes con ruido de cámara o compresión y también mejora la robustez del modelo.

Esto se hace para el modelo no aprende solo a clasificar solo las mismas imágenes, sino que este aprenda a generalizar las características de cada pokemon y generalice mejor evitando el overfitting del modelo.

2. Modelo

2.1 Elección del modelo

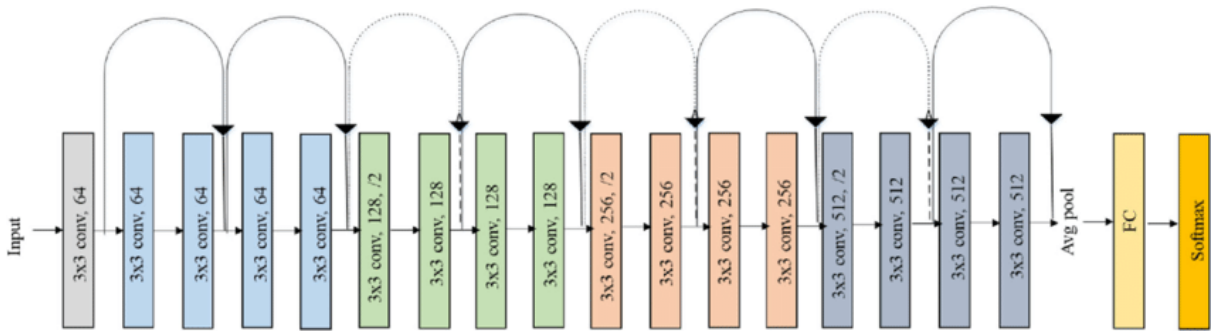
Se eligió ResNet18 porque ofrece un equilibrio ideal entre rendimiento y eficiencia para una tarea de clasificación de imágenes como esta. Es una arquitectura suficientemente profunda para aprender características visuales complejas, pero al mismo tiempo es más ligera y rápida que modelos más grandes como ResNet 50 o Efficient Net, lo que facilita entrenarla incluso en equipos con recursos limitados.

Además, su diseño basado en bloques residuales ayuda a que el entrenamiento sea estable y evita problemas comunes en redes profundas, como el desvanecimiento del gradiente.

2.2 Cómo funciona el modelo

ResNet18 es una red neuronal convolucional que procesa una imagen en varias etapas para extraer patrones cada vez más complejos, desde bordes y colores simples hasta formas y estructuras específicas.

Su particularidad es que utiliza “bloques residuales”: en lugar de solo pasar la información capa por capa, cada bloque suma la entrada original del bloque a la salida de unas pocas capas internas. Esto crea atajos (skip connections) que facilitan el flujo del gradiente durante el entrenamiento y permiten entrenar redes más profundas sin que el modelo “olvide” la información o deje de aprender. Al final, las características extraídas se pasan a una capa totalmente conectada que produce una probabilidad para cada clase, en este caso, para cada Pokémon.



2.3 Glosario

Input: Imagen que entra a la red, ya redimensionada y normalizada (por ejemplo 3×224×224, RGB).

Conv (Convolution / Convolución): Filtro que “escanea” la imagen y aprende a detectar patrones: bordes, texturas, formas, etc.

BN (Batch Normalization): Capa que normaliza las activaciones de la conv (las centra y escala) para que el entrenamiento sea más estable y rápido.

ReLU (Rectified Linear Unit): Función de activación que deja pasar valores positivos y pone en 0 los negativos. Añade no linealidad y ayuda a que el modelo aprenda cosas más complejas.

MaxPooling (Max Pooling): Reduce el tamaño de la imagen tomando el valor máximo en pequeñas ventanas (por ejemplo 2×2).
Resultado: menos tamaño espacial, mantiene la información más importante.

Residual block / Layer1–4 (bloques residuales): Secuencias de convoluciones + BN + ReLU donde la entrada se suma a la salida (skip connection).
Ayuda a entrenar redes más profundas sin que “se muera” el gradiente.

AvgPool / Global Average Pooling (GAP): Hace el promedio de cada mapa de activación completo, dándonos un resumen numérico por canal.

Flatten: Aplasta el tensor (por ejemplo [1, 512, 1, 1]) a un vector ([1, 512]) para poder conectarlo a una capa densa.

FC (Fully Connected / Capa densa):Capa lineal que combina todas las features del vector y produce una salida de tamaño = número de clases (por ejemplo 1000 en ImageNet).

Softmax: Convierte los valores de salida (logits) en probabilidades que suman 1, indicando qué clase es la más probable.

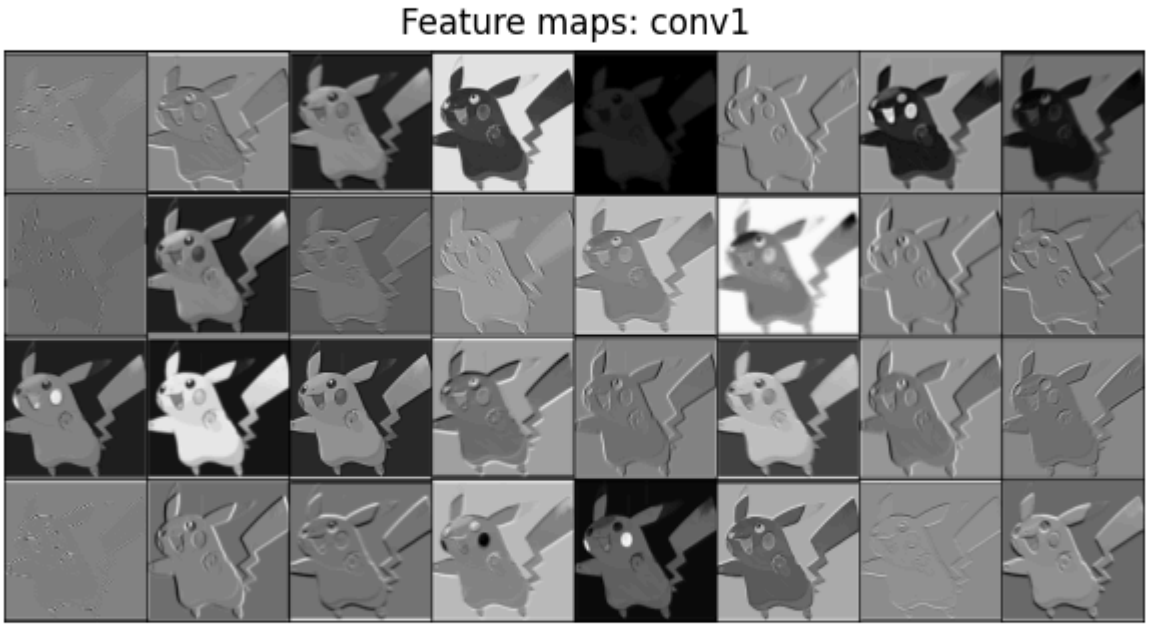
2.4 Paso a Paso

Paso	Shape	Descripción
input	[1, 3, 224, 224]	Imagen de entrada normalizada (RGB)
conv1	[1, 64, 112, 112]	Convolución inicial: extrae bordes y texturas básicas
bn1	[1, 64, 112, 112]	Batch Normalization: estabiliza y acelera el entrenamiento
relu1	[1, 64, 112, 112]	ReLU: activa características no lineales
maxpool	[1, 64, 56, 56]	Max Pooling: reduce tamaño espacial a la mitad
layer1	[1, 64, 56, 56]	Bloques residuales: combinan convoluciones + skip connection
layer2	[1, 128, 28, 28]	Extrae patrones más complejos y duplica canales
layer3	[1, 256, 14, 14]	Aprende partes de objetos (formas más abstractas)
layer4	[1, 512, 7, 7]	Features altamente abstractos (alto nivel)
avgpool (GAP)	[1, 512, 1, 1]	Global Average Pooling: promedia cada mapa de activación
flatten	[1, 512]	Convierte mapas de activación en un vector de características
fc (logits)	[1, 1000]	Capa lineal final que produce logits para clasificación
softmax probs	[1, 1000]	Convierte logits en probabilidades por clase

2.5 Ejemplo visual

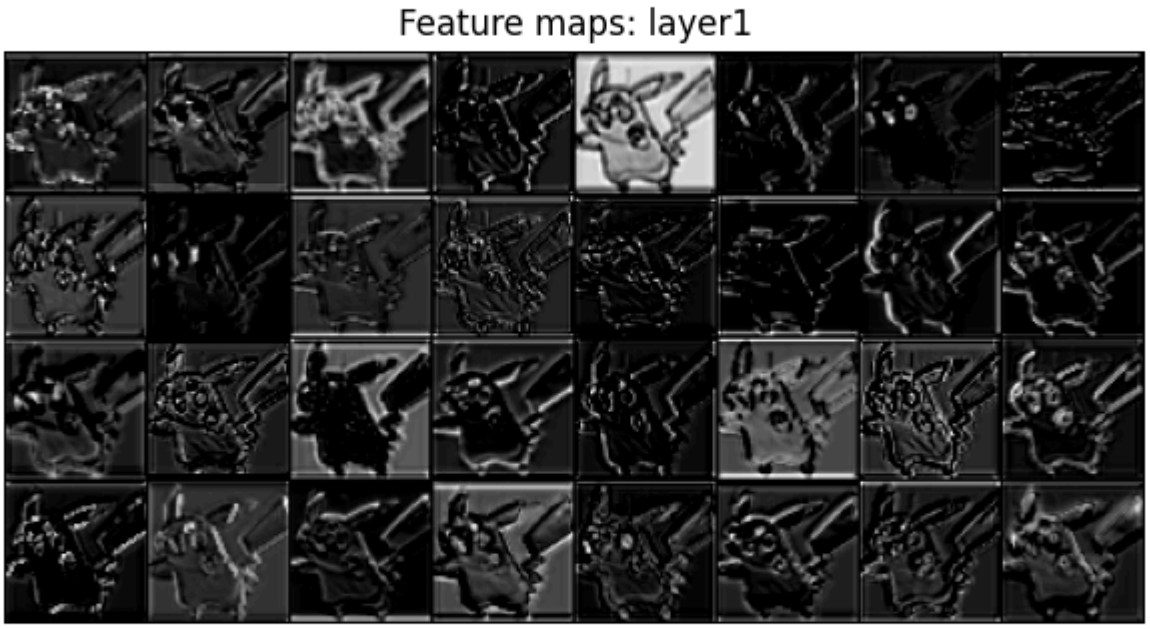
Imagen tal como viene en el dataset, en formato RGB, todavía sin ningún tipo de transformación ni normalización.

La siguiente parte del proceso consiste en aplicarle el preprocesamiento inicial, que incluye redimensionarla al tamaño definido convertirla a tensor, normalizarla con las estadísticas de ImageNet y, en el caso del conjunto de entrenamiento, aplicar aumentaciones como flips, rotaciones ligeras, variaciones de brillo/contraste, desenfoque o ruido; tras estos pasos, la imagen ya transformada se convierte en el input real que entra a ResNet18 con forma [1, 3, H, W], listo para pasar a la primera convolución conv1.



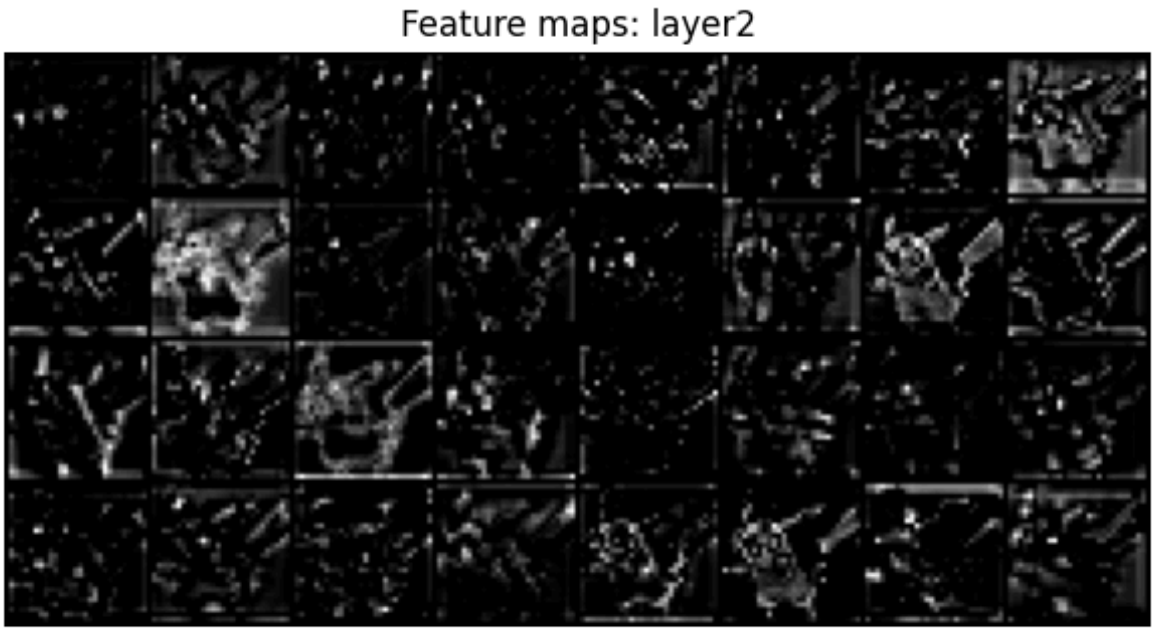
Esta imagen representa la salida de la primera convolución (conv1) de ResNet18, donde se generan los primeros feature maps o mapas de características —normalmente 64 canales— que capturan patrones muy básicos de la imagen como bordes, contornos, contrastes locales y texturas simples; cada recuadro es la respuesta de un filtro distinto aplicado al Pikachu original, mostrando qué partes “activa” cada kernel.

La etapa que sigue después de esto en el flujo del modelo es BatchNorm + ReLU, donde se normalizan estas activaciones y luego se aplica una función no lineal que permite que la red empiece a construir representaciones más complejas a partir de estos detectores iniciales.



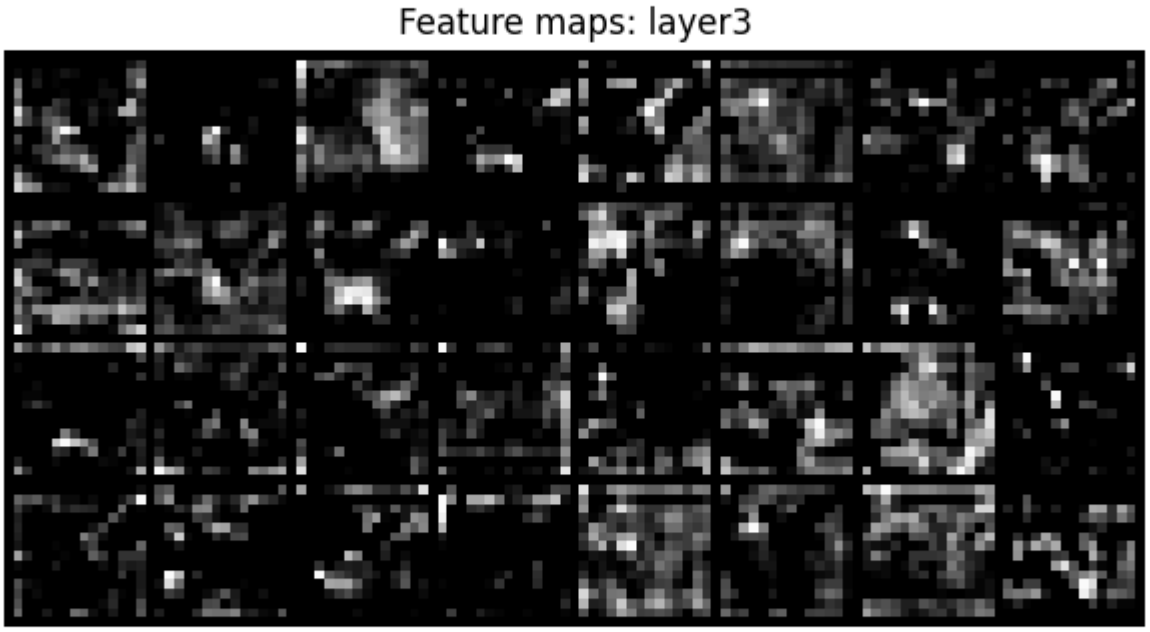
Esta imagen corresponde a la salida de layer1 de ResNet18, el primer bloque residual completo, donde los filtros ya no detectan simples bordes como en conv1, sino patrones un poco más complejos, como contornos combinados, zonas de textura, límites más definidos del cuerpo del Pokémon y transiciones de forma más abstractas; además, el skip connection permite conservar parte de la información original mientras se mezclan nuevas transformaciones, logrando representaciones más ricas y estables.

La etapa que sigue después de layer1 es layer2, donde la red duplica la cantidad de canales (de 64 a 128) y empieza a capturar estructuras más profundas, como partes del Pokémon (orejas, cola, mejillas, siluetas completas) con mayor abstracción.



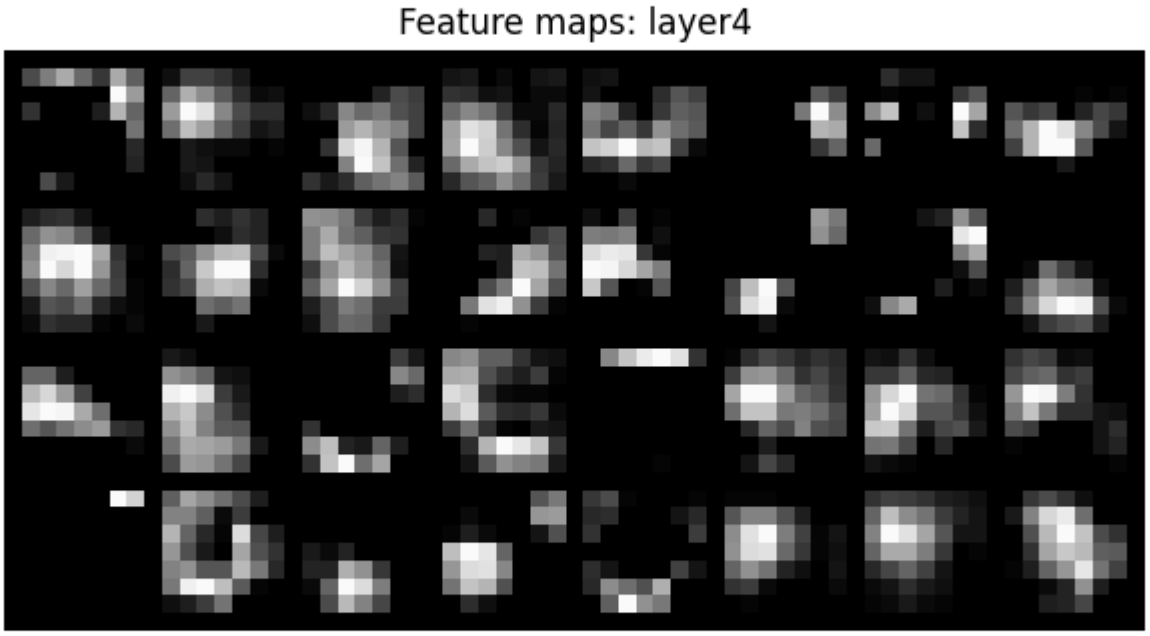
Esta imagen muestra la salida de layer2 de ResNet18, donde la red ya trabaja con 128 canales y comienza a identificar estructuras más complejas y abstractas del Pokémon: no solo bordes o contornos, sino combinaciones de ellos que forman partes más reconocibles, como orejas, ojos, siluetas parciales o patrones de textura característicos. A esta profundidad, muchos filtros responden solo a regiones específicas y dejan gran parte de la imagen en negro, porque cada canal se especializa en detectar patrones muy concretos.

La etapa que sigue a layer2 es layer3, donde se duplican nuevamente los canales (256) y la red pasa a extraer componentes aún más abstractos, como formas completas, regiones texturizadas complejas y patrones que corresponden a partes grandes del Pokémon o distribuciones espaciales más sofisticadas.



Esta imagen corresponde a los feature maps de layer3 de ResNet18, una etapa donde la red ya opera con 256 canales y comienza a representar la imagen en un nivel mucho más abstracto: aquí los filtros dejan de responder a formas simples y se enfocan en partes más grandes y combinaciones complejas de patrones, como regiones completas del cuerpo, distribuciones de textura, zonas de contraste específicas o configuraciones espaciales propias del Pikachu. Por eso los mapas se ven más “borrosos”, fragmentados y con activaciones muy localizadas: cada filtro ahora está altamente especializado y solo se activa ante configuraciones visuales muy concretas.

Después de esta etapa sigue layer4, donde la red aumenta a 512 canales y extrae conceptos de muy alto nivel, casi representaciones semánticas completas del Pokémon, antes de pasar al Global Average Pooling y la capa final de clasificación.



Esta imagen muestra los feature maps de layer4, la etapa más profunda de ResNet18 antes del Global Average Pooling, donde la red trabaja con 512 canales y extrae representaciones altamente abstractas y semánticas del Pokémon. A este nivel, los filtros ya no responden a bordes ni partes individuales, sino a combinaciones complejas que representan conceptos visuales completos, como la silueta global, regiones distintivas del cuerpo, patrones de contraste que caracterizan a Pikachu o configuraciones espaciales muy específicas que lo diferencian de otros Pokémon. Por eso los mapas se ven más pequeños, borrosos y esparcidos: la red está condensando la información relevante en activaciones muy específicas y de bajo detalle espacial, enfocándose solo en lo que realmente define la categoría

. La etapa que sigue después de layer4 es Global Average Pooling (avgpool), donde cada uno de esos 512 mapas se reduce a un único valor, creando un vector compacto de características que luego pasa a la capa totalmente conectada (fc) para generar la predicción final de clase.

3. Código

3.1 Split del Dataset

Separé mi dataset en tres subconjuntos: aproximadamente 70 % de las imágenes se usaron para train, 15 % para validation y 15 % para test, organizadas en carpetas físicas (train, validation, test) con subcarpetas por clase; sobre estas particiones, train aplica las transformaciones con aumentaciones de datos (resize, flips, rotaciones y ligeros cambios de color, más ToTensor y normalización), mientras que validation y test usan solo resize, ToTensor y normalización, sin augmentación, de modo que el modelo aprende con ejemplos más variados en entrenamiento pero se evalúa con imágenes limpias y representativas en validación y prueba.

3.1 Preprocesamiento de datos

En la parte la primera parte del código se define cómo se preparan las imágenes antes de entrar al modelo: tanto para entrenamiento como para validación y prueba, se redimensionan a un tamaño fijo (160×160), se convierten a tensores y se normalizan con las medias y desviaciones estándar de ImageNet para que los valores queden en una escala adecuada para ResNet18.

Después, con build_datasets, se cargan las carpetas train, val y test usando ImageFolder, se obtienen las clases presentes en cada una y se calcula la intersección de clases comunes entre los tres conjuntos; con esta lista común se filtran las muestras de cada split y se remapean los índices de clase, garantizando que train, val y test trabajen exactamente con las mismas clases y con los mismos índices numéricos para cada una.

3.2 DataLoaders

DataLoader es la parte de PyTorch que se encarga de “servirle” los datos al modelo en mini-lotes (batches).

En esta parte el código make_loaders toma el dataset de entrenamiento ya cargado, cuenta cuántas imágenes hay por clase y, si detecta que hay desbalance fuerte (que la clase más grande tiene al menos el doble de ejemplos que la más pequeña), construye

un WeightedRandomSampler que le da más probabilidad de ser muestreadas a las clases raras y menos a las muy frecuentes, para que el modelo vea un entrenamiento más equilibrado.

Además, configura parámetros como batch_size, num_workers y pin_memory (cuando hay GPU) para que la carga de datos sea más rápida y eficiente durante el entrenamiento.

3.2 Modelo

El código construye la red neuronal que se va a entrenar: toma una ResNet18 de touchvision, con pesos pre entrenados en ImageNes (para aprovechar lo que ya "sabe" de millones de imágenes), y luego reemplaza su última capa totalmente conectada (fc) por un pequeño bloque propio que tiene un Dropout (para reducir overfitting) y una capa Lineal que ajusta la salida al número de clases de pokemon.

3.2 Entrenamiento

El entrenamiento principal, se hace con la función train(...) arma todo el flujo para enseñar al modelo. La cual hace lo siguiente:

1. Fija semilla para reproducibilidad.

2. Detecta dispositivo (cuda o cpu).

3. Llama a build_datasets y make_loaders.

4. Crea el modelo ResNet18 y lo pasa a device.

5. Configura:

    a. GradScaler para mixed precisión si hay GPU.
    b. criterion = CrossEntropyLoss.
    c. optimizer = Adam con lr y weight_decay.
    d. scheduler = ReduceLROnPlateau (baja LR cuando no mejora el val_loss).

6. Crea carpeta outdir y guarda class_to_idx.json.

7. Loop de épocas:
    a. Pone model.train().
    b. Para cada batch de train
        i. Pasa por el modelo (con amp.autocast si GPU).
        ii. Calcula loss, backprop con scaler.
        iii. Actualiza parámetros.
        iv. Acumula:
        v. running_loss, running_correct.
        vi. y_true_train y y_pred_train para precisión macro.
    c. Al final de la época calcula:
        i. train_loss
        ii. train_acc
        iii. train_precision_macro.
    d. Llama a evaluate para validación:
        i. val_loss
        ii. val_acc
        iii. val_precision_macro.
    e. El scheduler se ajusta Learning Rate (lr) según valores.
    f. Guarda todo en el dict metrics.
    g. Si val_loss mejora:
        i. Guarda checkpoint en best_model.pth.
8. Terminado el entrenamiento:
    a. Llama a save_curves(metrics, outdir).
    b. Carga el mejor modelo (best_model.pth).
    c. Evalúa en test
        i. test_loss, test_acc, test_precision_macro.
        ii. Imprime matriz de confusión y classification_report.

d. Guarda test_metrics.json con métricas finales.

Con lo anterior podemos asegurar que, al usar una arquitectura como ResNet18, se esta aplicando realmente deep learning, porque se trata de una red neuronal profunda con múltiples capas convolucionales, activaciones no lineales y una capa final totalmente conectada, cuyos millones de parámetros se entrenan de extremo a extremo mediante backpropagation y un optimizador basado en gradientes (Adam) directamente sobre los pixeles de las imágenes, aprendiendo representaciones jerárquicas sin necesidad de diseñar manualmente las características.

4. Interacciones del código

4.1 Primera Iteración

En esta iteración definí los siguientes hiperparámetros, cada uno seleccionado para lograr un balance entre rendimiento, estabilidad y eficiencia durante el entrenamiento del modelo:

img_resize = 160
Elegí este tamaño porque ofrece suficiente detalle visual sin incrementar demasiado el consumo de memoria ni el tiempo de entrenamiento.

batch_size = 64
Lo seleccioné porque da un gradiente estable y, al mismo tiempo, permite aprovechar bien la GPU sin saturar.

epochs = 20
Decidí usar 20 épocas para permitir que el modelo converja correctamente sin extender el entrenamiento más de lo necesario.

learning rate (lr) = 1e-3
Usé este valor porque es un punto de partida sólido para Adam, permitiendo un aprendizaje rápido pero estable.

weight decay (wd) = 1e-4

Lo utilicé para agregar una regularización ligera que ayude a prevenir overfitting sin frenar demasiado el aprendizaje.

pretrained = False
Elegí no usar pesos preentrenados de ImageNet para que el modelo solo entrene con los datos que le proporcione de train

dropout = 0.2
Configuré un dropout moderado en la última capa para ayudar a la generalización sin afectar de manera agresiva la capacidad del modelo.

scheduler: ReduceLROnPlateau (factor = 0.5, patience = 3)
Usé este scheduler porque me permite disminuir el learning rate automáticamente cuando la pérdida de validación deja de mejorar, estabilizando aún más el entrenamiento.

use_sampler_if_imbalance = True
Activé el sampler para manejar el desbalance de clases y asegurar que el modelo no se sesgue hacia las clases más frecuentes.

num_workers = 4
Elegí este valor para acelerar la carga de datos aprovechando múltiplos procesos sin saturar la máquina.

seed = 42
Fijé la semilla para asegurar que los resultados puedan reproducirse exactamente en futuras corridas.

Desempeño por Clase: Precision, Recall y F1-score:

Clase	Precision	Recall	F1-score	Support
Abra	1	0.9787	0.9892	47
Aerodactyl	1	0.9552	0.9771	67
Alakazam	1	0.9275	0.9624	69
Arbok	0.9905	1	0.9952	104
Arcanine	0.9167	1	0.9565	66

Articuno	1	0.973	0.9863	74
Beedrill	0.9859	1	0.9929	70
Bellsprout	0.9813	0.9906	0.9859	106
Blastoise	0.9576	0.9912	0.9741	114
Bulbasaur	1	0.9746	0.9871	118
Butterfree	0.9894	1	0.9947	93
Caterpie	1	0.9905	0.9952	105
Chansey	1	0.9882	0.9941	85
Charizard	0.988	1	0.9939	82
Charmander	1	0.9903	0.9951	103
Charmeleon	1	0.9913	0.9956	115
Clefable	0.9647	1	0.982	82
Clefairy	0.9512	0.9873	0.9689	79
Cloyster	1	1	1	73
Cubone	0.9583	0.9892	0.9735	93
Dewgong	1	0.9545	0.9767	88
Diglett	0.971	1	0.9853	67
Ditto	0.9688	1	0.9841	62
Dodrio	0.9859	0.9859	0.9859	71
Doduo	0.9589	1	0.979	70
Dragonair	0.97	0.9898	0.9798	98
Dragonite	1	0.9576	0.9784	118
Dratini	0.9885	0.9556	0.9718	90
Drowzee	1	0.961	0.9801	77
Dugtrio	1	1	1	91
Eevee	0.9912	0.9739	0.9825	115
Ekans	1	0.9886	0.9943	88
Electabuzz	1	1	1	96
Electrode	0.9592	1	0.9792	47
Exeggcute	0.9694	1	0.9845	95
Exeggutor	1	0.9924	0.9962	132
Farfetchd	1	0.9604	0.9798	101
Fearow	0.9815	0.955	0.968	111
Flareon	0.9924	0.985	0.9887	133
Gastly	1	0.9832	0.9915	119
Gengar	0.9914	0.9914	0.9914	116
Geodude	1	0.9855	0.9927	69

Gloom	0.9828	0.9828	0.9828	58
Golbat	1	0.951	0.9749	102
Goldeen	0.9891	1	0.9945	91
Golduck	1	0.9739	0.9868	115
Graveler	0.9403	1	0.9692	63
Grimer	0.9531	1	0.976	61
Growlithe	1	0.9341	0.9659	91
Gyarados	0.9365	0.9833	0.9593	120
Haunter	1	0.9775	0.9886	89
Hitmonchan	1	1	1	86
Hitmonlee	1	0.9753	0.9875	81
Horsea	0.9901	0.9804	0.9852	102
Hypno	1	1	1	58
Ivysaur	0.9832	0.975	0.9791	120
Jigglypuff	0.9931	0.973	0.9829	148
Jolteon	1	0.99	0.995	100
Jynx	1	0.9851	0.9925	67
Kabutops	0.9875	0.9875	0.9875	80
Kadabra	0.9459	1	0.9722	70
Kakuna	0.9888	0.9888	0.9888	89
Kangaskhan	0.9865	1	0.9932	73
Kingler	0.9756	1	0.9877	80
Koffing	0.9405	0.9875	0.9634	80
Lapras	1	0.9583	0.9787	96
Lickitung	0.9861	0.9595	0.9726	74
Machamp	0.9859	0.9722	0.979	72
Machoke	0.9706	0.9851	0.9778	67
Machop	1	0.9857	0.9928	70
Magikarp	1	1	1	89
Magmar	1	0.971	0.9853	69
Magnemite	1	1	1	76
Magneton	0.962	1	0.9806	76
Mankey	0.96	0.96	0.96	75
Marowak	0.9726	0.9726	0.9726	73
Meowth	1	1	1	66
Metapod	0.9884	1	0.9942	85
Mew	1	0.961	0.9801	77

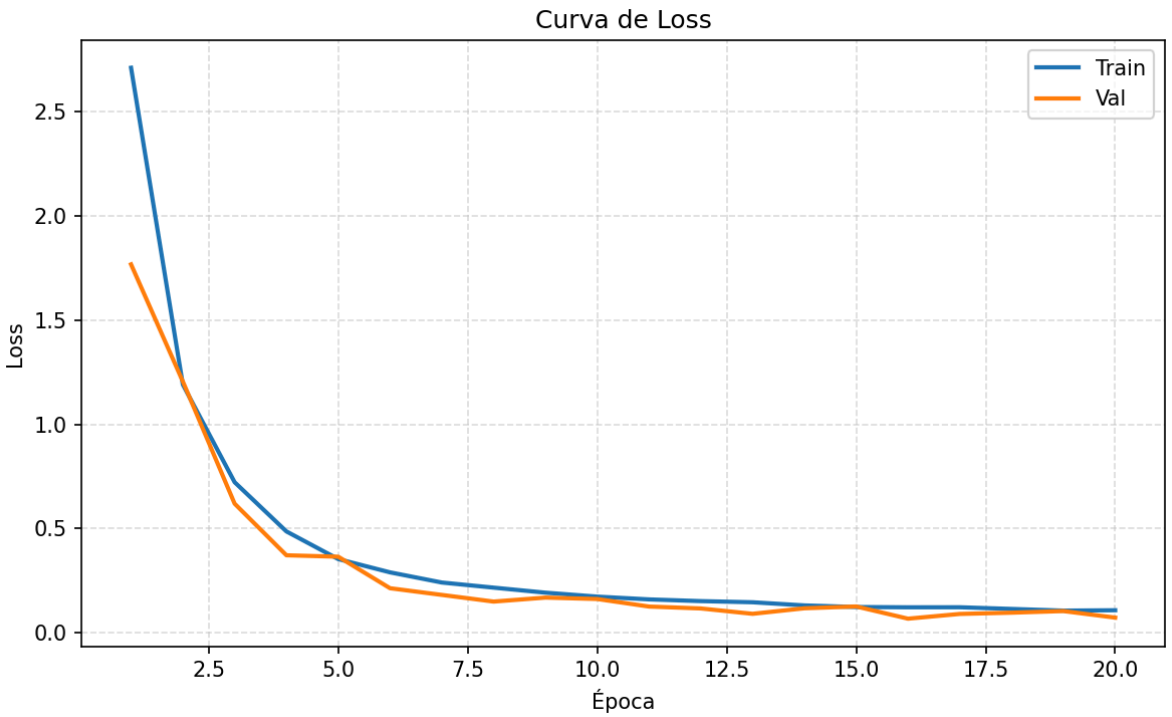
Mewtwo	0.9655	0.9882	0.9767	85
Moltres	0.95	0.987	0.9682	77
Mr. Mime	0.9302	1	0.9639	40
MrMime	1	0.9444	0.9714	36
Nidoking	0.9775	0.9886	0.9831	88
Nidoqueen	1	0.974	0.9868	77
Nidorina	0.9571	0.971	0.964	69
Nidorino	0.9875	0.9753	0.9814	81
Ninetales	0.9307	0.9895	0.9592	95
Oddish	1	1	1	92
Omanyte	0.9595	1	0.9793	71
Omastar	1	0.9726	0.9861	73
Parasect	1	0.9877	0.9938	81
Pidgeot	0.9459	0.6542	0.7735	107
Pidgeotto	0.6809	0.9796	0.8033	98
Pidgey	0.9658	0.9276	0.9463	152
Pikachu	0.9941	0.9883	0.9912	171
Pinsir	1	0.9619	0.9806	105
Poliwag	0.9923	0.9627	0.9773	134
Poliwhirl	0.8435	0.9764	0.9051	127
Poliwrath	0.9437	0.8072	0.8701	83
Ponyta	0.9655	0.9882	0.9767	85
Porygon	1	1	1	69
Primeape	0.9722	0.9545	0.9633	110
Psyduck	0.9739	1	0.9868	149
Raichu	1	0.9695	0.9845	131
Rapidash	0.9895	0.94	0.9641	100
Raticate	1	0.9891	0.9945	92
Rattata	1	0.9878	0.9939	82
Rhydon	0.9462	1	0.9724	88
Rhyhorn	1	0.9022	0.9486	92
Sandshrew	1	1	1	73
Sandslash	1	0.9789	0.9894	95
Scyther	0.9924	0.9924	0.9924	131
Seadra	0.975	0.9873	0.9811	79
Seaking	1	1	1	48
Seel	0.9362	1	0.967	44

Shellder	0.9783	1	0.989	90
Slowbro	0.9839	1	0.9919	61
Slowpoke	1	0.9651	0.9822	86
Snorlax	0.9923	0.9923	0.9923	130
Spearow	0.9608	0.9899	0.9751	99
Squirtle	1	1	1	146
Starmie	0.9333	0.9825	0.9573	114
Staryu	0.9897	0.9412	0.9648	102
Tangela	0.955	1	0.977	106
Tauros	0.982	0.9909	0.9864	110
Tentacool	0.9867	1	0.9933	74
Tentacruel	1	1	1	91
Vaporeon	1	0.9926	0.9963	135
Venomoth	1	0.9615	0.9804	104
Venonat	1	0.9911	0.9955	112
Venusaur	0.9735	1	0.9865	110
Victreebel	0.9907	0.9727	0.9817	110
Vileplume	0.982	0.9909	0.9864	110
Voltorb	0.9766	0.9921	0.9843	126
Vulpix	0.9821	0.991	0.9865	111
Wartortle	1	0.9727	0.9862	110
Weedle	0.9914	0.9914	0.9914	116
Weepinbell	0.9891	0.9891	0.9891	92
Weezing	0.9773	1	0.9885	86
Wigglytuff	0.9919	0.9919	0.9919	123
Zapdos	0.9915	0.9831	0.9872	118
Zubat	0.9333	1	0.9655	70

Desempeño general de las clases:

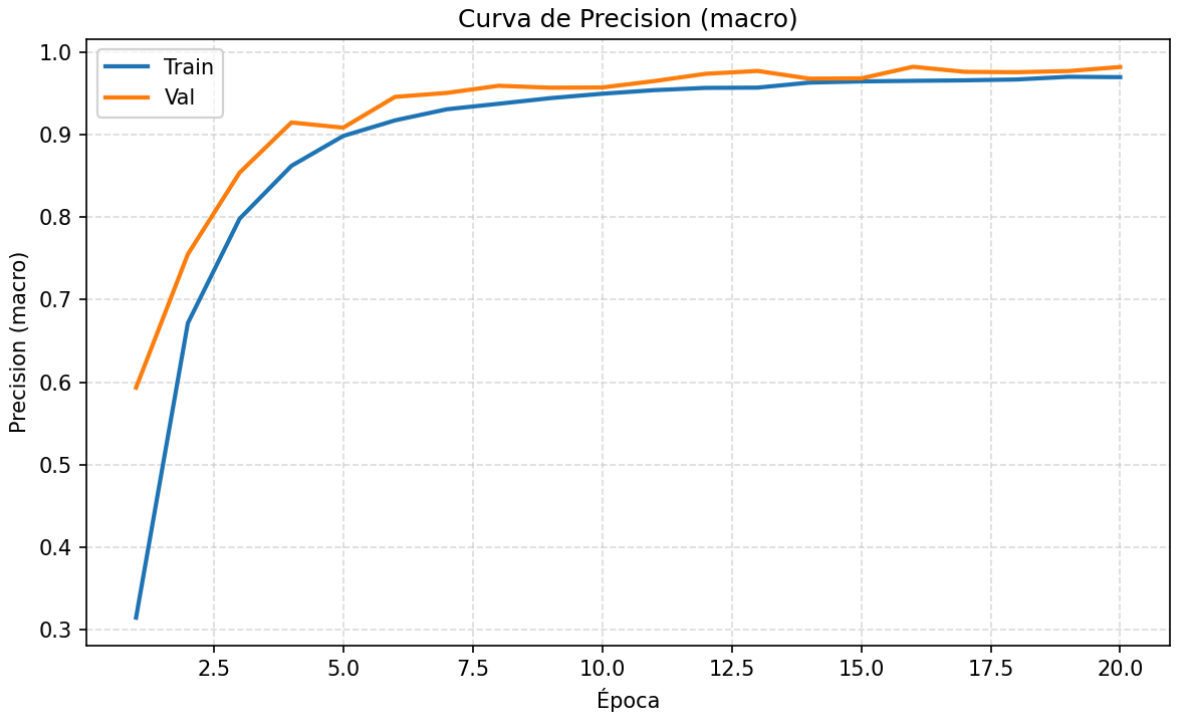
accuracy	0.979		13140	
macro avg	0.98	0.9799	0.9793	13140
weighted avg	0.9805	0.979	0.979	13140

Gráfica de Pérdida



- El modelo aprende progresivamente y de forma estable.
- No hay señales de sobreajuste.
- El rendimiento en validación es tan bueno como en entrenamiento.
- La curva es la forma ideal que quieres ver en un modelo bien entrenado.

Gráfica de Precisión



- La precisión aumenta de forma progresiva y estable durante las épocas.
- La curva de validación se mantiene muy cercana (e incluso ligeramente por encima) a la de entrenamiento.
- No hay señales de sobreajuste ni divergencia entre ambas curvas.
- El modelo mantiene una precisión alta y consistente a lo largo de todo el entrenamiento.
- Alcanzar valores cercanos a 0.98 confirma que el modelo generaliza excepcionalmente bien y está en un estado técnico excelente.

Conclusiones de la Iteración

Los resultados anteriores muestran que el modelo está funcionando de manera muy sólida: obtuvo una accuracy global de 0.979 y valores macro y weighted promedio cercanos a 0.98 en precisión, recall y F1, lo que indica un desempeño alto y bastante equilibrado entre todas las clases. La mayoría de los Pokémon alcanzan F1 por arriba

de 0.97, e incluso varias clases presentan métricas perfectas. Sin embargo, noto que algunas clases específicas —como Pidgeot, Pidgeotto, Poliwhirl y Poliwrath— tienen un rendimiento más bajo en comparación con el resto, probablemente por su similitud visual con sus líneas evolutivas o por variaciones en las imágenes disponibles.

Mejoras a implementar

Aunque el modelo ya muestra un desempeño excelente y métricas muy altas, todavía puedo mejorarlo ajustando algunos de sus hiper parámetros clave. Esta sería la parte de fine tuning como diría Benji.

Aumentar el tamaño de imagen

- Valor actual: img_size=160
- Mejora posible: probar 192 o 224 para capturar más detalle.

Ajustar el batch size

- Valor actual: batch_size=64
- Mejora posible: probar 32 (mejor generalización) o 128 (gradiente más estable).

Extender el número de épocas

- Valor actual: epochs=20
- Mejora posible: probar 25–30 para ver si mejora la precisión sin sobreajuste.

Afinar el learning rate

- Valor actual: lr=0.001
- Mejora posible: probar 0.0005 o 0.0002 para un ajuste más fino al final del entrenamiento.

Modificar el weight decay

- Valor actual: wd=0.0001
- Mejora posible: probar 0.0005 o 0.001 para mayor regularización.

Cambiar el dropout en la capa final

- Valor actual: p=0.2
- Mejora posible: probar 0.3 o 0.4 si aparece ligero sobreajuste en clases difíciles.

Ajustar el ReduceLROnPlateau

- Valores actuales: factor=0.5, patience=3
- Mejora posible: probar factor=0.1 o patience=2 para un ajuste de LR más agresivo.

Modificar el umbral del sampler por desbalance

- Valor actual: ratio >= 2
- Mejora posible: bajar a 1.5 para activar el sampler en más clases ligeramente desbalanceadas.

4.2 Segunda Iteración

En esta segunda iteración busco refinar el modelo sin modificar su arquitectura base, centrándome únicamente en un ajuste fino de los hiperparámetros de entrenamiento. El objetivo es aprovechar mejor la capacidad de ResNet18 para capturar detalles visuales sutiles entre Pokémon muy similares, permitiendo que el modelo entrene durante más tiempo pero con pasos de aprendizaje más pequeños y un control más sensible sobre la tasa de aprendizaje cuando la validación deja de mejorar.

Estos cambios no reinventan el modelo, sino que pulen su comportamiento para intentar exprimir un poco más de rendimiento a partir de una configuración que ya era sólida.

Parámetros nuevos para la segunda iteración:

- img_size=192
- epochs=30
- lr=0.0005
- scheduler patience=2 (en ReduceLROnPlateau)

Elegí ajustar solo img_size, epochs, lr y la patience del scheduler porque afectan directamente cómo aprende el modelo (más detalle, más tiempo de entrenamiento y pasos más finos, con una bajada de LR más sensible) sin tocar todavía la regularización fuerte ni la distribución de batches.

No cambié batch_size, weight_decay, dropout ni el umbral del sampler porque el modelo no muestra sobreajuste y ya es muy estable: si también modificamos esos parámetros, sería mucho más difícil saber qué cambió realmente mejoró o empeoró el resultado.

Desempeño por Clase: Precision, Recall y F1-score:

Clase	Precision	Recall	F1-score	Support
Abra	1	0.9787	0.9892	47
Aerodactyl	1	1	1	67
Alakazam	1	0.9855	0.9927	69
Arbok	1	1	1	104
Arcanine	1	1	1	66
Articuno	1	1	1	74
Beedrill	1	1	1	70
Bellsprout	0.9907	1	0.9953	106
Blastoise	1	1	1	114
Bulbasaur	1	1	1	118
Butterfree	1	1	1	93
Caterpie	1	1	1	105

Chansey	1	1	1	85
Charizard	1	1	1	82
Charmander	1	1	1	103
Charmeleon	1	1	1	115
Clefable	0.988	1	0.9939	82
Clefairy	1	0.9873	0.9936	79
Cloyster	1	1	1	73
Cubone	0.9892	0.9892	0.9892	93
Dewgong	1	1	1	88
Diglett	1	1	1	67
Ditto	1	1	1	62
Dodrio	1	1	1	71
Doduo	1	1	1	70
Dragonair	0.9899	1	0.9949	98
Dragonite	1	1	1	118
Dratini	1	0.9889	0.9944	90
Drowzee	1	1	1	77
Dugtrio	1	1	1	91
Eevee	1	1	1	115
Ekans	1	1	1	88
Electabuzz	1	1	1	96
Electrode	1	1	1	47
Exeggcute	1	1	1	95
Exeggutor	1	1	1	132
Farfetchd	1	1	1	101
Fearow	1	1	1	111
Flareon	1	1	1	133
Gastly	1	1	1	119
Gengar	1	1	1	116
Geodude	1	1	1	69
Gloom	0.9667	1	0.9831	58
Golbat	1	1	1	102
Goldeen	1	1	1	91
Golduck	1	1	1	115
Graveler	1	1	1	63
Grimer	1	1	1	61
Growlithe	1	1	1	91

Gyarados	1	1	1	120
Haunter	1	1	1	89
Hitmonchan	1	1	1	86
Hitmonlee	1	1	1	81
Horsea	1	1	1	102
Hypno	1	1	1	58
Ivysaur	1	0.9833	0.9916	120
Jigglypuff	1	0.9932	0.9966	148
Jolteon	1	1	1	100
Jynx	1	1	1	67
Kabutops	1	1	1	80
Kadabra	0.9722	1	0.9859	70
Kakuna	1	1	1	89
Kangaskhan	1	1	1	73
Kingler	1	1	1	80
Koffing	0.9877	1	0.9938	80
Lapras	1	1	1	96
Lickitung	1	1	1	74
Machamp	1	1	1	72
Machoke	1	1	1	67
Machop	1	1	1	70
Magikarp	1	1	1	89
Magmar	1	1	1	69
Magnemite	1	1	1	76
Magneton	1	1	1	76
Mankey	0.9867	0.9867	0.9867	75
Marowak	0.9863	0.9863	0.9863	73
Meowth	1	1	1	66
Metapod	1	1	1	85
Mew	1	1	1	77
Mewtwo	1	1	1	85
Moltres	1	1	1	77
Mr. Mime	0.9524	1	0.9756	40
MrMime	1	0.9444	0.9714	36
Nidoking	1	0.9886	0.9943	88
Nidoqueen	1	1	1	77
Nidorina	1	1	1	69

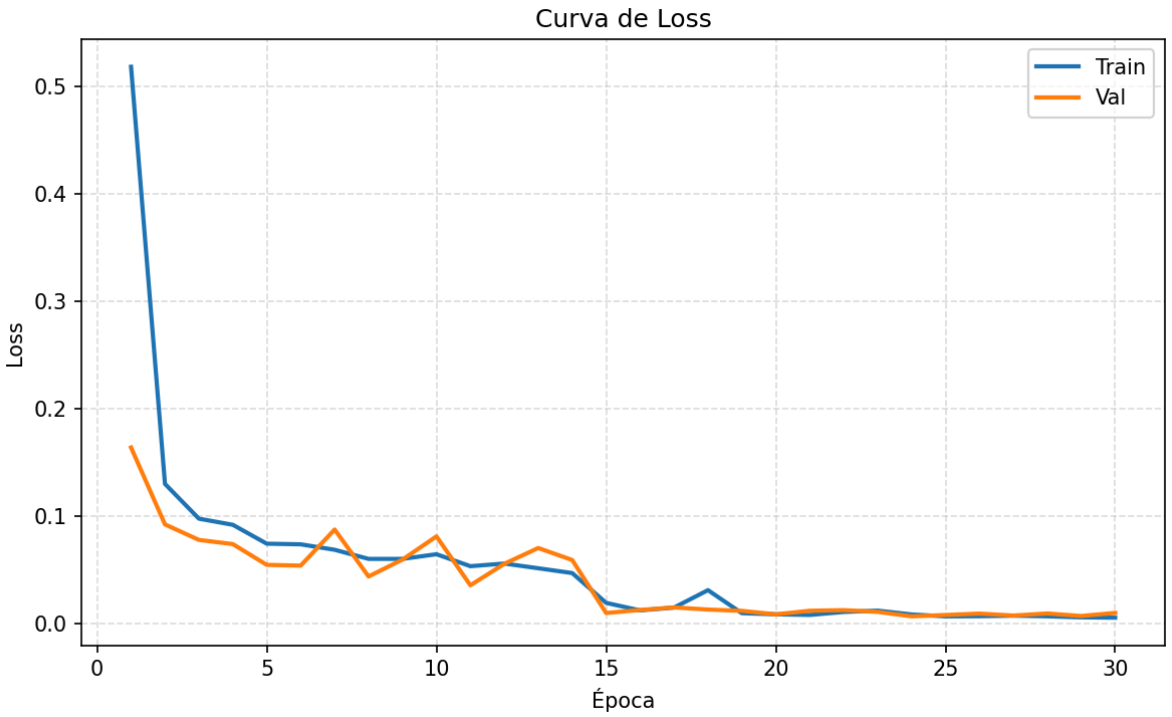
Nidorino	0.9878	1	0.9939	81
Ninetales	1	1	1	95
Oddish	1	1	1	92
Omanyte	1	0.9859	0.9929	71
Omastar	0.9865	1	0.9932	73
Parasect	1	1	1	81
Pidgeot	0.9298	0.9907	0.9593	107
Pidgeotto	0.9388	0.9388	0.9388	98
Pidgey	1	0.9539	0.9764	152
Pikachu	1	1	1	171
Pinsir	1	1	1	105
Poliwag	1	1	1	134
Poliwhirl	0.976	0.9606	0.9683	127
Poliwrath	0.9412	0.9639	0.9524	83
Ponyta	0.9765	0.9765	0.9765	85
Porygon	1	1	1	69
Primeape	0.9909	0.9909	0.9909	110
Psyduck	1	1	1	149
Raichu	1	1	1	131
Rapidash	0.98	0.98	0.98	100
Raticate	1	1	1	92
Rattata	1	1	1	82
Rhydon	0.967	1	0.9832	88
Rhyhorn	1	0.9674	0.9834	92
Sandshrew	1	1	1	73
Sandslash	1	1	1	95
Scyther	1	1	1	131
Seadra	1	1	1	79
Seaking	1	1	1	48
Seel	1	1	1	44
Shellder	1	1	1	90
Slowbro	1	1	1	61
Slowpoke	1	1	1	86
Snorlax	1	1	1	130
Spearow	1	1	1	99
Squirtle	1	1	1	146
Starmie	1	0.9912	0.9956	114

Staryu	0.9903	1	0.9951	102
Tangela	1	1	1	106
Tauros	1	1	1	110
Tentacool	1	1	1	74
Tentacruel	1	1	1	91
Vaporeon	1	1	1	135
Venomoth	1	1	1	104
Venonat	1	1	1	112
Venusaur	0.9821	1	0.991	110
Victreebel	1	0.9818	0.9908	110
Vileplume	1	0.9818	0.9908	110
Voltorb	1	1	1	126
Vulpix	1	1	1	111
Wartortle	1	1	1	110
Weedle	1	1	1	116
Weepinbell	0.9892	1	0.9946	92
Weezing	1	0.9884	0.9942	86
Wigglytuff	0.9919	1	0.996	123
Zapdos	1	1	1	118
Zubat	1	1	1	70

Desempeño general de las clases:

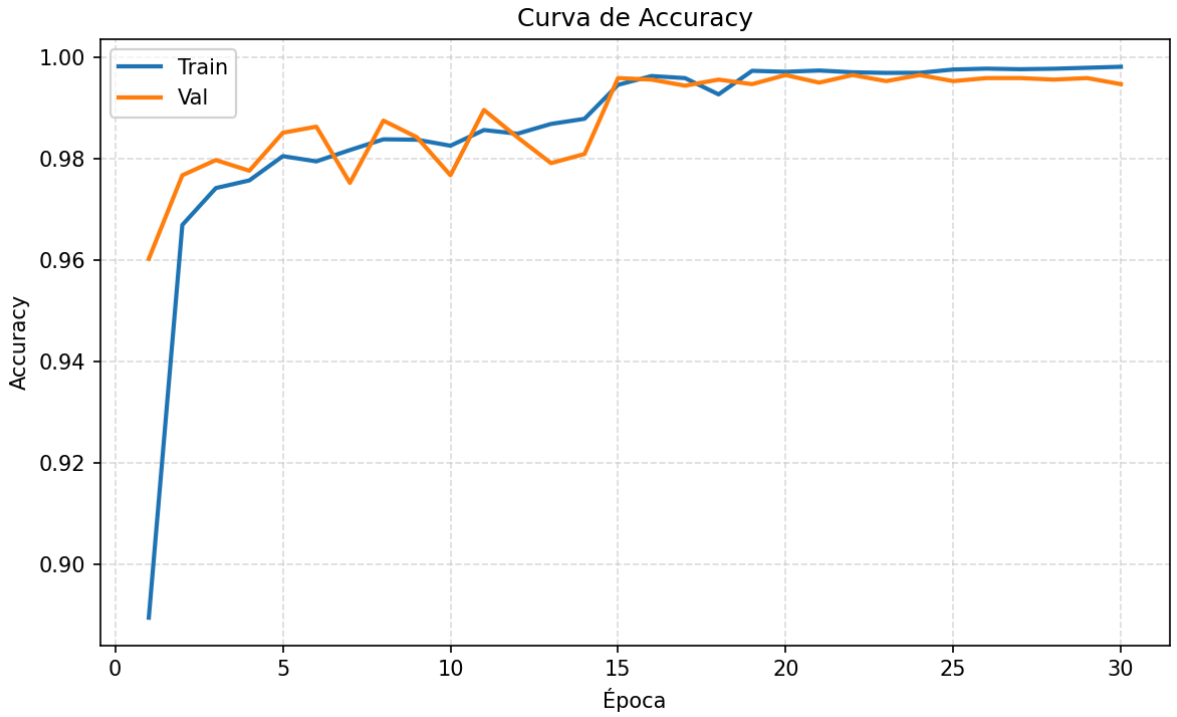
accuracy	0.9962			13140
macro avg	0.9961	0.9963	0.9961	13140
weighted avg	0.9963	0.9962	0.9962	13140

Gráfica de Pérdida



- La loss de entrenamiento cae muy fuerte al inicio y luego desciende suave hasta casi cero.
- La loss de validación baja en paralelo y casi siempre se mantiene ligeramente por debajo de la de train.
- Las oscilaciones en validación son pequeñas y se mantienen en un rango muy bajo.
- A partir de ~la época 15 ambas curvas se aplanan muy cerca de cero: el modelo casi no comete errores.
- No hay ningún pico ni separación grande entre train y val, la tendencia global es de aprendizaje estable y bien generalizada

Gráfica de Precisión



- El modelo aumenta su accuracy de forma rápida y progresiva hasta estabilizarse cerca de 0.996.

- Las curvas de entrenamiento y validación se mantienen siempre muy cercanas, sin separarse ni cruzarse de forma extraña.

- No hay señales de sobreajuste: la accuracy de validación es tan buena (o incluso ligeramente mejor al inicio) que la de entrenamiento.

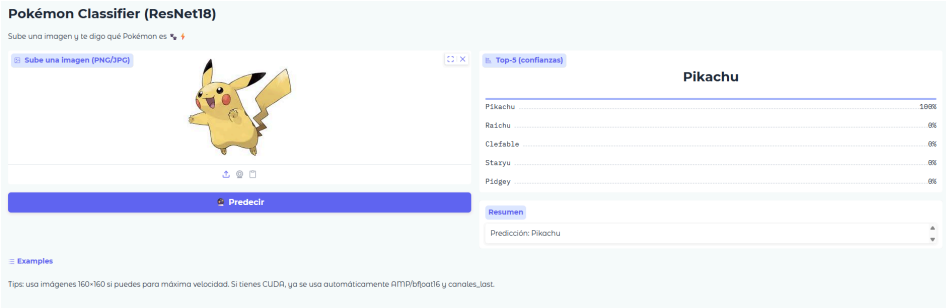
- La forma de ambas curvas es exactamente la que quieres ver en un modelo bien entrenado y bien generalizado.

Conclusiones de la Iteración

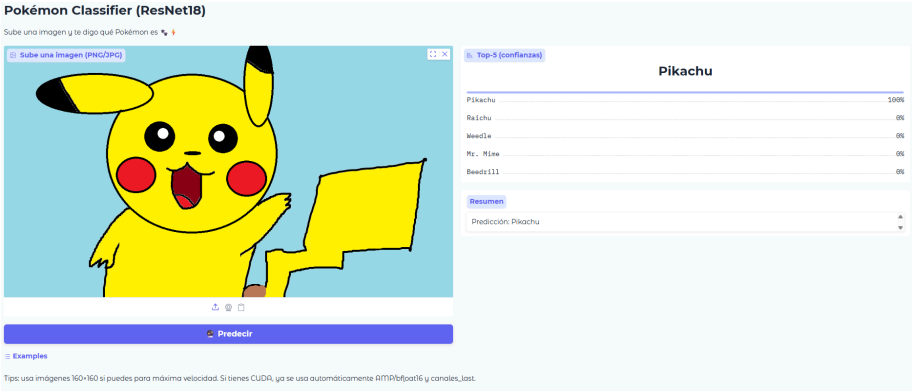
Mis resultados muestran que mi modelo está prácticamente resolviendo la tarea de clasificación: obtuve una accuracy global de 0.9962 y promedios macro y weighted de precisión, recall y F1 alrededor de 0.996, lo que significa que, en promedio, casi nunca me equivoco y que el rendimiento está muy equilibrado entre todas las clases. La gran mayoría de los Pokémon tienen un F1 de 1.0 o muy cercano, por lo que los estoy identificando de forma casi perfecta a pesar de la cantidad de clases. Las pocas clases donde las métricas bajan ligeramente (como Pidgeot, Pidgeotto, Poliwhirl, Poliwrath,

Gloom, Mr. Mime / MrMime, Venusaur, etc.) siguen con F1 por arriba de ~0.95, lo que indica que los errores residuales se concentran en líneas evolutivas muy parecidas visualmente. En conjunto, puedo decir que esta segunda iteración llevó mi modelo a un nivel de desempeño casi perfecto y totalmente defendible como solución sólida al problema planteado.

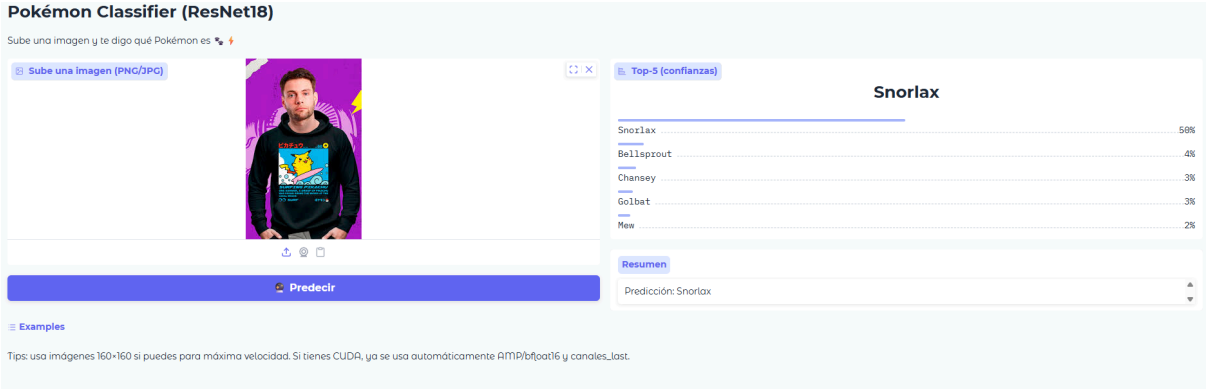
4.3 Modelo prediciendo



En este ejemplo el modelo ubica bien al pikachu, no hay ningún elemento que pueda confundir al modelo.



En este segundo caso es una imagen que yo mismo diseñe que igual ubica al Pikachu de manera perfecta. ya que las características más de este pokémon se encuentran en mi ilustración.



En este caso, el modelo no logró reconocer correctamente al Pikachu de la imagen debido a la gran cantidad de ruido presente. En una iteración futura, sería conveniente incorporar un modelo de segmentación que ayude a identificar mejor los elementos relevantes de la imagen y, con ello, mejorar el desempeño del clasificador.

5. Conclusión

Tras los ajustes de hiperparámetros en la segunda iteración —principalmente el aumento de la resolución de entrada, más épocas de entrenamiento y un learning rate más pequeño— el mismo modelo ResNet18 alcanzó un rendimiento casi perfecto: una accuracy global de 0.9962 y métricas macro y weighed de precisión, recall y F1 alrededor de 0.996. La mayoría de las clases obtuvieron F1 = 1.0 o muy cercano, y los pocos casos con valores ligeramente menores (como Pidgeot, Pidgeotto, Poliwhirl, Poliwrath, Gloom o Mr. Mime) siguen estando por encima de ~0.95, lo que indica que los errores se concentran en líneas evolutivas muy parecidas entre sí. En conjunto, los resultados muestran que este modelo no solo resuelve de manera muy efectiva la tarea de clasificación de Pokémon Gen1, sino que lo hace con un nivel de precisión y equilibrio entre clases que lo hacen totalmente defendible como una solución robusta y madura.

Sin embargo, también se observa que el modelo puede fallar cuando la imagen contiene demasiado ruido o elementos irrelevantes que distraen la atención del clasificador, lo que sugiere que en escenarios más complejos podría ser necesario incorporar una etapa previa de segmentación o limpieza de la imagen.