Projeto: Detector de Gatos vs Cachorros — Notebook Colab (célula por célula)

> **Resumo:** Notebook pronto para rodar no Google Colab que constrói, treina e avalia uma CNN para classificar imagens de gatos vs cachorros usando um dataset do Hugging Face Hub. O notebook está organizado célula-a-célula com explicações em Markdown antes de cada célula e comentários linha-a-linha no código.

> **Como usar:** copie cada bloco de código em uma célula do Colab (ou baixe o `.ipynb` se preferir). As células estão numeradas. As instruções de autenticação do Hugging Face e como ativar GPU estão incluídas.

<!-- ========== CÉLULA 1: TÍTULO e RESUMO (MARKDOWN) ========== -->

Célula 1 — Título e resumo

Detector de Gatos vs Cachorros (Colab)

Objetivo: construir e treinar uma CNN (do zero + opção transfer learning) para classificar imagens de gatos e cachorros. Entrega: notebook célula por célula, modelo salvo, métricas (CSV/JSON), plots e instruções para push to Hugging Face Hub.

<!-- ========== CÉLULA 2: INSTALAÇÕES E IMPORTS ========== -->

Célula 2 — Instalação de dependências e imports

```
# EXECUTE esta célula no Colab. Ela instala bibliotecas necessárias.
!pip install -q datasets huggingface_hub transformers torch torchvision
torchaudio scikit-learn albumentations matplotlib seaborn tensorboard wandb
tqdm pillow
```

```
# Imports principais (execute em célula Python)
import os
import random
import json
from pathlib import Path
from tqdm import tqdm

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
import seaborn as sns

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, random_split
import torchvision.transforms as T
import torchvision.models as models
from PIL import Image

from datasets import load_dataset
from huggingface_hub import login, HfApi, Repository

from sklearn.metrics import accuracy_score, f1_score, roc_auc_score,
classification_report, confusion_matrix, precision_recall_fscore_support

# Verificar GPU
print('Torch version:', torch.__version__)
print('CUDA available:', torch.cuda.is_available())
```

Explicação: instala tudo que o notebook usará e importa módulos. Se não quiser W&B, ignore a instalação relacionada.

<!-- ====================== CÉLULA 3: AUTENTICAÇÃO HUGGING FACE (MARKDOWN) ====================== -->

Célula 3 — Autenticação Hugging Face (Markdown)

Siga as instruções abaixo para autenticar sua conta Hugging Face no Colab. Você precisará do seu token (Settings -> Access Tokens).

<!-- ====================== CÉLULA 4: LOGIN HUGGING FACE (CÓDIGO) ====================== -->

Célula 4 — Login Hugging Face (código)

```
# Rode esta célula e cole seu token quando solicitado
from huggingface_hub import login

# Chame login() e cole o token no prompt interativo do Colab
login()

# Alternativa via CLI (executar no terminal do Colab):
# !huggingface-cli login
```

Explicação: `login()` abre prompt para colar seu token. Isso permite push_to_hub e download de datasets privados.

<!-- ========== CÉLULA 5: CARREGAR DATASET HUGGING FACE (MARKDOWN) ========== -->

Célula 5 — Carregar dataset do Hugging Face (Markdown)

Escolha o `dataset_name` do Hub. Exemplo sugerido: `nateraw/oxford-iiit-pet` (ou `beans`, `stanford_dogs`, etc.). Você pode substituir pelo dataset que preferir.

<!-- ========== CÉLULA 6: CARREGAR DATASET (CÓDIGO) ========== -->

Célula 6 — Código para baixar/carregar o dataset HF

```
# Parâmetros do usuário: altere aqui se quiser outro dataset
DATASET_NAME = 'nateraw/oxford-iiit-pet'
# exemplo; troque pelo dataset cats vs dogs que preferir
IMAGE_COLUMN = 'image'
LABEL_COLUMN = 'labels'

# Carregar o dataset via Hugging Face `datasets`
dataset = load_dataset(DATASET_NAME)

# Mostrar estrutura
print(dataset)

# Exibir algumas entradas
for split in dataset.keys():
    print(f"\n--- Split: {split} ---")
    print(dataset[split][0])
```

Explicação: usa `datasets.load_dataset` para baixar. A estrutura varia por dataset — ajuste `IMAGE_COLUMN` / `LABEL_COLUMN` conforme necessário.

<!-- ========== CÉLULA 7: INSPEÇÃO E LIMPEZA (MARKDOWN) ========== -->

Célula 7 — Inspeção e limpeza dos dados (Markdown)

Objetivos: verificar colunas, contar amostras por classe, identificar imagens corrompidas.

<!-- ========== CÉLULA 8: VERIFICAÇÃO E CORREÇÃO DE IMAGENS (CÓDIGO) ========== -->

Célula 8 — Verificar imagens corrompidas e contagem por classe

```
# Esta célula converte as imagens para um diretório local (opcional) e
verifica arquivos corrompidos.
OUT_DIR = Path('/content/dataset_images')
OUT_DIR.mkdir(parents=True, exist_ok=True)

# Função utilitária para salvar imagens localmente e coletar labels
examples = []
for split in dataset.keys():
    split_dir = OUT_DIR / split
    split_dir.mkdir(parents=True, exist_ok=True)
    for i, item in enumerate(tqdm(dataset[split], desc=f"Saving {split}")):
        try:
            img = item[IMAGE_COLUMN]
            # `datasets` pode já devolver PIL.Image or np.array depending on
dataset
            if isinstance(img, Image.Image):
                pil = img
            else:
                pil = Image.fromarray(img)
            label = None
            # Alguns datasets usam label int ou dict; aqui tratamos genérico
            if LABEL_COLUMN in item:
                label = item[LABEL_COLUMN]
            else:
                # tentar inferir por presença de 'label' ou 'class'
                label = item.get('label') or item.get('class')
            # Normalizar label para string
            if isinstance(label, (list, tuple)):
                label = label[0]
            img_path = split_dir / f"{split}_{i}.jpg"
            pil.convert('RGB').save(img_path)
            examples.append({'path': str(img_path), 'label': label, 'split':
split})
        except Exception as e:
            # Pula imagens corrompidas e registra
            print(f"Erro salvando item {i} do split {split}: {e}")

# Criar dataframe com os exemplos
df = pd.DataFrame(examples)

# Mostrar contagens por label
print('\nContagem por label:')
print(df['label'].value_counts())

# Converter labels para 0/1 (cats=0, dogs=1) — ajuste conforme necessário
```

```
unique_labels = sorted(df['label'].unique())
label2idx = {lbl: idx for idx, lbl in enumerate(unique_labels)}
print('\nMapeamento label->idx:', label2idx)

# Aplicar mapeamento
df['label_idx'] = df['label'].map(label2idx)

# Salvar CSV de referência
df.to_csv('/content/dataset_manifest.csv', index=False)
print('\nManifest salvo em /content/dataset_manifest.csv')
```

Explicação: exporta imagens para `/content/dataset_images`, cria um `manifest.csv` com paths e labels. Trate exceções de leitura.

<!-- ========== CÉLULA 9: PREPROCESSING E AUGMENTATION (MARKDOWN) ========== -->

Célula 9 — Preprocessing e Augmentation (Markdown)

Usaremos `torchvision.transforms` para pipelines simples; `albumentations` pode ser usado se preferir transformações mais avançadas.

<!-- ========== CÉLULA 10: DEFINIÇÃO DAS TRANSFORMS (CÓDIGO) ========== -->

Célula 10 — Definição de transforms para treino/val/test

```
IMAGE_SIZE = 224
BATCH_SIZE = 32

# Transforms para treino (augmentations)
train_transforms = T.Compose([
    T.Resize((IMAGE_SIZE, IMAGE_SIZE)),
    T.RandomHorizontalFlip(p=0.5),
    T.RandomRotation(15),
    T.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.1),
    T.ToTensor(),
    T.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

# Transforms para validação/teste (apenas resize + normalize)
val_transforms = T.Compose([
    T.Resize((IMAGE_SIZE, IMAGE_SIZE)),
    T.ToTensor(),
    T.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
```

Explicação: normalização baseada em ImageNet para compatibilidade com backbones pré-treinados.

<!-- ========== CÉLULA 11: DATASET & DATALOADER (MARKDOWN) ========== -->

Célula 11 — Criar dataset PyTorch e DataLoaders (Markdown)

Vamos criar uma classe `Dataset` que lê o `manifest.csv` e aplica transforms.

<!-- ========== CÉLULA 12: IMPLEMENTAR DATASET E DATALOADERS (CÓDIGO) ========== -->

Célula 12 — Implementar PyTorch Dataset e DataLoaders

```
class CatsDogsDataset(Dataset):
    def __init__(self, df, transform=None):
        self.df = df.reset_index(drop=True)
        self.transform = transform

    def __len__(self):
        return len(self.df)

    def __getitem__(self, idx):
        row = self.df.iloc[idx]
        img_path = row['path']
        label = int(row['label_idx'])
        # Abrir imagem
        image = Image.open(img_path).convert('RGB')
        if self.transform:
            image = self.transform(image)
        return image, label

# Ler manifest e criar splits 80/10/10 com seed fixo
manifest = pd.read_csv('/content/dataset_manifest.csv')
seed = 42
np.random.seed(seed)

# Shuffle
manifest = manifest.sample(frac=1, random_state=seed).reset_index(drop=True)

n = len(manifest)
train_end = int(0.8 * n)
val_end = train_end + int(0.1 * n)

train_df = manifest.iloc[:train_end]
val_df = manifest.iloc[train_end:val_end]
test_df = manifest.iloc[val_end:]
```

```
print(f"Tamanhos -> train: {len(train_df)}, val: {len(val_df)}, test:
{len(test_df)}")

# Instanciar datasets
dataset_train = CatsDogsDataset(train_df, transform=train_transforms)
dataset_val = CatsDogsDataset(val_df, transform=val_transforms)
dataset_test = CatsDogsDataset(test_df, transform=val_transforms)

# DataLoaders
train_loader = DataLoader(dataset_train, batch_size=BATCH_SIZE, shuffle=True,
num_workers=2)
val_loader = DataLoader(dataset_val, batch_size=BATCH_SIZE, shuffle=False,
num_workers=2)
test_loader = DataLoader(dataset_test, batch_size=BATCH_SIZE, shuffle=False,
num_workers=2)
```

Explicação: splits reprodutíveis com seed fixa. Ajuste `num_workers` conforme Colab.

<!-- ========== CÉLULA 13: ARQUITETURA (MARKDOWN) ========== -->

Célula 13 — Arquitetura do modelo (Markdown)

Implementaremos duas opções:

- **Opção A (padrão):** CNN do zero (pequena) — boa para aprendizado pedagógico.
- **Opção B:** Transfer learning com ResNet50 (recomendado para melhor desempenho).

Escolha a variável `USE_TRANSFER = True/False`.

<!-- ========== CÉLULA 14: CÓDIGO DO MODELO (CÓDIGO) ========== -->

Célula 14 — Implementação dos modelos

```
USE_TRANSFER = True  # Mude para False para treinar CNN do zero
NUM_CLASSES = len(unique_labels)

class SimpleCNN(nn.Module):
    def __init__(self, num_classes=2):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2),
```

```
            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2),

            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2),

            nn.AdaptiveAvgPool2d((1,1))
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Dropout(0.5),
            nn.Linear(128, 64),
            nn.ReLU(inplace=True),
            nn.Linear(64, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x

# Instanciar modelo conforme escolha
if USE_TRANSFER:
    model = models.resnet50(pretrained=True)
    # Trocar head
    in_features = model.fc.in_features
    model.fc = nn.Linear(in_features, NUM_CLASSES)
else:
    model = SimpleCNN(num_classes=NUM_CLASSES)

# Enviar para device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)
print(model)
```

Explicação: ResNet50 pré-treinado é geralmente superior; SimpleCNN serve para demonstração. Ajuste `USE_TRANSFER`.

<!-- ========== CÉLULA 15: CRITÉRIO, OTIMIZADOR, SCHEDULER (MARKDOWN) ========== -->

Célula 15 — Critério de perda, otimizador e scheduler (Markdown)

Usaremos `CrossEntropyLoss`, `AdamW` e `ReduceLROnPlateau` ou `StepLR`.

<!-- ====================== CÉLULA 16: INSTANTIAR LOSS/OPTIMIZER (CÓDIGO) ====================== -->

Célula 16 — Instanciar loss, optimizer e scheduler

```
LR = 1e-3
EPOCHS = 20

criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(model.parameters(), lr=LR)
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='max',
factor=0.5, patience=2)

# Optionally enable mixed precision
use_amp = True
scaler = torch.cuda.amp.GradScaler(enabled=use_amp)
```

Explicação: `ReduceLROnPlateau` reduz LR quando a métrica escolhida (val_f1 / val_auc) estagnar.

<!-- ====================== CÉLULA 17: FUNÇÕES AUXILIARES (MARKDOWN) ====================== -->

Célula 17 — Funções auxiliares: métricas, top-k, salvar checkpoints (Markdown)

Implementaremos funções para calcular métricas por batch/época e salvar checkpoints.

<!-- ====================== CÉLULA 18: IMPLEMENTAR FUNÇÕES AUXILIARES (CÓDIGO) ====================== -->

Célula 18 — Código das funções auxiliares

```
from collections import defaultdict
import math

# Função calcula acurácia top-k
def topk_accuracy(output, target, ks=(1,)):
    # output: logits (N, C), target: (N,)
    maxk = max(ks)
    batch_size = target.size(0)
    _, pred = output.topk(maxk, 1, True, True)
    pred = pred.t()
    correct = pred.eq(target.view(1, -1).expand_as(pred))
    res = []
```

```
    for k in ks:
        correct_k = correct[:k].reshape(-1).float().sum(0, keepdim=True)
        res.append((correct_k.mul_(100.0 / batch_size)).item())
    return res

# Função para calcular métricas de previsão
def compute_metrics_all(y_true, y_probs, y_pred):
    # y_true: list/np array, y_probs: prob of positive class shape (N,
num_classes)
    metrics = {}
    metrics['accuracy'] = accuracy_score(y_true, y_pred)
    metrics['f1_macro'] = f1_score(y_true, y_pred, average='macro')
    metrics['f1_micro'] = f1_score(y_true, y_pred, average='micro')
    # AUC: multiclass handling
    try:
        if y_probs.shape[1] == 2:
            # tomar prob da classe 1
            metrics['auc'] = roc_auc_score(y_true, y_probs[:,1])
        else:
            metrics['auc'] = roc_auc_score(y_true, y_probs,
multi_class='ovr')
    except Exception as e:
        metrics['auc'] = None
    # Precision/Recall per class
    p, r, f, s = precision_recall_fscore_support(y_true, y_pred,
average=None, zero_division=0)
    metrics['precision_per_class'] = p.tolist()
    metrics['recall_per_class'] = r.tolist()
    metrics['f1_per_class'] = f.tolist()
    return metrics

# Checkpoint helper
def save_checkpoint(state, is_best, checkpoint_dir='/content/checkpoints',
filename='checkpoint.pth'):
    os.makedirs(checkpoint_dir, exist_ok=True)
    path = os.path.join(checkpoint_dir, filename)
    torch.save(state, path)
    if is_best:
        best_path = os.path.join(checkpoint_dir, 'best_model.pth')
        torch.save(state, best_path)
```

Explicação: funções para Top-k, métricas e salvar checkpoints.

<!-- ========== CÉLULA 19: TRAIN + VAL EPISODE (MARKDOWN) ========== -->

Célula 19 — Loop de treino e validação (Markdown)

Implementaremos `train_one_epoch` e `evaluate` que calculam métricas por época e retornam resultados para logging.

<!-- ========== CÉLULA 20: IMPLEMENTAR TREINO E VALIDAÇÃO (CÓDIGO) ========== -->

Célula 20 — Código do training loop

```
from tqdm import tqdm

def train_one_epoch(model, dataloader, criterion, optimizer, device,
scaler=None):
    model.train()
    losses = []
    all_targets = []
    all_probs = []
    all_preds = []

    for images, targets in tqdm(dataloader, desc='Train', leave=False):
        images = images.to(device)
        targets = targets.to(device)

        optimizer.zero_grad()
        with torch.cuda.amp.autocast(enabled=(scaler is not None)):
            outputs = model(images)
            loss = criterion(outputs, targets)
        if scaler is not None:
            scaler.scale(loss).backward()
            scaler.step(optimizer)
            scaler.update()
        else:
            loss.backward()
            optimizer.step()

        losses.append(loss.item())

        probs = torch.softmax(outputs.detach().cpu(), dim=1).numpy()
        preds = np.argmax(probs, axis=1)

        all_targets.extend(targets.detach().cpu().numpy().tolist())
        all_probs.extend(probs.tolist())
        all_preds.extend(preds.tolist())

    # Agregar métricas
    all_targets = np.array(all_targets)
    all_probs = np.array(all_probs)
```

```
    all_preds = np.array(all_preds)
    metrics = compute_metrics_all(all_targets, all_probs, all_preds)
    metrics['loss'] = np.mean(losses)
    return metrics


def evaluate(model, dataloader, criterion, device):
    model.eval()
    losses = []
    all_targets = []
    all_probs = []
    all_preds = []

    with torch.no_grad():
        for images, targets in tqdm(dataloader, desc='Eval', leave=False):
            images = images.to(device)
            targets = targets.to(device)
            outputs = model(images)
            loss = criterion(outputs, targets)
            losses.append(loss.item())

            probs = torch.softmax(outputs.detach().cpu(), dim=1).numpy()
            preds = np.argmax(probs, axis=1)

            all_targets.extend(targets.detach().cpu().numpy().tolist())
            all_probs.extend(probs.tolist())
            all_preds.extend(preds.tolist())

    all_targets = np.array(all_targets)
    all_probs = np.array(all_probs)
    all_preds = np.array(all_preds)
    metrics = compute_metrics_all(all_targets, all_probs, all_preds)
    metrics['loss'] = np.mean(losses)
    return metrics
```

Explicação: `train_one_epoch` usa AMP (se `scaler` habilitado) e registra métricas por época.

<!-- ========== CÉLULA 21: RODAR TREINO (MARKDOWN) ========== -->

Célula 21 — Rodar treinamento (Markdown)

A célula abaixo executa o loop principal de treinamento por `EPOCHS`. Ela salva métricas a cada época em `metrics_history` e persiste o melhor modelo.

<!-- ========== CÉLULA 22: TREINAMENTO (CÓDIGO) ========== -->

Célula 22 — Executar training loop e salvar métricas

```
metrics_history = {'epoch': [], 'train_loss': [], 'val_loss': [],
'train_acc': [], 'val_acc': [], 'train_f1': [], 'val_f1': [], 'train_auc':
[], 'val_auc': []}
best_val_f1 = -math.inf

for epoch in range(1, EPOCHS+1):
    print(f"Epoch {epoch}/{EPOCHS}")
    train_metrics = train_one_epoch(model, train_loader, criterion,
optimizer, device, scaler if use_amp else None)
    val_metrics = evaluate(model, val_loader, criterion, device)

    # Atualizar scheduler com val_f1
    if val_metrics.get('f1_macro') is not None:
        scheduler.step(val_metrics['f1_macro'])

    # Logging simples
    print(f"Train loss: {train_metrics['loss']:.4f} - Train f1_macro:
{train_metrics.get('f1_macro'):.4f} - Train acc:
{train_metrics.get('accuracy'):.4f}")
    print(f"Val   loss: {val_metrics['loss']:.4f} - Val   f1_macro:
{val_metrics.get('f1_macro'):.4f} - Val   acc: {val_metrics.get('accuracy'):.
4f}")

    # Salvar métricas
    metrics_history['epoch'].append(epoch)
    metrics_history['train_loss'].append(train_metrics['loss'])
    metrics_history['val_loss'].append(val_metrics['loss'])
    metrics_history['train_acc'].append(train_metrics['accuracy'])
    metrics_history['val_acc'].append(val_metrics['accuracy'])
    metrics_history['train_f1'].append(train_metrics.get('f1_macro'))
    metrics_history['val_f1'].append(val_metrics.get('f1_macro'))
    metrics_history['train_auc'].append(train_metrics.get('auc'))
    metrics_history['val_auc'].append(val_metrics.get('auc'))

    # Checkpoint
    is_best = val_metrics.get('f1_macro', 0) > best_val_f1
    if is_best:
        best_val_f1 = val_metrics['f1_macro']
        save_checkpoint({'epoch': epoch, 'model_state_dict':
model.state_dict(), 'optimizer_state_dict': optimizer.state_dict(), 'val_f1':
best_val_f1}, is_best=True)

# Salvar metrics_history em CSV
metrics_df = pd.DataFrame(metrics_history)
metrics_df.to_csv('/content/metrics_history.csv', index=False)
print('\nMetrics salvas em /content/metrics_history.csv')
```

Explicação: Treina por `EPOCHS`, salva melhores checkpoints com base na métrica `val_f1_macro`.

<!-- ========== CÉLULA 23: PLOTS (MARKDOWN) ========== -->

Célula 23 — Plots das métricas por época (Markdown)

Gera gráficos de loss, accuracy e F1/AUC por época e salva em PNG.

<!-- ========== CÉLULA 24: GERAR PLOTS (CÓDIGO) ========== -->

Célula 24 — Código para gerar plots e salvar CSV

```
# Carregar metrics_history se necessário
# metrics_df = pd.read_csv('/content/metrics_history.csv')

plt.figure(figsize=(12,4))
plt.subplot(1,3,1)
plt.plot(metrics_df['epoch'], metrics_df['train_loss'], label='train_loss')
plt.plot(metrics_df['epoch'], metrics_df['val_loss'], label='val_loss')
plt.legend(); plt.title('Loss')

plt.subplot(1,3,2)
plt.plot(metrics_df['epoch'], metrics_df['train_acc'], label='train_acc')
plt.plot(metrics_df['epoch'], metrics_df['val_acc'], label='val_acc')
plt.legend(); plt.title('Accuracy')

plt.subplot(1,3,3)
plt.plot(metrics_df['epoch'], metrics_df['train_f1'], label='train_f1')
plt.plot(metrics_df['epoch'], metrics_df['val_f1'], label='val_f1')
plt.legend(); plt.title('F1 Macro')

plt.tight_layout()
plt.savefig('/content/training_plots.png')
print('Plots salvos em /content/training_plots.png')
```

Explicação: salva uma figura com três subplots (loss, acc, f1). Você pode gerar AUC separadamente se desejar.

<!-- ========== CÉLULA 25: AVALIAÇÃO FINAL (MARKDOWN) ========== -->

Célula 25 — Avaliação final no test set (Markdown)

Rodar avaliação completa no conjunto de teste: classification_report, ROC curve, confusion matrix e exemplos de predições erradas/certas.

<!-- ====================== CÉLULA 26: AVALIAR TEST SET (CÓDIGO) ====================== -->

Célula 26 — Código de avaliação no test set

```
# Carregar melhor checkpoint
best_path = '/content/checkpoints/best_model.pth'
if os.path.exists(best_path):
    ckpt = torch.load(best_path, map_location=device)
    model.load_state_dict(ckpt['model_state_dict'])
    print('Melhor modelo carregado do checkpoint (época', ckpt.get('epoch'),
')')
else:
    print('Nenhum checkpoint encontrado, usando modelo atual')

# Avaliar
test_metrics = evaluate(model, test_loader, criterion, device)
print('\nTeste metrics:\n', test_metrics)

# Classification report e confusion matrix
all_targets = []
all_probs = []
all_preds = []
model.eval()
with torch.no_grad():
    for images, targets in tqdm(test_loader, desc='Test'):
        images = images.to(device)
        outputs = model(images)
        probs = torch.softmax(outputs.detach().cpu(), dim=1).numpy()
        preds = np.argmax(probs, axis=1)
        all_targets.extend(targets.numpy().tolist())
        all_probs.extend(probs.tolist())
        all_preds.extend(preds.tolist())

print('\nClassification Report:\n')
print(classification_report(all_targets, all_preds, target_names=[str(x) for
x in unique_labels]))

cm = confusion_matrix(all_targets, all_preds)
plt.figure(figsize=(6,5))
sns.heatmap(cm, annot=True, fmt='d', xticklabels=unique_labels,
yticklabels=unique_labels)
plt.xlabel('Pred'); plt.ylabel('True'); plt.title('Confusion Matrix')
plt.savefig('/content/confusion_matrix.png')
print('Confusion matrix salva em /content/confusion_matrix.png')

# Salvar métricas finais
final_metrics = {'test': test_metrics}
with open('/content/final_metrics.json', 'w') as f:
```

```
    json.dump(final_metrics, f, indent=2)
print('Final metrics salvo em /content/final_metrics.json')
```

Explicação: carrega melhor checkpoint (se houver) e produz relatório detalhado.

<!-- ====================== CÉLULA 27: SALVAR MODELO E PUSH TO HUB (MARKDOWN) ====================== -->

Célula 27 — Salvar modelo local e instruções para push para Hugging Face Hub (Markdown)

Abaixo há um snippet para salvar o modelo `.pth` e (opcional) subir ao Hub. Para push, você precisa do `repo_id` e permissões.

<!-- ====================== CÉLULA 28: SALVAR E PUSH (CÓDIGO) ====================== -->

Célula 28 — Código para salvar o modelo e (opcional) push_to_hub

```
# Salvar modelo local
os.makedirs('/content/model', exist_ok=True)
MODEL_PATH = '/content/model/model.pth'
torch.save({'model_state_dict': model.state_dict(), 'label2idx': label2idx},
MODEL_PATH)
print('Modelo salvo em', MODEL_PATH)

# --- Push to Hugging Face Hub (opcional - duas opções) ---
# Opção A: Usando huggingface_hub.Repository (recomendado para múltiplos
arquivos)
# 1) Crie um repo no Hugging Face (https://huggingface.co/new) e use o
repo_id abaixo
# 2) Clone o repo localmente no Colab, copie os arquivos e faça push

repo_id = 'USERNAME/cats-vs-dogs-cnn'  # substitua pelo seu nome de usuário/
repo
local_repo_dir = '/content/hf_repo'

from huggingface_hub import Repository

# Clone o repo (se já existir vai levantar erro) - se já clonou, ignore
if not os.path.exists(local_repo_dir):
    repo = Repository(local_dir=local_repo_dir, clone_from=repo_id)
else:
    repo = Repository(local_dir=local_repo_dir)

# Copiar arquivos essenciais para o repo
```

```
import shutil
shutil.copy(MODEL_PATH, os.path.join(local_repo_dir, 'model.pth'))
shutil.copy('/content/metrics_history.csv', os.path.join(local_repo_dir,
'metrics_history.csv'))
shutil.copy('/content/training_plots.png', os.path.join(local_repo_dir,
'training_plots.png'))
shutil.copy('/content/confusion_matrix.png', os.path.join(local_repo_dir,
'confusion_matrix.png'))
shutil.copy('/content/final_metrics.json', os.path.join(local_repo_dir,
'final_metrics.json'))
shutil.copy('/content/dataset_manifest.csv', os.path.join(local_repo_dir,
'dataset_manifest.csv'))

# Criar um README mínimo para o modelo
readme_text = f"""# {repo_id}

Modelo PyTorch para classificação Cats vs Dogs.

Arquivos incluídos:
- model.pth
- metrics_history.csv
- training_plots.png
- confusion_matrix.png
- final_metrics.json
- dataset_manifest.csv

Como usar: carregar model.pth e rodar inferência conforme notebook.
"""
with open(os.path.join(local_repo_dir, 'README.md'), 'w') as f:
    f.write(readme_text)

# Commit & push
repo.push_to_hub(commit_message='Add trained model and artifacts')
print('Arquivos enviados para o repo Hugging Face:', repo_id)

# Opção B: Usando HfApi para upload de arquivos individuais (útil para
arquivos grandes ou uploads pontuais)
# from huggingface_hub import HfApi
# api = HfApi()
# api.upload_file(path_or_fileobj=MODEL_PATH, path_in_repo='model.pth',
repo_id=repo_id)
# api.upload_file(path_or_fileobj='/content/metrics_history.csv',
path_in_repo='metrics_history.csv', repo_id=repo_id)

print('
Observação: substitua `USERNAME/cats-vs-dogs-cnn` pelo seu repo criado no
Hugging Face.
Você deve ter feito login com `huggingface_hub.login()` anteriormente para
permitir o push.')
```

Explicação: salvar em `.pth`. Para push, clonar o repo HF e copiar arquivos.

<!-- ========== CÉLULA 29: INFERÊNCIA (MARKDOWN) ========== -->

Célula 29 — Como usar o modelo para inferência em imagens novas (Markdown)

Mostraremos código para carregar o modelo salvo e prever em 5 imagens de exemplo.

<!-- ========== CÉLULA 30: CÓDIGO DE INFERÊNCIA (CÓDIGO) ========== -->

Célula 30 — Exemplo de inferência em 5 imagens

```
from glob import glob

# Carregar modelo salvo
ckpt = torch.load(MODEL_PATH, map_location=device)
model.load_state_dict(ckpt['model_state_dict'])
model.to(device)
model.eval()

# Selecionar 5 imagens (pasta /content/dataset_images/test)
sample_paths = manifest.iloc[val_end:].sample(5, random_state=seed)
['path'].tolist()

for p in sample_paths:
    img = Image.open(p).convert('RGB')
    inp = val_transforms(img).unsqueeze(0).to(device)
    with torch.no_grad():
        out = model(inp)
        probs = torch.softmax(out.cpu(), dim=1).numpy()[0]
        pred = np.argmax(probs)
    print(f"Imagem: {p} -> Pred: {unique_labels[pred]} (prob={probs[pred]:.
3f})")
```

Explicação: exemplo simples para inferência. Você pode adaptar para API ou Gradio.

<!-- ========== CÉLULA 31: CONCLUSÕES, LIMITAÇÕES E PRÓXIMOS PASSOS (MARKDOWN) ========== -->

Célula 31 — Conclusões, limitações e próximos passos

- **Conclusão:** descreva performance alcançada e insights.
- **Limitações:** dataset pode ser tendencioso; imagens pequenas; overfitting; classe desbalanceada.

- **Próximos passos:** usar cross-validation, hyperparameter tuning (Optuna), augmentations avançadas, export ONNX, deploy com Gradio/Flask, testar outros backbones (EfficientNet), usar treinamento distribuído.

<!-- ========== CÉLULA 32: CHECKLIST DE ENTREGA (MARKDOWN) ========== -->

Célula 32 — Checklist de arquivos gerados e instruções para download

Arquivos esperados: - `/content/model/model.pth` — modelo salvo - `/content/metrics_history.csv` — métricas por época - `/content/training_plots.png` — plots das métricas - `/content/confusion_matrix.png` — matriz de confusão - `/content/final_metrics.json` — métricas finais - `/content/dataset_manifest.csv` — manifest com paths e labels

Para baixar arquivos do Colab: use o painel lateral esquerdo (Files) ou `from google.colab import files; files.download(path)`.

FIM

> **Observação técnica:** este notebook foi gerado como um template completo. Dependendo do dataset escolhido (nome/colunas/tipos), será preciso ajustar `IMAGE_COLUMN` e `LABEL_COLUMN` e às vezes o pipeline de salvamento (alguns datasets retornam imagens já salvas como paths). Se quiser, eu posso gerar o `.ipynb` pronto para download ou adaptar este notebook para TensorFlow/Keras. Solicite: `Gerar .ipynb` ou `Adaptar para TensorFlow`.