CAPÍTULO 6

BACKUP AUTOMATIZADO E OTIMIZADO COM RCLONE E PYTHON

1. O CENÁRIO DE RISCO E A NECESSIDADE DE BACKUP

O treinamento de redes neurais profundas, como os modelos da família YOLO, demanda tempo de processamento e gera múltiplos arquivos durante a execução: checkpoints de pesos (*best.pt* e *last.pt*), matrizes de confusão, curvas de precisão e revocação, imagens de validação e logs tabulares (*results.csv*). Interrupções de energia ou falhas na máquina host durante o processo podem corromper os arquivos locais e causar a perda de horas ou dias de processamento.

O envio manual ou a sincronização padrão para serviços de nuvem (como o Google Drive) enfrenta dois obstáculos práticos:

1. **Volume de arquivos pequenos:** as pastas de predições e logs contêm milhares de arquivos de poucos kilobytes. Serviços de nuvem impõem limites de transações por segundo (TPS) e rejeitam conexões que abrem muitas requisições simultâneas.
2. **Tamanho dos checkpoints:** os arquivos *best.pt* e *last.pt* ocupam dezenas ou centenas de megabytes. Conexões interrompidas exigem retomada automática sem corrupção do arquivo.

Para contornar esses limites, o Rclone é uma opção adequada para automatizar o envio.

2. O QUE É O RCLONE?

O Rclone é um utilitário de linha de comando para transferência e sincronização de arquivos entre sistemas locais e provedores de nuvem, com suporte a mais de 70 serviços (incluindo Google Drive, Amazon S3, Dropbox e OneDrive). Na sincronização de saídas do YOLO, seus principais recursos são:

- **Limite de requisições (*–tpslimit*):** restringe as transações por segundo para evitar o bloqueio de API pelo provedor.
- **Transferências simultâneas (*–transfers*):** gerencia o envio paralelo de arquivos, aproveitando a largura de banda de upload disponível.
- **Comparação por data de modificação e hash:** valida se o arquivo precisa ser atualizado pelo timestamp e pela assinatura. Como os arquivos *best.pt* e *last.pt* mantêm o mesmo tamanho em bytes entre épocas, a flag *–size-only* não deve ser usada, pois impediria a cópia dos novos pesos.
- **Tentativas automáticas (*–retries*):** repete a transferência após falhas momentâneas de conexão.

3. INTEGRAÇÃO ENTRE PYTHON E RCLONE

A execução do Rclone pode ser acoplada ao treinamento por meio de um script Python. O uso de *callbacks* da biblioteca Ultralytics permite disparar a sincronização ao final de cada época, preservando os pesos mais recentes na nuvem caso o computador seja desligado.

3.1 Script de Treinamento com Sincronização Periódica

O script a seguir configura as opções de GPU (cuDNN, AMP e TF32), instancia o modelo *yolov8s.pt*, registra a rotina de envio ao término de cada época e executa uma cópia final ao fechar o processo:

Script: *orquestrador_yolo_rclone.py*

```
import os
import site
import sys
import re
import time
import subprocess
from pathlib import Path

# Configuracao de bibliotecas nativas C++/CUDA para o ambiente
try:
    site_pkgs = site.getsitepackages()[0]
    cudnn_lib_path = os.path.join(site_pkgs, "nvidia", "cudnn", "lib")
    cublas_lib_path = os.path.join(site_pkgs, "nvidia", "cublas", "lib")
    ld_path = os.environ.get("LD_LIBRARY_PATH", "")
    os.environ["LD_LIBRARY_PATH"] = f"{cudnn_lib_path}:{cublas_lib_path}:{ld_path}"
    print("[OK] Variaveis do ambiente de suporte CUDA validadas")
```

```
except Exception as e:
    print(f"Aviso ao verificar caminhos CUDA: {e}")

try:
    import torch
    print("[OK] PyTorch importado com sucesso")
except Exception as e:
    print(f"Erro ao importar PyTorch: {e}")
    raise

try:
    from ultralytics import YOLO
    print("[OK] Ultralytics importado com sucesso")
except Exception as e:
    print(f"Erro ao importar Ultralytics: {e}")
    raise

# Habilitacao de cuDNN, AMP e precisao TF32 para alto desempenho
torch.backends.cudnn.enabled = True
torch.backends.cudnn.benchmark = True
if hasattr(torch.backends.cuda.matmul, "allow_tf32"):
    torch.backends.cuda.matmul.allow_tf32 = True
if hasattr(torch.backends.cudnn, "allow_tf32"):
    torch.backends.cudnn.allow_tf32 = True

base_dir = Path(".")
datasets_dir = base_dir / "datasets_extraidos"
modelos_dir = base_dir / "modelos"
output_dir = base_dir / "treinamentos"

# Nome correto do arquivo de pesos oficiais da Ultralytics
modelos_para_treinar = {
    "v8": "yolov8s.pt"
}

EPOCHS = 200
PATIENCE = 50
IMGSZ = 640
BATCH = 16
DEVICE = 0
WORKERS = 8
```

```
RCLONE_REMOTE = "gdrive"
RCLONE_PASTA = "BackupsYOLO/treinamentos"
RCLONE_TPS_LIMIT = "4"
RCLONE_TRANSFERS = "2"
RCLONE_CHECKERS = "2"
RCLONE_DELAY = 5

log_file = output_dir / "historico.txt"
log_file.parent.mkdir(parents=True, exist_ok=True)

def registrar_log(msg):
    timestamp = time.strftime("%Y-%m-%d %H:%M:%S")
    with open(log_file, "a", encoding="utf-8", buffering=1) as f:
        f.write(f"[{timestamp}] {msg}\n")
        f.flush()
        os.fsync(f.fileno())
    print(f"[{timestamp}] {msg}", flush=True)

def sanitizar_nome_pasta(nome: str):
    nome_limpo = re.sub(r"[\r\n]+", " ", nome)
    nome_limpo = re.sub(r"[^\w\s-]", "_", nome_limpo)
    return re.sub(r"\s+", "_", nome_limpo).strip("_")

def verificar_rclone():
    try:
        resultado = subprocess.run(
            ["rclone", "version"],
            capture_output=True,
            text=True
        )
        return resultado.returncode == 0
    except Exception as e:
        registrar_log(f"Rclone nao encontrado no sistema: {e}")
        return False

def sincronizar_treinamento_gdrive(pasta_treinamento: Path, dataset_nome: str, versao: str):
    if not rclone_disponivel or not pasta_treinamento.exists() or not pasta_treinamento.is_dir():
        return False
    destino = f"{RCLONE_REMOTE}:{RCLONE_PASTA}/{pasta_treinamento.name}"
    max_tentativas = 5
    tempos_espera = [30, 60, 120, 240, 300]
    for tentativa in range(1, max_tentativas + 1):
```

```
        try:
            # Nota: NAO utilizar --size-only em arquivos .pt!
            comando = [
            "rclone", "copy", str(pasta_treinamento), destino, "--transfers", RCLONE_TRANSFERS, "--checkers",
            RCLONE_CHECKERS, "--tpslimit", RCLONE_TPS_LIMIT, "--tpslimit-burst", "1",
            "-P", "--stats", "30s", "--log-level", "INFO"
            ]
            subprocess.run(comando,  check=True)
            registrar_log(f"Backup Rclone concluido para: {dataset_nome}/{versao}")
            return True




        except subprocess.CalledProcessError as e:
            registrar_log(f"Tentativa {tentativa}/{max_tentativas} de backup falhou: {e}")
            if tentativa < max_tentativas:
                time.sleep(tempos_espera[tentativa - 1])
            else:
                registrar_log(f"ERRO CRITICO: Falha definitiva de backup para {dataset_nome}")
                return False
        except Exception as e:
            registrar_log(f"Excecao inesperada durante backup: {e}")
            return False
    return False


def criar_data_yaml_padrao(pasta_dataset: Path) -> Path:
    yaml_path = pasta_dataset / "data.yaml"
    if yaml_path.exists():
        return yaml_path
    train_images = pasta_dataset / "images" / "train"
    val_images = pasta_dataset / "images" / "val"
    # Exige a presenca simultanea dos diretorios de treino e validacao
    if not (train_images.exists() and val_images.exists()):
        return None
    yaml_content = f"# Auto-generated data.yaml\npath: {pasta_dataset.resolve().as_posix()}\n"
    yaml_content += "train: images/train\nval: images/val\n"
    yaml_content += "\nnc: 2\nnames: ['class0', 'class1']\n"
    try:
        with open(yaml_path, "w", encoding="utf-8") as f:
            f.write(yaml_content)
        return yaml_path
```

```
        except Exception as e:
            registrar_log(f"Erro ao criar data.yaml em {pasta_dataset}: {e}")
            return None

configuracoes_data = []
if datasets_dir.exists():
    for pasta in sorted(datasets_dir.iterdir()):
        if not pasta.is_dir():
            continue
        yamls_encontrados = list(pasta.rglob("data.yaml"))
        if not yamls_encontrados:
            yaml_path = criar_data_yaml_padrao(pasta)
            if yaml_path is None:
                continue
            yamls_encontrados = [yaml_path]
        configuracoes_data.append({
            "nome": pasta.name,
            "nome_sanitizado": sanitizar_nome_pasta(pasta.name),
            "path": str(yamls_encontrados[0].resolve()),
            "task": "detect"
        })

rclone_disponivel = verificar_rclone()

for dataset in configuracoes_data:
    for versao, arquivo_modelo in modelos_para_treinar.items():
        caminho_modelo = modelos_dir / arquivo_modelo
        projeto_dir = output_dir / dataset["nome_sanitizado"]
        nome_run = versao
        pasta_treinamento = projeto_dir / nome_run

        # Callback para disparar backup periodico ao termino de cada epoca
        def on_fit_epoch_end(trainer):
            if rclone_disponivel and (trainer.epoch + 1) % 10 == 0:
                sincronizar_treinamento_gdrive(
                    pasta_treinamento, dataset["nome"], versao
                )

        try:
            # Se o arquivo local nao existir, o Ultralytics realiza o download
            fonte_modelo = str(caminho_modelo) if caminho_modelo.exists() else arquivo_modelo model =
            YOLO(fonte_modelo)
            model.add_callback("on_fit_epoch_end", on_fit_epoch_end)
```

```
                model.train(
                    data=dataset["path"],
                    task=dataset["task"],
                    epochs=EPOCHS, patience=PATIENCE,
                    imgsz=IMGSZ, batch=BATCH,
                    device=DEVICE, workers=WORKERS,
                    plots=True, amp=True,
                    project=str(projeto_dir), name=nome_run, exist_ok=True
                )
                registrar_log(f"Treinamento concluido com sucesso: {dataset['nome']}/{versao}")
            except Exception as e:
                registrar_log(f"Erro durante treinamento de {dataset['nome']}: {e}")
            finally:
                if rclone_disponivel:
                    time.sleep(RCLONE_DE
                    LAY)
                    sincronizar_treinamento_g
                    drive(
                        pasta_treinamento, dataset["nome"], versao
```

4. ENTENDENDO O CÓDIGO

A rotina combina o treinamento do detector com a transferência remota tolerante a falhas. Os principais blocos do script operam da seguinte forma:

4.1 Aceleração de Hardware (cuDNN, AMP e TF32)

O parâmetro *torch.backends.cudnn.enabled = True* ativa as rotinas de convolução otimizadas da cuDNN. A precisão mista (*amp=True*) reduz o consumo de memória de vídeo e acelera o processamento mantendo a estabilidade numérica. Em GPUs com arquitetura Ampere ou superior (RTX 30xx/40xx/50xx), a opção *allow_tf32 = True* acelera as multiplicações matriciais em ponto flutuante. A variável *LD_LIBRARY_PATH* deve ser configurada no ambiente do sistema operacional antes da chamada do script para permitir que o vinculador dinâmico (*ld.so*) localize as bibliotecas compartilhadas (*.so*) da NVIDIA.

4.2 Variáveis de Configuração e Parâmetros do Rclone

A configuração do Rclone define limites operacionais para a conexão:

- *RCLONE_REMOTE = "gdrive"*: nome do acesso remoto previamente configurado via terminal (*rclone config*).
- *RCLONE_TPS_LIMIT = "4"*: teto de requisições por segundo para prevenir bloqueios temporários de cota no Google Drive.
- *RCLONE_TRANSFERS = "2"*: envio paralelo de até 2 arquivos por ciclo.

4.3 Auto-Configuração do Arquivo data.yaml

A função *criar_data_yaml_padrao()* verifica a existência do arquivo de configuração do conjunto de dados. Se ele não for encontrado, a função valida a presença simultânea das pastas *images/train* e *images/val* (layout padronizado no Capítulo 3) e gera um arquivo *data.yaml* válido com o número de classes (*nc: 2*) coerente com a lista de nomes *['class0', 'class1']* delimitada por aspas simples retas.

4.4 Tolerância a Falhas e Descarte Obrigatório da Flag –size-only

O envio dos dados é orquestrado pela função *sincronizar_treinamento_gdrive()*, que combina o disparo do Rclone via *subprocess.run* com re-tentativas temporizadas em intervalos incrementais (30, 60, 120, 240 e 300 segundos) em caso de oscilação da conexão.

É necessário **não utilizar a flag *–size-only*** ao sincronizar repositórios do YOLO. Como os arquivos binários de checkpoints (*best.pt* e *last.pt*) contêm a mesma estrutura de camadas e contagem de parâmetros, seu tamanho em bytes permanece rigorosamente constante entre épocas sucessivas de treinamento. A flag *–size-only* levaria o Rclone a assumir incorretamente que o arquivo na nuvem já está atualizado, ignorando o envio dos novos pesos otimizados. Por essa razão, utiliza-se a comparação padrão baseada no horário de modificação (ModTime) e hash.

Por fim, o *callback on_fit_epoch_end* aciona a sincronização a cada época finalizada. Em caso de interrupção por falta de energia, os checkpoints na nuvem permitem retomar a execução a partir da última época sincronizada.

REFERÊNCIAS

JOCHER, Glenn; CHAURASIA, Ayush; QIU, Jing. Ultralytics YOLO. [S.l.: s.n.], 2023. Software de detecção, segmentação, classificação e estimativa de pose baseado na família YOLO. Disponível em: https://github.com/ultralytics/ultralytics.

RCLONE AUTHORS. Rclone: rsync for cloud storage. 2026. Disponível em: https://rclone.org.