main

June 12, 2025

1 Trabalho Computacional 3. Rede Convolucional e Transfer Learning

1.1 1. Introdução e Base de Dados

Neste trabalho usaremos uma rede convolucional pré-treinada e a aplicaremos em um problema novo. Também experimentaremos com a divisão da base em treinamento, validação e teste, e usaremos o conjunto de validação para a técnica "early stopping", na tentativa de controlar o sobre-ajuste.

A base de dados é a CIFAR10. Ela contém 60000 imagens 32x32 coloridas (3 canais) das seguintes categorias de objetos: 'airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'.

Ela pode ser baixada com o código abaixo.

[1]:
```
import torch
import torchvision
import torchvision.transforms as transforms
```

[2]:
```
class CIFAR10():  #@save
    def __init__(self, root, resize=(224, 224)):
        trans = transforms.Compose([transforms.Resize(resize),
                                    transforms.ToTensor(),
                                    transforms.Normalize((0.5, 0.5, 0.5), (0.5,␣
 ↪0.5, 0.5))])
        self.train = torchvision.datasets.CIFAR10(
            root=root, train=True, transform=trans, download=True)
        # use 20% of training data for validation
        train_set_size = int(len(self.train) * 0.8)
        valid_set_size = len(self.train) - train_set_size
         # split the train set into two
        seed = torch.Generator().manual_seed(42)
        self.train, self.val = torch.utils.data.random_split(self.train,␣
 ↪[train_set_size, valid_set_size], generator=seed)
        self.test = torchvision.datasets.CIFAR10(
            root=root, train=False, transform=trans, download=True)

dataset = CIFAR10(root="./data/")
```

```
train_dataloader = torch.utils.data.DataLoader(dataset.train, batch_size=64,␣
 ↪shuffle=True, num_workers=5)
val_dataloader = torch.utils.data.DataLoader(dataset.val, batch_size=64,␣
 ↪shuffle=False, num_workers=3)
test_dataloader = torch.utils.data.DataLoader(dataset.test, batch_size=64,␣
 ↪shuffle=False, num_workers=3)

print(f"Number of training examples: {len(dataset.train)}")
print(f"Number of validation examples: {len(dataset.val)}")
print(f"Number of test examples: {len(dataset.test)}")
```

```
Number of training examples: 40000
Number of validation examples: 10000
Number of test examples: 10000
```

Observe como foi feita a separação de 10.000 exemplos do conjunto de treinamento original para serem o conjunto de validação. Dessa forma, temos ao final 40.000 exemplos para treinamento, 10.000 exemplos para validação e 10.000 exemplos para teste, com os seus respectivos `DataLoader`'s instanciados.

Também redimensionamos as imagens para 224x224pixels, já preparando o dado para a posterior aplicação na rede convolucional.

1.2 2. Treinando um MLP

Use esta base de dados para treinar um Perceptron Multicamadas, como feito no trabalho anterior com a base MNIST. Escolha um MLP com 2 camadas escondidas. Não perca muito tempo variando a arquitetura porque este problema é difícil sem o uso de convoluções e o resultado não será totalmente satisfatório.

Você pode usar este código, baseado na biblioteca Pytorch Lightning como base para definição da rede:

[3]:
```
%pip install pytorch-lightning
import pytorch_lightning as pl
import torch.nn as nn
from torchmetrics.functional import accuracy

# The model is passed as an argument to the `LightModel` class.
class LightModel(pl.LightningModule):
        def __init__(self,model,lr=1e-5):
                super().__init__()
                self.model = model
                self.lr = lr
        def training_step(self, batch):
                X, y = batch
                y_hat = self.model(X)
                loss = nn.functional.cross_entropy(y_hat, y)
                self.log("train_loss", loss)
```

```
                return loss
        def validation_step(self, batch):
                X, y = batch
                y_hat = self.model(X)
                loss = nn.functional.cross_entropy(y_hat, y)
                self.log("val_loss", loss)
                return loss
        def test_step(self, batch):
                X, y = batch
                y_hat = self.model(X)
                preds = torch.argmax(y_hat, dim=1)
                acc = accuracy(preds, y, task="multiclass", num_classes=10)
                self.log("test_acc", acc)
                loss = nn.functional.cross_entropy(y_hat, y)
                self.log("test_loss", loss)
        def configure_optimizers(self):
                optimizer = torch.optim.Adam(self.parameters(), self.lr)
                return optimizer


arch = nn.Sequential(
                        nn.Flatten(),
                        nn.Linear(3*224*224, 512),
                        nn.ReLU(),
                        nn.Linear(512,128),
                        nn.ReLU(),
                        nn.Linear(128,10)
        )

mlp = LightModel(arch)
```

```
Requirement already satisfied: pytorch-lightning in c:\python311\lib\site-
packages (2.5.1.post0)
Requirement already satisfied: torch>=2.1.0 in c:\python311\lib\site-packages
(from pytorch-lightning) (2.7.1)
Requirement already satisfied: tqdm>=4.57.0 in c:\python311\lib\site-packages
(from pytorch-lightning) (4.67.1)
Requirement already satisfied: PyYAML>=5.4 in c:\python311\lib\site-packages
(from pytorch-lightning) (6.0.2)
Requirement already satisfied: fsspec[http]>=2022.5.0 in c:\python311\lib\site-
packages (from pytorch-lightning) (2025.5.1)
Requirement already satisfied: torchmetrics>=0.7.0 in c:\python311\lib\site-
packages (from pytorch-lightning) (1.7.2)
Requirement already satisfied: packaging>=20.0 in
c:\users\tucol\appdata\roaming\python\python311\site-packages (from pytorch-
lightning) (25.0)
Requirement already satisfied: typing-extensions>=4.4.0 in
c:\python311\lib\site-packages (from pytorch-lightning) (4.14.0)
```

```
Requirement already satisfied: lightning-utilities>=0.10.0 in
c:\python311\lib\site-packages (from pytorch-lightning) (0.14.3)
Requirement already satisfied: aiohttp!=4.0.0a0,!=4.0.0a1 in
c:\python311\lib\site-packages (from fsspec[http]>=2022.5.0->pytorch-lightning)
(3.12.12)
Requirement already satisfied: setuptools in c:\python311\lib\site-packages
(from lightning-utilities>=0.10.0->pytorch-lightning) (65.5.0)
Requirement already satisfied: filelock in c:\python311\lib\site-packages (from
torch>=2.1.0->pytorch-lightning) (3.18.0)
Requirement already satisfied: sympy>=1.13.3 in c:\python311\lib\site-packages
(from torch>=2.1.0->pytorch-lightning) (1.14.0)
Requirement already satisfied: networkx in c:\python311\lib\site-packages (from
torch>=2.1.0->pytorch-lightning) (3.5)
Requirement already satisfied: jinja2 in c:\python311\lib\site-packages (from
torch>=2.1.0->pytorch-lightning) (3.1.6)
Requirement already satisfied: numpy>1.20.0 in c:\python311\lib\site-packages
(from torchmetrics>=0.7.0->pytorch-lightning) (2.3.0)
Requirement already satisfied: colorama in
c:\users\tucol\appdata\roaming\python\python311\site-packages (from
tqdm>=4.57.0->pytorch-lightning) (0.4.6)
Requirement already satisfied: aiohappyeyeballs>=2.5.0 in c:\python311\lib\site-
packages (from aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]>=2022.5.0->pytorch-
lightning) (2.6.1)
Requirement already satisfied: aiosignal>=1.1.2 in c:\python311\lib\site-
packages (from aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]>=2022.5.0->pytorch-
lightning) (1.3.2)
Requirement already satisfied: attrs>=17.3.0 in c:\python311\lib\site-packages
(from aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]>=2022.5.0->pytorch-lightning)
(25.3.0)
Requirement already satisfied: frozenlist>=1.1.1 in c:\python311\lib\site-
packages (from aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]>=2022.5.0->pytorch-
lightning) (1.7.0)
Requirement already satisfied: multidict<7.0,>=4.5 in c:\python311\lib\site-
packages (from aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]>=2022.5.0->pytorch-
lightning) (6.4.4)
Requirement already satisfied: propcache>=0.2.0 in c:\python311\lib\site-
packages (from aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]>=2022.5.0->pytorch-
lightning) (0.3.2)
Requirement already satisfied: yarl<2.0,>=1.17.0 in c:\python311\lib\site-
packages (from aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]>=2022.5.0->pytorch-
lightning) (1.20.1)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in c:\python311\lib\site-
packages (from sympy>=1.13.3->torch>=2.1.0->pytorch-lightning) (1.3.0)
Requirement already satisfied: MarkupSafe>=2.0 in c:\python311\lib\site-packages
(from jinja2->torch>=2.1.0->pytorch-lightning) (3.0.2)
Requirement already satisfied: idna>=2.0 in c:\python311\lib\site-packages (from
yarl<2.0,>=1.17.0->aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]>=2022.5.0->pytorch-
lightning) (3.10)
```

```
Note: you may need to restart the kernel to use updated packages.


[notice] A new release of pip available: 22.3 -> 25.1.1
[notice] To update, run: python.exe -m pip install --upgrade pip
```

1.2.1 Callback personalizada: LastNCheckpoints

A célula define a classe `LastNCheckpoints`, que herda de `ModelCheckpoint` do PyTorch Lightning. Ela serve para manter apenas os **N últimos checkpoints** do treinamento, apagando automaticamente os mais antigos após cada validação. Isso evita o acúmulo excessivo de arquivos de checkpoint no disco, facilitando o gerenciamento de espaço.

Como funciona: - No final de cada validação (`on_validation_end`), a classe busca todos os arquivos `.ckpt` no diretório de checkpoints. - Ordena os arquivos por data de modificação. - Remove os checkpoints mais antigos, mantendo apenas os `keep_last_n` mais recentes.

Essa abordagem é útil para economizar espaço em disco durante experimentos longos ou com muitos checkpoints salvos.

[4]:
```python
from pytorch_lightning.callbacks import ModelCheckpoint
import os
import glob

class LastNCheckpoints(ModelCheckpoint):
    def __init__(self, keep_last_n=5, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.keep_last_n = keep_last_n

    def on_validation_end(self, trainer, pl_module):
        super().on_validation_end(trainer, pl_module)

        # Clean up old checkpoints
        all_ckpts = sorted(
            glob.glob(os.path.join(self.dirpath, "*.ckpt")),
            key=os.path.getmtime
        )
        if len(all_ckpts) > self.keep_last_n:
            for ckpt in all_ckpts[:-self.keep_last_n]:
                os.remove(ckpt)
```

Observe que as imagens são achatadas (transformadas em vetor). Substitua as interrogações pelo tamanho desejado das camadas escondidas.

Neste problema vamos verificar o fenômeno do sobreajuste, e vamos tentar equilibrá-lo pela técnica de parada prematura de treinamento (early-stopping). Por isso foi necessário, a partir dos dados de treinamento, fazer uma nova separação para validação. Quando a função custo (`loss`) no conjunto de validação não diminui num dado número de épocas (o parâmetro `patience`), o treinamento é interrompido. Este trecho de código pode ser útil:

1.2.2 Treinamento com Early Stopping e Checkpoints

Esta célula configura o treinamento do MLP usando PyTorch Lightning, com duas callbacks principais:

- **EarlyStopping:** Interrompe o treinamento automaticamente se a métrica de validação (`val_loss`) não melhorar após um número definido de épocas (`patience=5`). Isso ajuda a evitar sobreajuste.
- **LastNCheckpoints:** Salva checkpoints do modelo a cada época, mas mantém apenas os N mais recentes, economizando espaço em disco.

O código permite alternar entre treinar o modelo (`TRAIN=True`) ou carregar um checkpoint salvo para análise e avaliação. Após o treinamento ou carregamento, o modelo é colocado em modo de avaliação.

[5]:
```
from pytorch_lightning.callbacks import EarlyStopping, ModelCheckpoint
from pytorch_lightning import Trainer
from pytorch_lightning.utilities.model_summary import summarize

checkpoint = LastNCheckpoints(
    dirpath='model1_checkpoints/',
    filename='{epoch:02d}-{val_loss:.4f}',
    every_n_epochs=1,  # Save every epoch
    save_top_k=-1  # Save ALL checkpoints
)

early_stopping = EarlyStopping(
    monitor='val_loss',  # metric to monitor
    patience=5,          # epochs with no improvement after which training will␣
 ↪stop
    mode='min',          # mode for min loss; 'max' if maximizing metric
    min_delta=0.001      # minimum change to qualify as an improvement
)

trainer = Trainer(
    callbacks=[early_stopping, checkpoint],
    max_epochs=50,
)
mlp.train()
TRAIN = False
# TRAIN = True
if TRAIN:
    trainer.fit(
        model=mlp,
        train_dataloaders=train_dataloader,
        val_dataloaders=val_dataloader,
        ckpt_path="model1_checkpoints/checkpoint-epoch=15-val_loss=1.3520.ckpt"
    )
else:
```

```
    print("LOADING MODEL")

    checkpoint = torch.load("model1_checkpoints/epoch=16-val_loss=1.3592.ckpt",␣
 ↪map_location=torch.device('cpu'))  # or 'cuda'

    # Load state dict into your model
    mlp.load_state_dict(checkpoint['state_dict'])

    summary = summarize(mlp, max_depth=2)  # max_depth controls how deep to␣
 ↪show layers
    print(summary)

mlp.eval()
```

```
GPU available: False, used: False
TPU available: False, using: 0 TPU cores
HPU available: False, using: 0 HPUs
c:\Python311\Lib\site-packages\pytorch_lightning\trainer\connectors\logger_conne
ctor\logger_connector.py:76: Starting from v1.9.0, `tensorboardX` has been
removed as a dependency of the `pytorch_lightning` package, due to potential
conflicts with other packages in the ML ecosystem. For this reason,
`logger=True` will use `CSVLogger` as the default logger, unless the
`tensorboard` or `tensorboardX` packages are found. Please `pip install
lightning[extra]` or one of them to enable TensorBoard support by default

LOADING MODEL
  | Name    | Type       | Params | Mode
-------------------------------------------------
0 | model   | Sequential | 77.1 M | train
1 | model.0 | Flatten    | 0      | train
2 | model.1 | Linear     | 77.1 M | train
3 | model.2 | ReLU       | 0      | train
4 | model.3 | Linear     | 65.7 K | train
5 | model.4 | ReLU       | 0      | train
6 | model.5 | Linear     | 1.3 K  | train
-------------------------------------------------
77.1 M    Trainable params
0         Non-trainable params
77.1 M    Total params
308.551   Total estimated model params size (MB)
7         Modules in train mode
0         Modules in eval mode
```

[5]:
```
LightModel(
  (model): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=150528, out_features=512, bias=True)
    (2): ReLU()
```

```
    (3): Linear(in_features=512, out_features=128, bias=True)
    (4): ReLU()
    (5): Linear(in_features=128, out_features=10, bias=True)
  )
)
```

Os parâmetros dados são sugestões. Você agora pode testar o seu modelo, por exemplo, com:

[6]:
```
# Evaluate the model on the test dataset
trainer.test(model=mlp, dataloaders=test_dataloader)
```

```
c:\Python311\Lib\site-
packages\pytorch_lightning\trainer\connectors\data_connector.py:420: Consider
setting `persistent_workers=True` in 'test_dataloader' to speed up the
dataloader worker initialization.
```

```
Testing: |          | 0/? [00:00<?, ?it/s]
```

```
       Test metric             DataLoader 0

        test_acc            0.5370000004768372
        test_loss            1.365114688873291
```

[6]:
```
[{'test_acc': 0.5370000004768372, 'test_loss': 1.365114688873291}]
```

Mais uma vez, procure realizar ajustes, mas não espere um bom desempenho. Como dissemos, é um problema complexo de classificação de imagem, e é difícil fazer o MLP funcionar sozinho. Precisamos de um pré-processamento com base em uma rede convolucional.

1.3 3. Uso da rede VGG16 pré-treinada

Lembre-se que a rede VGG usa como bloco básico cascata de convoluções com filtros 3x3, com "padding" para que a imagem não seja diminuída, seguida de um "max pooling" reduzindo imagens pela metade. O número de mapas vai aumentando e seu tamanho vai diminuindo ao longo de suas 16 camadas. Este é um modelo gigantesco e o treinamento com recursos computacionais modestos levaria dias ou semanas, se é que fosse possível.

No entanto, vamos aproveitar uma característica central das grandes redes convolucionais. Elas podem ser usadas como pré-processamento fixo das imagens, mesmo em um novo problema (lembre-se, a rede VGG original foi treinada na base ImageNet, que tem muitas categorias de imagens).

O código abaixo realiza o download do modelo treinado e configura os seus parâmetros como não ajustáveis.

[7]:
```
from torchvision.models import vgg16
vgg16_model = vgg16(weights="DEFAULT", progress=True)
```

```
for param in vgg16_model.parameters():
        param.requires_grad = False

print(vgg16_model)
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
```

```
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

Redefine o classificador da VGG16 para três camadas lineares (50, 20 e 10 neurônios) com ReLU, adaptando a saída para 10 classes do CIFAR-10. Apenas os parâmetros do classificador são ajustáveis no treinamento. O modelo é então encapsulado na classe `LightModel` para uso com PyTorch Lightning.

[8]:
```
import torch.nn as nn

print(vgg16_model.classifier)
vgg16_model.classifier = nn.Sequential(
    # nn.Flatten(),
    nn.Linear(25088, 50),
    nn.ReLU(),
    nn.Linear(50, 20),
    nn.ReLU(),
    nn.Linear(20, 10)
)

# make sure we unfreeze here for training:
for param in vgg16_model.classifier.parameters():
    param.requires_grad = True

print(vgg16_model.classifier)

cnn = LightModel(vgg16_model)
```

```
Sequential(
  (0): Linear(in_features=25088, out_features=4096, bias=True)
  (1): ReLU(inplace=True)
  (2): Dropout(p=0.5, inplace=False)
  (3): Linear(in_features=4096, out_features=4096, bias=True)
  (4): ReLU(inplace=True)
  (5): Dropout(p=0.5, inplace=False)
  (6): Linear(in_features=4096, out_features=1000, bias=True)
)
Sequential(
  (0): Linear(in_features=25088, out_features=50, bias=True)
  (1): ReLU()
  (2): Linear(in_features=50, out_features=20, bias=True)
  (3): ReLU()
  (4): Linear(in_features=20, out_features=10, bias=True)
```

)

Configura o treinamento do modelo com PyTorch Lightning, usando early stopping para evitar sobreajuste e checkpoints para salvar o progresso. Se `TRAIN` for verdadeiro, treina o modelo; caso contrário, carrega um checkpoint salvo e exibe o resumo da arquitetura.

[9]:
```
from pytorch_lightning.callbacks import EarlyStopping, ModelCheckpoint
from pytorch_lightning import Trainer
from pytorch_lightning.utilities.model_summary import summarize

checkpoint = LastNCheckpoints(
    dirpath='model2_checkpoints/',
    filename='{epoch:02d}-{val_loss:.4f}',
    every_n_epochs=1,  # Save every epoch
    save_top_k=-1  # Save ALL checkpoints
)

early_stopping = EarlyStopping(
    monitor='val_loss',  # metric to monitor
    patience=5,          # epochs with no improvement after which training will␣
 ↪stop
    mode='min',          # mode for min loss; 'max' if maximizing metric
    min_delta=0.001      # minimum change to qualify as an improvement
)

trainer = Trainer(
    callbacks=[early_stopping, checkpoint],
    max_epochs=50,
)

cnn.train()

TRAIN = False
# TRAIN = True
if TRAIN:
    trainer.fit(
        model=cnn,
        train_dataloaders=train_dataloader,
        val_dataloaders=val_dataloader,
        # ckpt_path="model2_checkpoints/checkpoint-epoch=15-val_loss=1.3520.
 ↪ckpt"
    )
else:
    print("LOADING MODEL")

    checkpoint = torch.load(
        "model2_checkpoints/epoch=34-val_loss=0.4055.ckpt",
        map_location=torch.device('cpu')
```

```
    )  # or 'cuda'

    # Load state dict into your model
    cnn.load_state_dict(checkpoint['state_dict'])

    summary = summarize(cnn, max_depth=2)  # max_depth controls how deep to␣
 ↪show layers
    print(summary)

cnn.eval()
```

```
GPU available: False, used: False
TPU available: False, using: 0 TPU cores
HPU available: False, using: 0 HPUs
```

```
LOADING MODEL
  | Name             | Type              | Params | Mode
-------------------------------------------------------------
0 | model            | VGG               | 16.0 M | train
1 | model.features   | Sequential        | 14.7 M | train
2 | model.avgpool    | AdaptiveAvgPool2d | 0      | train
3 | model.classifier | Sequential        | 1.3 M  | train
-------------------------------------------------------------
1.3 M     Trainable params
14.7 M    Non-trainable params
16.0 M    Total params
63.881    Total estimated model params size (MB)
40        Modules in train mode
0         Modules in eval mode
```

[9]:
```
LightModel(
  (model): VGG(
    (features): Sequential(
      (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): ReLU(inplace=True)
      (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (3): ReLU(inplace=True)
      (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
      (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (6): ReLU(inplace=True)
      (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (8): ReLU(inplace=True)
      (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
      (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (11): ReLU(inplace=True)
      (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
      (13): ReLU(inplace=True)
      (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (15): ReLU(inplace=True)
      (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
      (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (18): ReLU(inplace=True)
      (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (20): ReLU(inplace=True)
      (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (22): ReLU(inplace=True)
      (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
      (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (25): ReLU(inplace=True)
      (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (27): ReLU(inplace=True)
      (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (29): ReLU(inplace=True)
      (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    )
    (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
    (classifier): Sequential(
      (0): Linear(in_features=25088, out_features=50, bias=True)
      (1): ReLU()
      (2): Linear(in_features=50, out_features=20, bias=True)
      (3): ReLU()
      (4): Linear(in_features=20, out_features=10, bias=True)
    )
  )
)
```

1.3.1 Avalia o modelo

[10]:
```
trainer.test(model=cnn, dataloaders=test_dataloader)
```

```
Testing: |          | 0/? [00:00<?, ?it/s]
```

```
       Test metric             DataLoader 0

        test_acc             0.859499990940094
        test_loss            0.420192688703537
```

[10]:
```
[{'test_acc': 0.859499990940094, 'test_loss': 0.420192688703537}]
```

1.4 4. Transfer Learning com ResNet18

Agora, vamos testar o uso de outra rede convolucional pré-treinada como base para o nosso classificador. Em vez da VGG16, utilizaremos a **ResNet18**, que é uma arquitetura moderna e eficiente, bastante utilizada em tarefas de visão computacional.

Assim como antes, vamos ajustar apenas o classificador final para as 10 classes do CIFAR-10 e aplicar **regularização L2 (weight decay)** durante o treinamento, buscando reduzir o sobreajuste e melhorar a generalização do modelo.

[14]:
```python
from torchvision.models import resnet18

# Carrega a ResNet18 pré-treinada
resnet = resnet18(weights='IMAGENET1K_V1')
```

[15]:
```python
import torch.nn as nn

# Congela todas as camadas convolucionais
for param in resnet.parameters():
    param.requires_grad = False

# Substitui o classificador final para 10 classes (CIFAR-10)
num_ftrs = resnet.fc.in_features
resnet.fc = nn.Linear(num_ftrs, 10)
```

Abaixo, definimos a classe `LightModel` para usar a `ResNet18` no framework do PyTorch Lightning, agora com `weight_decay` (L2 regularização) configurado no otimizador.

[16]:
```python
class LightModelResNet(pl.LightningModule):
    def __init__(self, model, lr=1e-4, weight_decay=1e-2):
        super().__init__()
        self.model = model
        self.lr = lr
        self.weight_decay = weight_decay

    def training_step(self, batch, batch_idx):
        X, y = batch
        y_hat = self.model(X)
        loss = nn.functional.cross_entropy(y_hat, y)
        self.log("train_loss", loss)
        return loss

    def validation_step(self, batch, batch_idx):
        X, y = batch
        y_hat = self.model(X)
        loss = nn.functional.cross_entropy(y_hat, y)
        self.log("val_loss", loss)
```

```
        return loss

    def test_step(self, batch, batch_idx):
        X, y = batch
        y_hat = self.model(X)
        preds = torch.argmax(y_hat, dim=1)
        acc = accuracy(preds, y, task="multiclass", num_classes=10)
        loss = nn.functional.cross_entropy(y_hat, y)
        self.log("test_acc", acc)
        self.log("test_loss", loss)

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(
            self.parameters(),
            lr=self.lr,
            weight_decay=self.weight_decay
        )
        return optimizer
```

[17]:
```
resnet_model = LightModelResNet(resnet)

trainer = pl.Trainer(
    max_epochs=10,
    accelerator="auto",
    devices=1,
    enable_progress_bar=True
)

trainer.fit(resnet_model, train_dataloaders=train_dataloader,␣
 ↪val_dataloaders=val_dataloader)
```

```
Using default `ModelCheckpoint`. Consider installing `litmodels` package to
enable `LitModelCheckpoint` for automatic upload to the Lightning model
registry.
GPU available: False, used: False
TPU available: False, using: 0 TPU cores
HPU available: False, using: 0 HPUs

  | Name  | Type   | Params | Mode
-----------------------------------------
0 | model | ResNet | 11.2 M | train
-----------------------------------------
5.1 K     Trainable params
11.2 M    Non-trainable params
11.2 M    Total params
44.727    Total estimated model params size (MB)
68        Modules in train mode
0         Modules in eval mode
```

```
Sanity Checking: |          | 0/? [00:00<?, ?it/s]

c:\Python311\Lib\site-
packages\pytorch_lightning\trainer\connectors\data_connector.py:420: Consider
setting `persistent_workers=True` in 'val_dataloader' to speed up the dataloader
worker initialization.
c:\Python311\Lib\site-
packages\pytorch_lightning\trainer\connectors\data_connector.py:420: Consider
setting `persistent_workers=True` in 'train_dataloader' to speed up the
dataloader worker initialization.

Training: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

`Trainer.fit` stopped: `max_epochs=10` reached.
```

[18]:
```
trainer.test(resnet_model, dataloaders=test_dataloader)
```

```
Testing: |          | 0/? [00:00<?, ?it/s]
```

Test metric	DataLoader 0
test_acc	0.7918000221252441
test_loss	0.635003924369812

[18]:
```
[{'test_acc': 0.7918000221252441, 'test_loss': 0.635003924369812}]
```

1.5 Conclusão

Este trabalho teve como objetivo desenvolver e treinar uma rede neural convolucional (CNN) baseada na arquitetura VGG16, utilizando a estrutura do PyTorch Lightning para facilitar o gerenciamento do treinamento e a organização do código.

1.5.1 Principais Etapas

- **Preparação dos dados**: Os dados foram carregados, normalizados e divididos em conjuntos de treino, validação e teste.
- **Modelagem**: Foi utilizada a arquitetura VGG16 com modificações específicas, encapsulada em uma classe customizada e adaptada ao PyTorch Lightning.
- **Treinamento**: O modelo foi treinado com callbacks de *EarlyStopping* para evitar overfitting e *ModelCheckpoint* para salvar os melhores estados do modelo.
- **Avaliação**: O modelo com melhor desempenho na validação (`val_loss  0.4055`) foi carregado e avaliado no conjunto de teste.

1.5.2 Resultados

- O uso de técnicas de regularização, como *early stopping*, contribuiu para a estabilidade do treinamento e para a generalização do modelo.
- O checkpoint carregado indica que o modelo alcançou um bom equilíbrio entre performance e generalização, evitando overfitting.
- O resumo da arquitetura mostrou que o modelo possui uma estrutura profunda, adequada para tarefas de classificação complexas.

1.5.3 Considerações Finais

O experimento demonstrou que é possível adaptar arquiteturas consagradas, como a VGG16, a diferentes conjuntos de dados e problemas utilizando ferramentas modernas como PyTorch Lightning. O pipeline implementado é modular, reutilizável e preparado para experimentos futuros com diferentes arquiteturas, hiperparâmetros e estratégias de treinamento.