Trabalho Computacional 3. Rede Convolucional e Transfer Learning

- Gabriel Caixeta Romero - 232036896
- Vitor Amorim Mello - 231037048

Introdução

Neste trabalho exploramos a aplicação de redes neurais para classificação de imagens da base CIFAR-10.
Nosso objetivo foi comparar o desempenho de um Perceptron Multicamadas (MLP) com o de uma rede convolucional pré-treinada (VGG16), através da técnica de Transfer Learning.

A base CIFAR-10 é composta por 60.000 imagens 32x32 coloridas, com 10 classes diferentes de objetos.

O trabalho foi dividido em três partes principais:

- Treinamento de um MLP simples, com imagens redimensionadas para 64x64 pixels.
- Treinamento de uma rede VGG16 pré-treinada, com imagens redimensionadas para 224x224 pixels.
- Comparação de resultados entre MLP e VGG16, com observação dos fenômenos de overfitting e uso de técnicas como Early Stopping.

```
import torch
import torchvision
import torchvision.transforms as transforms


class CIFAR10():
    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
        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_s
        self.test = torchvision.datasets.CIFAR10(
            root=root, train=False, transform=trans, download=True)

dataset = CIFAR10(root="./data/", resize=(64, 64))
```

```
train_dataloader = torch.utils.data.DataLoader(dataset.train, batch_size=64, shuf
val_dataloader = torch.utils.data.DataLoader(dataset.val, batch_size=64, shuffle=
test_dataloader = torch.utils.data.DataLoader(dataset.test, batch_size=64, shuffl

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.

2. Treinando um MLP

Nesta etapa, treinamos um Perceptron Multicamadas (MLP) sobre a base CIFAR-10, com o objetivo de avaliar seu desempenho em um problema de classificação de imagens.

Como redes MLP não exploram a estrutura espacial das imagens, e para tornar o treinamento mais leve, redimensionamos as imagens para 64x64 pixels.

Utilizamos uma arquitetura com duas camadas escondidas, conforme proposto no trabalho, e aplicamos a técnica de Early Stopping para evitar overfitting.

O objetivo não era alcançar o melhor desempenho possível, mas sim comparar a capacidade do MLP com uma rede convolucional, que é naturalmente mais adequada a esse tipo de tarefa.

O código foi implementado com o auxílio da biblioteca PyTorch Lightning, que facilita a organização do treinamento e monitoramento das métricas.

```
%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 * 64 * 64, 256),
        nn.ReLU(),
        nn.Linear(256, 128),
        nn.ReLU(),
        nn.Linear(128, 10)
    )

    mlp = LightModel(arch)
```

```
Requirement already satisfied: pytorch-lightning in /usr/local/lib/python3.11/
Requirement already satisfied: torch>=2.1.0 in /usr/local/lib/python3.11/dist-
Requirement already satisfied: tqdm>=4.57.0 in /usr/local/lib/python3.11/dist-
Requirement already satisfied: PyYAML>=5.4 in /usr/local/lib/python3.11/dist-p
Requirement already satisfied: fsspec>=2022.5.0 in /usr/local/lib/python3.11/d
Requirement already satisfied: torchmetrics>=0.7.0 in /usr/local/lib/python3.1
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.11/di
Requirement already satisfied: typing-extensions>=4.4.0 in /usr/local/lib/pyth
Requirement already satisfied: lightning-utilities>=0.10.0 in /usr/local/lib/p
Requirement already satisfied: aiohttp!=4.0.0a0,!=4.0.0a1 in /usr/local/lib/py
Requirement already satisfied: setuptools in /usr/local/lib/python3.11/dist-pa
Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-pack
Requirement already satisfied: networkx in /usr/local/lib/python3.11/dist-pack
Requirement already satisfied: jinja2 in /usr/local/lib/python3.11/dist-packag
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.4.127 in /usr/local/
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.4.127 in /usr/loca
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.4.127 in /usr/local/
Requirement already satisfied: nvidia-cudnn-cu12==9.1.0.70 in /usr/local/lib/p
Requirement already satisfied: nvidia-cublas-cu12==12.4.5.8 in /usr/local/lib/
Requirement already satisfied: nvidia-cufft-cu12==11.2.1.3 in /usr/local/lib/p
Requirement already satisfied: nvidia-curand-cu12==10.3.5.147 in /usr/local/li
Requirement already satisfied: nvidia-cusolver-cu12==11.6.1.9 in /usr/local/li
Requirement already satisfied: nvidia-cusparse-cu12==12.3.1.170 in /usr/local/
```

```
Requirement already satisfied: nvidia-cusparselt-cu12==0.6.2 in /usr/local/lil
Requirement already satisfied: nvidia-nccl-cu12==2.21.5 in /usr/local/lib/pytl
Requirement already satisfied: nvidia-nvtx-cu12==12.4.127 in /usr/local/lib/p
Requirement already satisfied: nvidia-nvjitlink-cu12==12.4.127 in /usr/local/
Requirement already satisfied: triton==3.2.0 in /usr/local/lib/python3.11/dis
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.11/dis
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.1
Requirement already satisfied: numpy>1.20.0 in /usr/local/lib/python3.11/dist
Requirement already satisfied: aiohappyeyeballs>=2.3.0 in /usr/local/lib/pyth
Requirement already satisfied: aiosignal>=1.1.2 in /usr/local/lib/python3.11/
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.11/dis
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/python3.11
Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/python3.
Requirement already satisfied: propcache>=0.2.0 in /usr/local/lib/python3.11/
Requirement already satisfied: yarl<2.0,>=1.17.0 in /usr/local/lib/python3.11
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.11/d
Requirement already satisfied: idna>=2.0 in /usr/local/lib/python3.11/dist-pa
```

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:

```
from pytorch_lightning.callbacks import EarlyStopping
from pytorch_lightning import Trainer

early_stopping = EarlyStopping(
    monitor='val_loss',  # metric to monitor
    patience=5,          # epochs with no improvement after which training will s
    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], max_epochs=12)
trainer.fit(model=mlp, train_dataloaders=train_dataloader, val_dataloaders=val_da
```

```
INFO:pytorch_lightning.utilities.rank_zero:Using default `ModelCheckpoint`. C
INFO:pytorch_lightning.utilities.rank_zero:GPU available: False, used: False
INFO:pytorch_lightning.utilities.rank_zero:TPU available: False, using: 0 TPU
INFO:pytorch_lightning.utilities.rank_zero:HPU available: False, using: 0 HPU
INFO:pytorch_lightning.callbacks.model_summary:
  | Name  | Type       | Params | Mode
-------------------------------------------
0 | model | Sequential | 3.2 M  | train
-------------------------------------------
3.2 M     Trainable params
0         Non-trainable params
3.2 M     Total params
12.721    Total estimated model params size (MB)
7         Modules in train mode
0         Modules in eval mode
```

Epoch 11: 100% 625/625 [01:00<00:00, 10.25it/s, v_num=4]

Testando o modelo:

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

Testing DataLoader 0: 100% 157/157 [00:15<00:00, 9.84it/s]

Test metric	DataLoader 0
test_acc	0.489300012588501
test_loss	1.47456693649292

```
[{'test_acc': 0.489300012588501, 'test_loss': 1.47456693649292}]
```

Resultados do MLP

Após o treinamento do MLP com imagens redimensionadas para 64x64 pixels e com 12 épocas máximas, o modelo atingiu uma acurácia de aproximadamente 50% no conjunto de teste.

Este resultado é consistente com o esperado: como o MLP não explora as relações espaciais das imagens, sua capacidade de aprendizado em um problema de classificação de imagens complexas como CIFAR-10 é limitada.

A acurácia de 50% já demonstra que o MLP consegue identificar alguns padrões básicos, mas não atinge um desempenho competitivo.

3. Uso da rede VGG16 pré-treinada

Nesta etapa, utilizamos a rede VGG16 pré-treinada como base para a tarefa de classificação das imagens da base CIFAR-10.

A arquitetura VGG16 foi originalmente treinada no ImageNet, o que permite que suas camadas convolucionais já contenham filtros capazes de extrair representações eficazes das imagens.

No nosso experimento, congelamos as camadas convolucionais da VGG16 e treinamos apenas um novo bloco de classificação (com camadas densas), adaptado para as 10 classes da base CIFAR-10.

Com o objetivo de otimizar o tempo de execução e ainda assim obter uma comparação válida com o MLP, limitamos o treinamento a no máximo 5 épocas, utilizando a técnica de Early Stopping.

Mesmo com essa configuração simples, a VGG16 foi capaz de superar significativamente o desempenho do MLP, evidenciando o poder do Transfer Learning para este tipo de tarefa.

```
# Importações necessárias (caso ainda não tenha feito)
from torchvision.models import vgg16
import torch.nn as nn
from pytorch_lightning import Trainer
from pytorch_lightning.callbacks import EarlyStopping
import matplotlib.pyplot as plt

# Carregar a VGG16 pré-treinada
vgg16_model = vgg16(weights="DEFAULT", progress=True)

# Congelar os parâmetros da VGG16
for param in vgg16_model.parameters():
    param.requires_grad = False

# Substituir o bloco classifier da VGG16
vgg16_model.classifier = nn.Sequential(
    nn.Flatten(),
    nn.Linear(25088, 50),
    nn.ReLU(),
    nn.Linear(50, 20),
    nn.ReLU(),
    nn.Linear(20, 10)
)

# Definir o modelo no LightModel
vgg16_light_model = LightModel(vgg16_model)

# Configurar EarlyStopping
early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=5,
    mode='min',
    min_delta=0.001
)

# Criar o Trainer (com max_epochs=20)
trainer = Trainer(callbacks=[early_stopping], max_epochs=5)

# Treinar o modelo
```

```
trainer.fit(model=vgg16_light_model, train_dataloaders=train_dataloader, val_data

# Avaliar no conjunto de teste
trainer.test(model=vgg16_light_model, dataloaders=test_dataloader)
```

```
INFO:pytorch_lightning.utilities.rank_zero:Using default `ModelCheckpoint`. C
INFO:pytorch_lightning.utilities.rank_zero:GPU available: False, used: False
INFO:pytorch_lightning.utilities.rank_zero:TPU available: False, using: 0 TPU
INFO:pytorch_lightning.utilities.rank_zero:HPU available: False, using: 0 HPU
INFO:pytorch_lightning.callbacks.model_summary:
  | Name  | Type | Params | Mode
-------------------------------------
0 | model | VGG  | 16.0 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)
41        Modules in train mode
0         Modules in eval mode
```

Epoch 4: 100% 625/625 [50:46<00:00, 0.21it/s, v_num=5]

```
INFO:pytorch_lightning.utilities.rank_zero:`Trainer.fit` stopped: `max_epochs=
```

Testing DataLoader 0: 100% 157/157 [09:54<00:00, 0.26it/s]

Test metric	DataLoader 0
test_acc	0.7378000020980835
test_loss	0.8328250050544739

Resultados da VGG16 com Transfer Learning

Na segunda etapa do trabalho, utilizamos a rede VGG16 pré-treinada no ImageNet, mantendo as camadas convolucionais congeladas e treinando apenas um novo bloco de classificação.

O treinamento foi realizado com no máximo 5 épocas, de forma a otimizar o tempo de execução. Mesmo com apenas 5 épocas, a rede VGG16 alcançou uma acurácia de aproximadamente 74% no conjunto de teste.

Este resultado demonstra de forma clara a superioridade do Transfer Learning para este tipo de tarefa: a VGG16, com seus filtros convolucionais já treinados, foi capaz de extrair representações muito mais eficazes das imagens, permitindo um desempenho significativamente superior ao MLP.

Conclusão

Os experimentos realizados demonstraram de forma clara as vantagens do uso de redes convolucionais e da técnica de Transfer Learning para tarefas de classificação de imagens.

O Perceptron Multicamadas (MLP), mesmo com arquitetura simples e otimizada, atingiu um desempenho limitado (~50% de acurácia), devido à sua incapacidade de explorar as estruturas espaciais das imagens.

Por outro lado, a VGG16 pré-treinada, mesmo com apenas 5 épocas de ajuste do classificador, foi capaz de alcançar ~74% de acurácia, mostrando o poder das representações aprendidas nas camadas convolucionais.

Este trabalho reforça a importância de utilizar arquiteturas adequadas ao tipo de dado, e evidencia o potencial do Transfer Learning como ferramenta para melhorar o desempenho de modelos em tarefas específicas.