TC3

June 15, 2025

1 Trabalho Computacional 3. Rede Convolucional e Transfer Learning

1.0.1 Nome: Antonio Leonardo Souto Gomes

1.0.2 Matrícula: 211027607

1.1 Introdução

Este trabalho computacional explora o uso de redes neurais para classificação de imagens, utilizando a base de dados CIFAR10. O estudo abrange desde a implementação de um Perceptron Multicamadas (MLP) "do zero" até a aplicação de técnicas avançadas como o transfer learning com uma Rede Neural Convolucional (CNN) pré-treinada, especificamente a VGG16.|

1.2 Implementação(MLP from scratch)

O primeiro passo é o de importar algumas importantes bibliotecas python:

[1]:
```
import torch
import torchvision
import torchvision.transforms as transforms
import pytorch_lightning as pl
import torch.nn as nn
from torchmetrics.functional import accuracy
from pytorch_lightning.callbacks import EarlyStopping
from pytorch_lightning import Trainer
from torchvision.models import vgg16
from torchvision.models import mobilenet_v3_large
```

Inicialmente, aborda-se a preparação dos dados da base CIFAR10, que consiste em 60.000 imagens 32x32 coloridas de 10 categorias distintas. As imagens são redimensionadas para 224x224 pixels e a base é dividida em conjuntos de treinamento (40.000 exemplos), validação (10.000 exemplos) e teste (10.000 exemplos), com seus respectivos DataLoaders já configurados.

[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)
        train_set_size = int(len(self.train) * 0.8)
        valid_set_size = len(self.train) - train_set_size
        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=6,persistent_workers=True)
val_dataloader = torch.utils.data.DataLoader(dataset.val, batch_size=64,␣
 ↪shuffle=False, num_workers=6, persistent_workers=True)
test_dataloader = torch.utils.data.DataLoader(dataset.test, batch_size=64,␣
 ↪shuffle=False, num_workers=6, persistent_workers=True)
```

Em seguida, o trabalho foca no treinamento de um Perceptron Multicamadas (MLP) com duas camadas escondidas. A estrutura de treinamento e avaliação do modelo é facilitada pela biblioteca PyTorch Lightning, através da classe LightModel. Esta classe encapsula as etapas de treinamento (training_step), validação (validation_step) e teste (test_step), além de configurar o otimizador Adam. A training_step e validation_step calculam a loss de entropia cruzada, enquanto a test_step também calcula a acurácia.

[3]:
```
class LightModel(pl.LightningModule):
    def __init__(self, model, lr=1e-5, weight_decay=0.0,l1_lambda=0.0):
        super().__init__()
        self.model = model
        self.lr = lr
        self.weight_decay = weight_decay
        self.l1_lambda = l1_lambda

    def training_step(self, batch):
        X, y = batch
        y_hat = self.model(X)
        loss = nn.functional.cross_entropy(y_hat, y)
        if self.l1_lambda > 0:
            l1_penalty = sum(torch.norm(param, 1) for param in self.model.
 ↪parameters())
            loss += self.l1_lambda * l1_penalty
        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(), lr=self.lr,␣
↪weight_decay=self.weight_decay)
        return optimizer
```

Em seguida, o trabalho foca no treinamento de um Perceptron Multicamadas (MLP) com duas camadas escondidas. A arquitetura deste MLP, definida em arch, é composta por uma camada Flatten para transformar as imagens 2D em vetores 1D, seguida por duas camadas densas (nn.Linear) com ativação ReLU, e uma camada de saída final com 10 neurônios para as classes do CIFAR10. Este modelo é então encapsulado pela classe LightModel do PyTorch Lightning para facilitar o treinamento e avaliação.

[4]:
```
arch = nn.Sequential(
                        nn.Flatten(),
                        nn.Linear(3*224*224,256),
                        nn.ReLU(),
                        nn.Linear(256,64),
                        nn.ReLU(),
                        nn.Linear(64,10)
        )

mlp = LightModel(arch)
```

Em seguida, para controlar o sobreajuste e otimizar o processo de treinamento, a técnica de early stopping é implementada. Utilizando o EarlyStopping do PyTorch Lightning, o treinamento é monitorado pela loss de validação (val_loss). Se essa métrica não apresentar melhoria (diminuição mínima de 0.001) por 5 épocas consecutivas, o treinamento é automaticamente interrompido. Um Trainer do PyTorch Lightning é então configurado com este callback de early stopping e um limite máximo de 50 épocas, sendo responsável por coordenar o processo de ajuste do modelo mlp com os dataloaders de treinamento e validação.

[ ]:
```
early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=5,
```

```
    mode='min',
    min_delta=0.001
)

trainer = Trainer(callbacks=[early_stopping], max_epochs=10)
trainer.fit(model=mlp, train_dataloaders=train_dataloader,␣
 ↪val_dataloaders=val_dataloader)
```

Por fim, para avaliar o desempenho final do modelo treinado, o trainer é utilizado para executar a avaliação no conjunto de teste. O método trainer.test() é chamado com o modelo mlp e o test_dataloader, que irá calcular métricas como a loss e a acurácia no conjunto de dados não visto durante o treinamento e validação.

[6]:
```
trainer.test(model=mlp, dataloaders=test_dataloader)
```

```
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

Testing: |
 ↪                | 0/? [00:00<…
```

Test metric	DataLoader 0
test_acc	0.5134000182151794
test_loss	1.3991731405258179

[6]:
```
[{'test_acc': 0.5134000182151794, 'test_loss': 1.3991731405258179}]
```

1.3 Implementação(VGG-16)

Após as etapas iniciais, o trabalho avança para a aplicação do transfer learning utilizando a rede VGG16. Neste estágio, uma VGG16 pré-treinada na base ImageNet é empregada como um extrator de características fixo, com seus parâmetros de convolução congelados para evitar o retreinamento. O bloco classificador original da VGG16 é então substituído por um novo Perceptron Multicamadas (MLP), personalizado para as 10 classes da base de dados CIFAR10. Somente os pesos deste novo classificador são treinados, demonstrando como o conhecimento pré-existente de grandes Redes Neurais Convolucionais (CNNs) pode ser eficientemente adaptado e reutilizado para problemas específicos de classificação de imagens, prometendo um desempenho superior ao MLP treinado "do zero".

[7]:
```
vgg16_model = vgg16(weights="DEFAULT", progress=True)

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

O trecho de código demonstra a customização do classificador de uma rede VGG16 pré-treinada para uma tarefa específica de classificação de imagens. O vgg16_model.classifier original é substituído por uma nova sequência de camadas. Esta nova sequência começa com nn.Flatten(), responsável

por transformar a saída do avgpool (que é de 7×7×512 características) em um vetor unidimensional. Em seguida, uma camada nn.Linear mapeia essas 25088 características para 50 neurônios, seguida por uma função de ativação nn.ReLU(). Outra camada nn.Linear então projeta de 50 para 20 neurônios, novamente seguida por uma nn.ReLU(). Por fim, uma camada nn.Linear final com 10 neurônios é definida, correspondendo às 10 classes de saída do CIFAR10. O vgg16_model modificado é então encapsulado por LightModel, preparando-o para o treinamento onde apenas os parâmetros do novo classificador serão ajustados.

[8]:
```
vgg16_model.classifier = nn.Sequential(
    nn.Flatten(),
    nn.Linear(7*7*512, 50),
    nn.ReLU(),
    nn.Linear(50, 20),
    nn.ReLU(),
    nn.Linear(20, 10)
)

vgg_transfer_model = LightModel(vgg16_model)
```

[ ]:
```
early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=3,
    mode='min',
    min_delta=0.001
)

trainer = Trainer(accelerator='gpu', devices=1, callbacks=[early_stopping],␣
 ↪max_epochs=5)
trainer.fit(model=vgg_transfer_model, train_dataloaders=train_dataloader,␣
 ↪val_dataloaders=val_dataloader)
```

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

```
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

Testing: |
 ↪                 | 0/? [00:00<…                                                  ␣
```

Test metric	DataLoader 0
test_acc	0.8030999898910522
test_loss	0.6667813062667847

[10]:
```
[{'test_acc': 0.8030999898910522, 'test_loss': 0.6667813062667847}]
```

1.3.1 Adcionando regularização L1 e L2 ao modelo VGG16

A regularização é uma técnica essencial para evitar overfitting em modelos de aprendizado profundo, especialmente em arquiteturas grandes como o VGG16, que possui milhões de parâmetros.

[11]:
```
vgg16_model_L1L2 = vgg16(weights="DEFAULT", progress=True)

for param in vgg16_model_L1L2.parameters():
        param.requires_grad = False
```

A regularização L2 adiciona uma penalização proporcional ao quadrado dos pesos (w^2) na função de custo. No PyTorch, isso é facilmente aplicada via parâmetro weight_decay do otimizador. Esse método incentiva os pesos a serem pequenos, promovendo modelos mais simples e melhorando a generalização. Já a regularização L1 penaliza a soma dos valores absolutos dos pesos (|w|). Ela tende a gerar pesos exatamente zero, promovendo esparsidade no modelo, o que pode ser útil para simplificar a rede e interpretar quais neurônios são mais importantes.

[12]:
```
vgg16_model_L1L2.classifier = nn.Sequential(
    nn.Flatten(),
    nn.Linear(7*7*512, 50),
    nn.ReLU(),
    nn.Linear(50, 20),
    nn.ReLU(),
    nn.Linear(20, 10)
)

vgg_L1L2_transfer_model = LightModel(vgg16_model_L1L2, weight_decay=1e-4,␣
↪l1_lambda=1e-5)
```

[ ]:
```
trainer = Trainer(accelerator='gpu',devices=1,max_epochs=5)
trainer.
↪fit(model=vgg_L1L2_transfer_model,train_dataloaders=train_dataloader,val_dataloaders=val_da
```

[14]:
```
trainer.test(model=vgg_L1L2_transfer_model, dataloaders=test_dataloader)
```

```
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

Testing: |
↪                | 0/? [00:00<…
```

Test metric	DataLoader 0
test_acc	0.8015000224113464
test_loss	0.6867914795875549

[14]:
```
[{'test_acc': 0.8015000224113464, 'test_loss': 0.6867914795875549}]
```

O VGG16 é uma rede profunda com muitas camadas e parâmetros, o que aumenta o risco de overfitting. Aplicar regularização L2 (weight decay) é uma prática comum para melhorar a robustez do modelo. A L1 pode ser usada para incentivar esparsidade, embora seja menos frequente em redes convolucionais profundas.

Combinando essas regularizações e outras técnicas como Dropout, conseguimos controlar melhor o aprendizado do VGG16 e alcançar melhores resultados em tarefas de visão computacional.

1.3.2 Adcionando Dropout ao modelo VGG16

O Dropout é uma técnica de regularização que ajuda a evitar o overfitting em redes neurais profundas, como o VGG16, que possuem muitos parâmetros e podem facilmente memorizar os dados de treinamento.

[15]:
```
vgg16_model_drop = vgg16(weights="DEFAULT", progress=True)

for param in vgg16_model_drop.parameters():
        param.requires_grad = False
```

Durante o treinamento, o Dropout desliga aleatoriamente uma porcentagem dos neurônios em cada camada (normalmente entre 20% e 50%). Isso força a rede a não depender excessivamente de neurônios específicos, tornando o modelo mais robusto e capaz de generalizar melhor para dados novos.

[16]:
```
vgg16_model_drop.classifier = nn.Sequential(
    nn.Flatten(),
    nn.Linear(7*7*512, 50),
    nn.ReLU(),
    nn.Dropout(p=0.5),
    nn.Linear(50, 20),
    nn.ReLU(),
    nn.Dropout(p=0.3),
    nn.Linear(20, 10)
)

vgg_drop_transfer_model = LightModel(vgg16_model_drop)
```

[ ]:
```
trainer = Trainer(accelerator='gpu', devices=1, max_epochs=5)
trainer.fit(model=vgg_drop_transfer_model, train_dataloaders=train_dataloader,␣
 ↪val_dataloaders=val_dataloader)
```

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

```
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

Testing: |
 ↪                 | 0/? [00:00<…                                              ␣
```

```
        Test metric               DataLoader 0
```

```
test_acc              0.7663000226020813
test_loss             1.0494647026062012
```

[18]:
```
[{'test_acc': 0.7663000226020813, 'test_loss': 1.0494647026062012}]
```

No VGG16 original, o Dropout é aplicado principalmente nas camadas totalmente conectadas (fully connected) no final da rede. Isso ajuda a reduzir a co-adaptação dos neurônios e melhora o desempenho em tarefas de classificação de imagens.

1.4 Implementação(MobileNetV3)

O MobileNetV3 é uma arquitetura de rede neural convolucional desenvolvida para ser eficiente e leve, especialmente voltada para dispositivos com recursos limitados, como smartphones e dispositivos IoT. Ela combina técnicas modernas como convoluções separáveis por profundidade (depthwise separable convolutions), módulos de atenção (SE blocks) e busca neural automatizada para otimizar o desempenho e a velocidade.

[19]:
```
mobilenet_model = mobilenet_v3_large(weights="DEFAULT", progress=True)

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

[20]:
```
mobilenet_model.classifier = nn.Sequential(
    nn.Linear(mobilenet_model.classifier[0].in_features, 50),
    nn.ReLU(),
    nn.Linear(50, 20),
    nn.ReLU(),
    nn.Linear(20, 10)
)
```

[21]:
```
mobilenet_transfer_model = LightModel(mobilenet_model)
```

[ ]:
```
early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=3,
    mode='min',
    min_delta=0.001
)

trainer =␣
 ↪Trainer(accelerator='gpu',devices=1,callbacks=[early_stopping],max_epochs=50)
trainer.
 ↪fit(model=mobilenet_transfer_model,train_dataloaders=train_dataloader,val_dataloaders=val_d
```

[23]:
```
trainer.test(model=mobilenet_transfer_model, dataloaders=test_dataloader)
```

```
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
```

```
Testing: |
  ↪                 | 0/? [00:00<…
```

Test metric	DataLoader 0
test_acc	0.7414000034332275
test_loss	0.7419145703315735

[23]:
```
[{'test_acc': 0.7414000034332275, 'test_loss': 0.7419145703315735}]
```

2 Conclusão

Este trabalho computacional foi fundamental para a familiarização e aprimoramento no uso de diversas ferramentas essenciais no desenvolvimento de modelos de aprendizado profundo. Exploramos na prática como essas ferramentas influenciam o desempenho dos modelos. A biblioteca PyTorch Lightning, em particular, mostrou-se bastante eficiente ao simplificar e organizar o processo de treinamento, oferecendo diversas facilidades.

Além disso, aprofundamos nosso entendimento tanto na construção de uma MLP do zero quanto na utilização de arquiteturas pré-treinadas, o que nos permitiu observar diretamente o impacto dessas escolhas no desempenho final dos modelos. Por fim, experimentamos diferentes técnicas de regularização, demonstrando como elas contribuem para o aprimoramento do treinamento e a melhora da capacidade de generalização das redes neurais.