
# Relatório de Atividade Prática

## Deep Learning

IA1s2024

**Aluno**
Rennam Faria - RA 164933

São José dos Campos
2024

# 1 Introdução

Primeiramente, adotaremos os processos padrões de mineração de dados para que todos os passos sejam corretos e que não ocorra nenhuma implicação futura no manuseamento desses dados, assim faremos a etapa de conhecimento do domínio, pré-processamento, pós-processamento e, por fim, análise dos resultados. No código usaremos o DeepLeraning e a base de dados "UC Merced – Land Use Classification"[1], com o objetivo de treinar nossa IA da melhor forma possível para classificar as imagens dadas a ela e dizer qual classe a mais pertence. A partir daí, iniciaremos o processo de manipulação dos dados, transformando-os e ajustando-os conforme necessário, até que possamos começar a rodar nossa IA para verificar a precisão dela.

# 2 Conhecimento do Domínio

Com a base de dados da UC Merced, podemos começar a analisá-los para ver o que precisa ser feito futuramente no processo de Pré-processamento.

A UC Mercedes possui uma base de dados de imagens via satélites tiradas pela "USGS National Map Urban Area Imagery", disponibilizada para propósito de pesquisas. O total de imagens disponibilizadas é de 2100 imagens, de tamanho 256x256 pixels, possuindo 21 classes diferentes de imagens, sendo elas: "agricultural", "airplane", "baseballdiamond", "beach", "buildings", "chaparral", "denseresidential", "forest", "freeway", "golfcourse", "harbor", "intersection", "mediumresidential", "mobilehomepark", "overpass", "parkinglot", "river", "runway", "sparseresidential", "storagetanks" e "tenniscourt".

E com esses dados devemos ajustar da melhor forma possível, utilizar o modelo para treinar com esses dados, por final devemos achar os melhores parâmetros que dê a melhor precisão possível dos dados e possamos analisar os resultados finais para tirar conclusões dos dados e modelo usado.

# 3 Pré-processamento

## 3.1 Predefinições

Em geral, as predefinições utilizadas antes de modelarmos o código foi feito baseado no que já tínhamos conhecimento do dataset anteriormente, são os objetivos básicos que a nossa IA deve ter comprir ou que devemos fazer antes de começar a treiná-la. Como as 50 épocas, as 21 classes que ela possui, pre-setar o dataset para tensor pelo transform e assim por diante

Figura 1 - Predefinições iniciais usado nos treinamentos

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

num_epochs = 50
batch_size = 32
learning_rate = 0.01

numb_class = 21
numb_images = 100     # * numb_classes = total(2100)
image_size = 256      # 256 x 256 pixel

transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
```

## 3.2 Extração do Dataset

Para a extração de imagens, foi baixado o dataset disponibilizado no site[1] e colocado em um drive google, assim utilizando o collab foram extraídos pastas classificadas com as imagens.

Como, a função “ImageFolder” já classifica as pastas das imagens por índice inteiro não é preciso transformar todos os nomes das pastas para tipo inteiro

Além de, a função “ImageFolder” pegar todas as imagens de uma certa pasta dada para a função e já classificar por índice, ela também possui a opção de já aplicar uma transformação enquanto pega as imagens da pasta, então podemos aplicar nossa transformação citada anteriormente e aplicar nessa função

Figura 2 - Extração do dataset

```
data_dir =
'/content/drive/MyDrive/IA/Data/Aeral_UC_Merced_Land/UCMerced_LandUse/Images'
dataset = datasets.ImageFolder(root=data_dir, transform=transform)
```

Assim, o dataset já estará todo transformado para tensor e normalizado, como resultado:

dataset[0] = [224, 224, indice]

## 3.3 Alteração de tipo de dados

Como o dataset extraído já transformou os nomes das pastas, normalizou e transformou o tamanho para 224x224, então faltou separar todo o dataset em duas variáveis, o data_train e o data_test para que possamos usá-los no nosso treinamento. Para isso iremos separar proporcionalmente para ambos de acordo com a porcentagem escolhida, sendo que data_train terá 80% do dataset e data_test terá 20% do dataset,  e também essa separação será igual para cada tipo de classe, por exemplo, cada classe possui 100 imagens das 21 classes então nosso data_train terá 80 imagens de cada classe e o data_test terá 20 imagens da cada classe. Dessa forma proporcional não há perigo de dividir o dataset aleatoriamente e ao executar nosso modelo não possui certas imagens não treinadas

Primeiramente, para separar proporcionalmente, fazemos um dicionário para cada rótulo e armazenamos em 80%, 20%, separadamente, para que data_train e data_test tenham a quantidade correta.

Figura 3 - Criação dos índices para data_train e data_test

```
train_indices = []
test_indices = []
for i, (image, label) in enumerate(dataset.imgs):
    label = convert_label(label)
    if train_counts[label] < 0.8 * 100:  # 80%
        train_indices.append(i)
        train_counts[label] += 1
    else:
        test_indices.append(i)          #20%
        test_counts[label] += 1
```

Enfim, é criado o data_train e o data_set respectivamente 80% 20%, que é dito pelo train_indices e o test_indices que calculamos acima, assim a função “Subset” consegue separar os dados do dataset baseado nos índices de ambos os parâmetros

Figura 4 - Atribuição do dataset e índice em data_train e data_test

```
data_train = Subset(dataset, train_indices)
data_test = Subset(dataset, test_indices)
```

Sendo essas todas as transformações feitas no dataset, poderíamos fazer mais alterações adicionando mais imagens, rotacionando elas ou adicionando ruído para deixar o nosso modelo melhor, mas não foi feito por agora.

# 4 Pós-Processamento

Neste projeto, para o processo de usar o nosso dataset utilizaremos o modelo VGG16 do PyTorch para fazer a nossa rede neural e testá-la, faremos com três diferentes formatos, sem o pré treinamento do VGG16 ,ou seja, fromScratch, depois usaremos o pré treinamento do VGG16 com apenas a última camada fully connected descongelada e, por fim, com o pré treinamento do VGG16 com apenas a última da convolucional e a última camada fully connected descongelada. Com objetivo de analisar todos os treinos aplicados acima, explorando os resultados e verificando quais dos treinos possuem a melhor precisão de treino.

Para executar todo os tipos de treinos baseado no VGG16 foi criado uma classe que atende todos os parâmetros que podem ser modificados para o treino.

Figura 5 - Código para o treinamento do modelo VGG16

```
import torch

class VGG16Trainer:
    def __init__(self, model, train_loader, test_loader, criterion, optimizer,
device=torch.device('cpu')):
        self.model = model
        self.train_loader = train_loader
        self.test_loader = test_loader
        self.criterion = criterion
        self.optimizer = optimizer
        self.device = device
        self.model.to(self.device)
```

```
    def train(self, num_epochs):
        loss_list = []
        for epoch in range(num_epochs):
            self.model.train()
            running_loss = 0.0

            for i, (inputs, labels) in enumerate(self.train_loader):
                inputs, labels = inputs.to(self.device), labels.to(self.device)
                self.optimizer.zero_grad()
                outputs = self.model(inputs)

                 loss = self.criterion(outputs, labels)
                loss.backward()
                self.optimizer.step()

                running_loss += loss.item() * inputs.size(0)

            epoch_loss = running_loss / len(self.train_loader.dataset)
            loss_list.append(epoch_loss)
            print(f'Epoch [{epoch+1}/{num_epochs}], Training Loss: {epoch_loss:.4f}')

        return loss_list
    def test(self):
        self.model.eval()
        correct = 0
        total = 0

        with torch.no_grad():
            for inputs, labels in self.test_loader:
                inputs, labels = inputs.to(self.device), labels.to(self.device)
                outputs = self.model(inputs)
                _, predicted = torch.max(outputs, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        test_accuracy = correct / total
        test_accuracy = test_accuracy * 100
        print(f'Test Accuracy: {test_accuracy:.2f}%')

        return test_accuracy

    def save_model(self, path):
        torch.save(self.model.state_dict(), path)
```

Com essa classe é possível executar o treinamento, checar a precisão baseada com algumas imagens do test e salvar o modelo em alguma pasta desejada. Para rodar diferentes treinos você deve instanciar o objeto, passando os parâmetros, para depois começar a utilizar os métodos train, test e save_model.

## 4.1 VGG16 sem pré treino

Como esperado, esse deve ser o treino mais lento e com a pior precisão, pois teremos que treinar com o nosso dataset passando por todas as camadas que o VGG16 possui, que no total são 36. Foi usado estes parâmetros abaixo.

Figura 6 - Parâmetros para modelo sem pré treino

```
vgg16 = models.vgg16(pretrained=False)

num_features = vgg16.classifier[6].in_features
vgg16.classifier[6] = nn.Linear(4096, 21)

train_loader = DataLoader(data_train, batch_size=32, shuffle=True)
test_loader = DataLoader(data_test, batch_size=32)

criterion = nn.CrossEntropyLoss()

optimizer = optim.SGD(vgg16.parameters(), lr=0.01)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
vgg16.to(device)

num_epochs = 50
```

**Precisão:** 43.10%

## 4.2 VGG16 com pré treino e última FC

Agora que estamos utilizando um modelo pré-treinado do VGG16, o tempo de espera é significativamente reduzido, além de que o nosso treino pode aumentar de precisão, mesmo com apenas usando uma Fully Connected. Isso ocorre porque os recursos já treinados pelo VGG16 em tarefas anteriores, então o que o nosso treinamento está fazendo é apenas adaptar o treino já feito e tentar gerar um treino novo que atenda com o nosso objetivo de classificação. Para este treino, foi usado os parâmetros abaixo

Figura 7 - Parâmetros para modelo com pré treino e última FC

```
vgg16 = models.vgg16(pretrained=True)

vgg16 = models.vgg16(pretrained=True)

for param in vgg16.parameters():
    param.requires_grad = False

vgg16.classifier[6].requires_grad = True  #last FC
num_features = vgg16.classifier[6].in_features
vgg16.classifier[6] = nn.Linear(4096, 21)

train_loader = DataLoader(data_train, batch_size=64, shuffle=True)
test_loader = DataLoader(data_test, batch_size=64)

criterion = nn.CrossEntropyLoss()

optimizer = optim.SGD(vgg16.parameters(), lr=0.01)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
vgg16.to(device)

num_epochs = 50
```

**Precisão:** 84.76%

## 4.3 VGG16 com pré treino e última Convolutional e FC

Por fim, vamos usar a VGG16 pré-treinada e modificar tanto a última camada convolucional quanto a última camada fully connected para serem as únicas descongeladas para o treino. Com o seguinte parametros

Figura 8 - Parâmetros para modelo com pré treino e com última Convolucional e FC

```
vgg16 = models.vgg16(pretrained=True)

for param in vgg16.parameters():
    param.requires_grad = False

for param in vgg16.features[-4:].parameters():
    param.requires_grad = True   #last convolutional

vgg16.classifier[6].requires_grad = True  #last FC
num_features = vgg16.classifier[6].in_features
vgg16.classifier[6] = nn.Linear(4096, 21)

train_loader = DataLoader(data_train, batch_size=32, shuffle=True)
test_loader = DataLoader(data_test, batch_size=32)

criterion = nn.CrossEntropyLoss()

optimizer = optim.SGD(vgg16.parameters(), lr=0.01)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
vgg16.to(device)

num_epochs = 50
```

**Precisão:** 88.33%

## 4.4 Análise de Resultados

Agora, com os resultados recebidos pelos três treinamentos realizados , podemos analisar a função de perda e a precisão. Avaliaremos o desempenho dos modelos treinados através de gráficos, para melhor visualização e entendimento, que por final podemos supor o que foi feito durante o treinamento. Por fim, escolhemos qual dos treinos é o melhor para esse nosso dataset.

Para a função de perda de cada treino foi gerado um gráfico de linhas que recebe todas as funções de perdas de cada época do treinamento.

E para a precisão dos três testes foi gerado um gráfico de barras simples, que mede a precisão deles.

Figura 9 - Gráfico de função de perda dos treinamentos

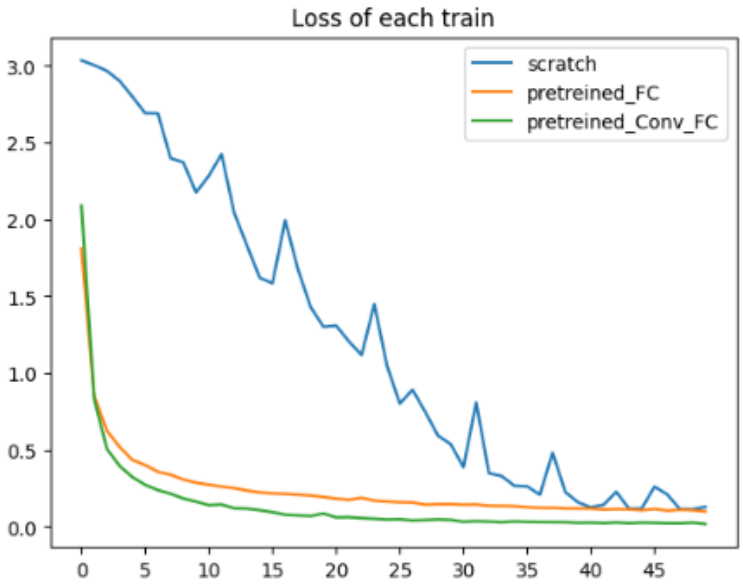
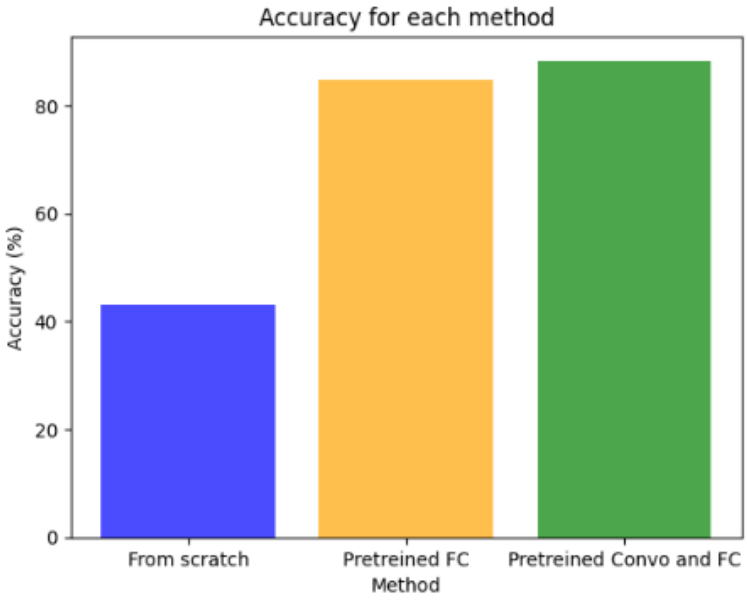



Figura 10 - Gráfico precisão dos treinamentos




Como observado no gráfico, a função de perda que mais se difere dos restando é do treino fromScratch(sem pré treino), ela é decrescente porém a função de perda dela é muito devagar e inconstante algumas vezes, tendo alguns valores maiores que os anteriores, que não difere muito do resultado, já que ele foi o com a pior precisão dos treinamentos. Algo que poderia ajudar a aumentar a precisão é aumentar a quantidade de épocas, já que ele está entendendo cada vez mais para um valor baixo ou mudar a taxa de aprendizado dele, que foi de 0.01.

Já os outros dois gráficos de função de perda são bem similares entre si, ambos começam com  um valor de função de perda alto que desce rapidamente nas primeiras épocas. Esse começo alto que acontece na função de perda de ambos está relacionado com o primeira passagem do dataset pelo treino, pois inicialmente não possuem nenhum peso já calculado para o nosso dataset e assim na primeira passagem de volta, quando está arrumando os pesos gera uma diferença muito grande entre a primeira e segunda época, que vai diminuindo com a passagem das épocas. E entre os dois últimos treinamentos o com a última Convolucional e FC possui melhor resultado, tanto na função de perda e na precisão do treino.

Algo a se atentar é que o treino não foi feito com diversos parâmetros diferentes, apenas os mostrados acima, então ainda é possível gerar melhores resultados com esses treinos mudando os parâmetros iniciais. Principalmente os dois últimos treinos, já que pela função de loss ter uma pouca variação da metade pro final das épocas alto risco de ocorrer Overfitting com esses treinos, uma tática boa é diminuir a quantidade de épocas para quando eles começam a variar pouco na função de perda e ver o resultado da precisão.

# 5 Conclusão

Ao finalizar este estudo sobre redes neurais utilizando o conjunto de dados UC Merced, fizemos alterações no nosso dataset até ser utilizada pelo modelo VGG16 proposto e fizemos testes com três tipos de parâmetros iniciais diferentes, todo esse processo foi feito com o intuito de obter o melhor treino possível para esse dado dataset e com o certo objetivo de classificação das imagens.

E por fim, dos treinos que fizemos com o nosso dataset, aquele que mais obteve resultados foi com, pré treinado com a última camada Convolucional e FC, além de possuir a melhor função de perda obteve a maior precisão, comparada com os demais, de 88.33%. Com seus parâmetros citados já anteriormente em **[4.3]**.

Ao investigar os resultados desses treinamentos, pudemos compreender um pouco melhor do que houve na fase de treinamento e de ajuste de cada treino, observando e analisando os gráficos gerados pela função de perda e precisão deles. Essas análises fornecem resultados que podem ser melhorados modificando o treino atual ou os parâmetros iniciais de cada treino, assim podendo desenvolver abordagens mais eficazes na construção de sistemas de IA para reconhecimento de imagens.

Links:
DataSet utilizada:
[1]: http://weegee.vision.ucmerced.edu/datasets/landuse.html

Código via:
RennamFariaUNIFESP/Aeral_Scenes_Classification (github.com)
ou
🗀 Aeral scenes work