Um pequeno estudo sobre data augmentation e transfer learning no contexto de redes neurais profundas

Carlos Eduardo Gonçalves de Oliveira

Programa de Pós-Graduação em Engenharia Elétrica e da Computação, Universidade Federal de Goiás

E-mail: carlosedgonc@gmail.com

Dezembro, 2023

Abstract. Este artigo tem como objetivo atender aos requisitos do primeiro projeto da disciplina de Redes Neurais Profundas e também de matar a minha própria curiosidade. Aqui, decidi implementar *from scratch* uma versão simplificada da ResNet e um segundo modelo misturando a ResNet com blocos *inception*. Para averiguar o impacto da técnica de data augmentation na performance desses modelos, eles foram treinados na ausência e na presença da técnica. Também foi feito o *finetuning* de um terceiro modelo: a ResNet-18 pré-treinada no *dataset* ImageNet. Os dados usados neste estudo são imagens referentes a uma dentre 100 categorias de esportes.

Como resultados, obteve-se que os blocos *inception* não acrescentaram à performance da ResNet: ao contrário, houve uma piora na performance. Isso obviamente não diz nada a respeito dos blocos *inception* em si, mas tão somente da aplicabilidade dos mesmos no dataset deste estudo. Ademais, o impacto da técnica de data augmentation foi evidente: houve uma melhora significativa na performance tanto do modelo 1 quanto do modelo 2. Com relação ao finetuning da ResNet-18, ela foi o melhor modelo e atingiu cerca de 97% de acurácia total nos dados de teste. A performance geral dos outros modelos sobre os dados de teste foi: modelo 1 sem data augmentation - acurácia total = 80.80%; modelo 1 com data augmentation - acurácia total = 87.40%; modelo 2 sem data augmentation - acurácia total = 76.60%; modelo 2 com data augmentation - acurácia total = 84.60%.

Todo o código em Python utilizado para conduzir o estudo está disponível no próprio artigo. O código, aliás, contém uma função interessante que criei para uma melhor visualização da performance de modelos de classificação no contexto de problemas de múltiplas classes.

Na seção de Materiais suplementares há o cumprimento de um dos requisitos do trabalho, que é de mostrar exemplos de imagens com as predições de cada modelo. O código utilizado para gerar as figuras está na mesma seção.

1. Introdução

1.1. Contextualização

O foco em classificação de imagens sempre tem sido a extração de *features*, que é realizada manualmente no caso dos algoritmos de classificação tradicionais. Essa abordagem, entretanto, é limitada em termos de portabilidade e generalização por depender exageradamente da criatividade e da seletividade do pesquisador responsável pela modelagem. Assim, deixar com que o próprio computador extraia as principais *features* de classificação, analogamente à forma como ocorre o fenômeno da visão biológica, foi o que pesquisadores sonharam e conseguiram realizar.

As redes neurais artificiais (*artificial neural networks*, ANN) são uma rede neural abstrata, um modelo de operações matemáticas compostas de um grande número de neurônios interconectados. O modelo simula grosseiramente o processamento neural em rede dos sinais neuronais biológicos e, portanto, pode ter inúmeras camadas, o que deu origem ao nome *deep learning* (aprendizado profundo, de muitas camadas). No contexto de imagens, os pesquisadores tentaram utilizar o modelo de *multilayer perceptron* (MLP), mas notaram que a adição de operações convolucionais (comumente aplicadas como técnica de processamento de imagens) poderia maximizar o processo de extração de *features* em imagens. Adicionado ao fato de que as operações convolucionais poderiam ser otimizadas se realizadas na GPU em vez da CPU, houve uma intensificação nas pesquisas nessa área que representou o início da era das *convolutional neural networks* (CNN). Começando com a LeNet-5 (1989 - 1998), arquiteturas novas foram surgindo, como a AlexNet (2012), ZFNet (2013), VGGNet (2014), GoogLeNet (2014) e a ResNet (2015), cada uma com quantidades de camadas e blocos constituintes diferentes. Vale ressaltar que o grande catalisador do surgimento de novas arquiteturas foram as competições envolvendo datasets como o ImageNet, que contém cerca de 1 milhão de imagens com 1000 classes diferentes.

Como ilustração de algumas operações importantes envolvendo a forma como as ANNs funcionam e como evoluíram para as CNNs, tem-se as imagens que se seguem. Na figura 1 é mostrado o funcionamento de um neurônio em ANN; na figura 2 tem-se a estrutura de uma MLP; na figura 3 é ilustrada a operação de convolução; por último, na figura 4 é mostrada a arquitetura de uma CNN (mais especificamente a LeNet-5).

Para mais detalhes quanto ao que foi brevemente explicado neste artigo, recomenda-se a leitura da referência [1]. Nela é feita uma revisão bastante completa de CNNs, o que não é o propósito do meu artigo.

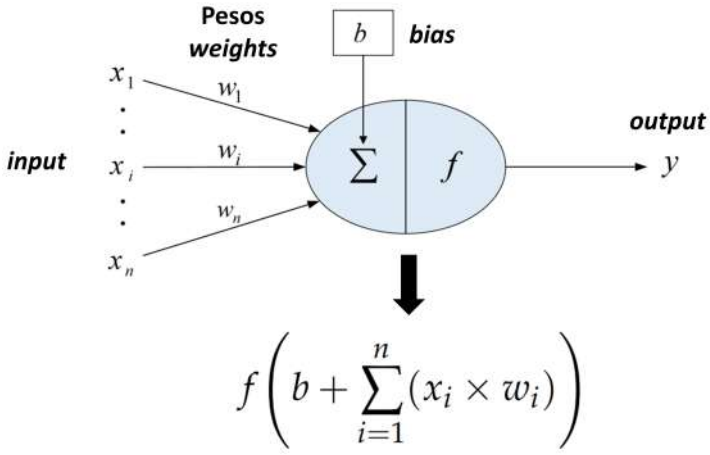

Figure 1. Análogo a um neurônio biológico, o neurônio em ANN recebe um *input*, aplica uma transformação matemática e em seguida retorna um *output*. Para cada componente i do input vetorial x, existe um peso respectivo w do neurônio. Eles são multiplicados componente a componente e depois somados junto a um viés (*bias*), tal como ocorreria num modelo linear simples. Para que haja não-linearidade, o resultado dessa operação é passado como argumento de uma função não-linear f, denominada função de ativação (*activation function*). Figura adaptada de [1].

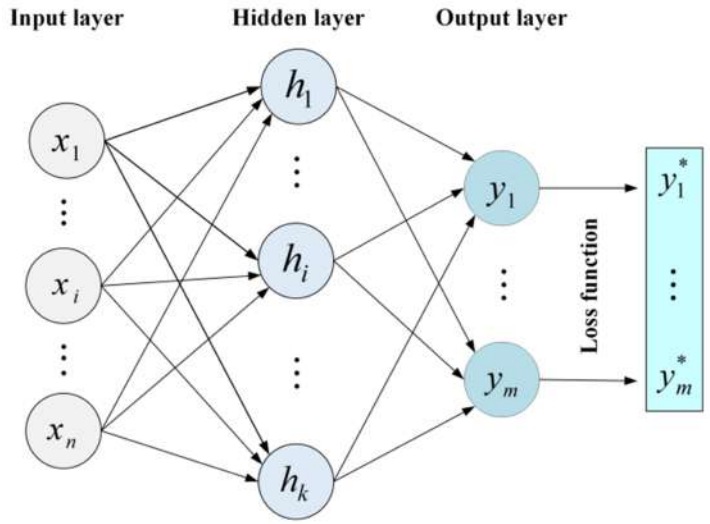

Figure 2. Múltiplos neurônios em ANN podem ser organizados na forma de camadas, dando origem ao MLP. Devido às funções de ativação, quanto mais camadas do tipo *hidden layer*, mais não-linear é a fronteira de decisão do modelo de classificação. No entanto, é preciso precaução ao aumentar demasiadamente a quantidade de camadas devido ao alto risco de *overfitting*. Note que existe a *loss function*, que é a função que mede a similaridade entre os *outputs* y_i de saída do modelo e os *outputs* y_i^* de referência. Figura adaptada de [1].

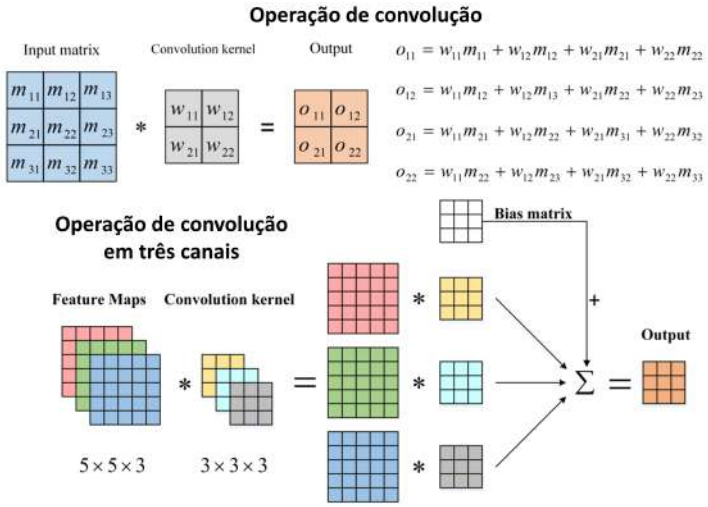

Figure 3. A operação de convolução em matrizes é análoga ao produto escalar em vetores. Cada campo receptivo (ou "pedaço") da imagem é tratado como sendo da mesma dimensão do *kernel* (ou filtro). Nesse sentido, dada uma *kernel*, é possível "medir a semelhança" de diferentes "pedaços" da imagem com o *kernel* em questão, podendo-se realçar bordas ou padrões mais complexos. Em imagens de três canais, o *kernel* também possuirá três canais, e o *output* final será o somatório do resultado de todos os canais. No contexto de redes neurais, há a possibilidade de se adicionar ao resultado final uma matriz de vieses (*bias matrix*), mas isso é raramente realizado na prática. Figura adaptada de [1].

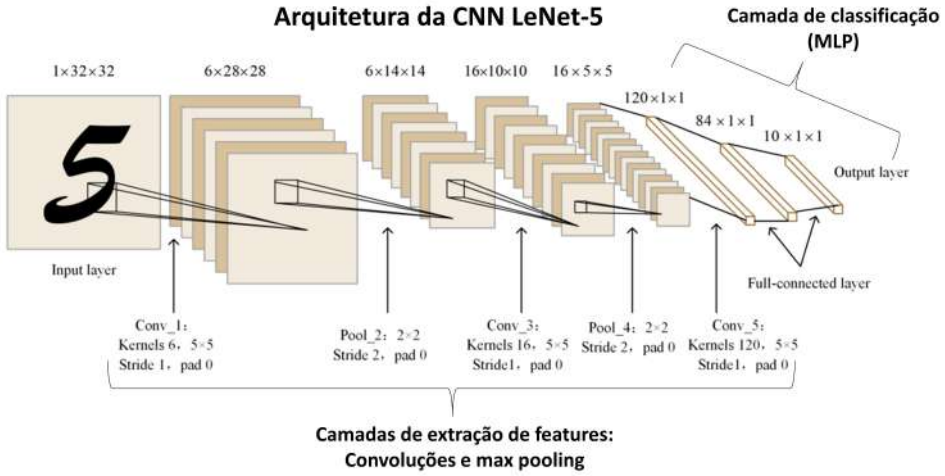

Figure 4. A operação de convolução é comumente aplicada na forma de camadas intermediárias dedicadas à extração de *features*. Associada à operação de convolução, existe a operação de redução de dimensionalidade, frequentemente utilizada na forma de *max pooling*, onde o maior elemento da matriz do campo receptivo é selecionado. Na "cabeça" das CNNs, fica a camada final de classificação, onde os *feature maps* são reduzidos a vetores e em seguida processados por uma MLP, responsável pelo uso dos features extraídos para gerar um *output* de classificação. Figura adaptada de [1].

1.2. Caracterização do estudo

Com o propósito de atender aos requisitos do primeiro trabalho da disciplina de Redes Neurais Profundas e de matar a minha própria curiosidade, decidi implementar *from scratch* dois modelos de redes neurais convolucionais: (1) um inspirado na ResNet (isto é, com blocos residuais, que explicarei adiante) e (2) outro misturando blocos residuais e blocos *inception* do GoogLeNet (que também explicarei adiante). Apenas utilizar os dados *as it is* seria meio sem graça, então decidi comparar os resultados desses modelos em dois contextos diferentes: (1) sem data augmentation e (2) com data augmetation. Por último, como foi do meu desejo também estudar a implementação de *transfer learning* com *finetuning*, vou mostrar qual a performance da ResNet-18 treinada previamente no *dataset* do ImageNet no *dataset* do trabalho.

1.3. Objetivos específicos do estudo

Os objetivos específicos deste artigo são:

- Comparar a performance de dois modelos de redes neurais profundas (um contendo blocos residuais e outro contendo blocos residuais e blocos inception);
- Comparar a performance dos dois modelos anteriores com o modelo ResNet-18 do PyTorch treinado via *transfer learning* e *finetuning*;

Para que seja possível atender aos objetivos citados acima, serão utilizadas diferentes métricas de performance e técnicas de visualização. Ademais, compreendo que o trabalho da disciplina exige que eu faça o gráfico do histórico de aprendizado dos modelos contendo tanto a curva de validação quanto a de teste. Optei, entretanto, por seguir a metodologia comumente aplicada nos estudos mais atuais envolvendo *deep learning*, que é de adicionar apenas a curva de validação e, por último, analisar a performance dos modelos sobre os dados de teste.

2. Metodologia

2.1. Dataset com 100 categorias de esportes

O *dataset* (link do Kaggle) utilizado neste estudo contém 13493 imagens de treino, 500 de validação e 500 de teste. Todas elas têm formato de 224x224x3 e representam um esporte dentre 100 categorias no total. A distribuição de categorias nos dados de treino é praticamente homogênea, com uma média de cerca de 130 imagens por categoria. Já nos dados de validação e de teste existem 5 imagens por categoria.

Na descrição do *dataset* no Kaggle é dito que o *dataset* está limpo e que quem for usá-lo para estudos de redes neurais convolucionais podem chegar a uma acurácia total nos dados de teste de 95%, podendo passar os 98% caso façam uso de *transfer learning*. Analisando alguns dos notebooks que utilizaram esse dataset, entretanto, é possível notar que na verdade, sem *transfer learning*, os melhores modelos atingem uma acurácia total entre 70% e 85%, o que me faz questionar se a expectativa do dono do *dataset* não foi muito otimista. Já com *transfer learning* a expectiva se realiza em vários notebooks.

Na figura 5 é possível ver alguns exemplos de imagens contidas no dataset.

Figure 5. Exemplos de imagens contidas no *dataset* de 100 categorias de esportes. Como é possível perceber, a diferença de cores e de formas entre as categorias mostradas é significativa e serve como indicativo de que uma CNN pode ter uma boa performance nesse *dataset*.

Para que as imagens possam ser carregadas na forma de tensores (que é o formato aceito para processamento dentro do PyTorch), é aconselhável criar uma classe da forma *Dataset* (considere fazer a leitura da documentação aqui).

Logo abaixo, mostro como escrevi o código referente a essa classe. Tive que usar módulos específicos para abrir os arquivos de imagem e também considerar a possibilidade de aplicação de *transformers* (já que preciso converter os dados em tensores, por exemplo, e também aplicar *data augmentation*)

```
1
2  import os
3  from torch.utils.data import Dataset
4  from torch import zeros
5  import cv2
6
7  class SportsDataset(Dataset):
8      def __init__(self, parent_path, transforms=None):
9          self.parent_path = parent_path
10         self.transforms = transforms
11
12         self.image_paths = []
13         self.labels = []
14         for class_folder in os.listdir(parent_path):
15             for image_file in os.listdir(parent_path + "/" + class_folder):
16                 if "lnk" not in image_file:
17                     self.image_paths.append(parent_path + "/" + class_folder + "/" + image_file)
18                     self.labels.append(class_folder)
19
20         self.unique_classes = list(set(self.labels))
21         self.unique_classes.sort()
22
23         self.class_dict = dict(enumerate(self.unique_classes))
24
25     def __len__(self):
26         return len(self.image_paths)
27
28     def __getitem__(self, i):
29         image_path = self.image_paths[i]
30         label = self.labels[i]
31
32         image = cv2.imread(image_path)
33         image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
34
35         if self.transforms is not None:
36             image = self.transforms(image)
37
38         y = zeros(len(self.unique_classes))
```

```
39          y[self.unique_classes.index(label)] = 1
40
41          return image, y
```

2.2. Arquitetura do modelo 1: ResNet simplificada

O principal bloco constituinte de uma ResNet é, sem dúvidas, o bloco residual. Ele surgiu como uma solução do fato experimental do decréscimo na performance das redes neurais convolucionais quando há um excesso na quantidade de camadas e foi denominado "degradação" (*degradation*) da rede. É verdade que o problema poderia ser a explosão ou a desaparição dos gradientes devido a múltiplas regras da cadeia, mas foi efetivamente constatado que o problema de degradação dos gradientes é simplesmente a impossibilidade de uma boa otimização da rede. Os blocos residuais resolvem esse problema a partir do princípio da opcionalidade (vide figura 6). Se, no bloco conseguinte, adicionarmos o *input* do bloco "de agora", então o bloco "de agora", caso ele não seja necessário do ponto de vista do processo de otimização, terá seus pesos convergindo para zero e será tratado como se não existisse. Assim, múltiplos blocos residuais podem ser adicionados, sem o efeito de degradação. Para mais detalhes, considere ler o artigo original [2].

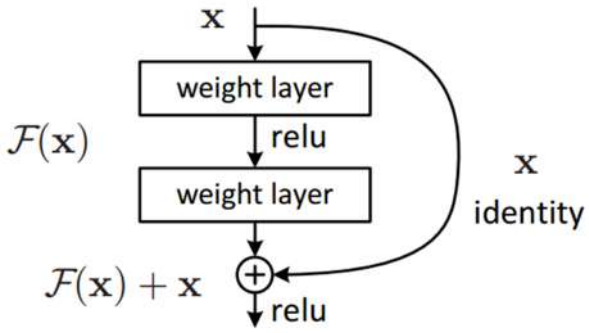

Figure 6. Ilustração do bloco residual. O *input* do bloco residual é "passado para frente", junto com o resultado do processamento do *input* pelo bloco. Caso durante o processo de otimização o bloco se demonstre desnecessário, o efeito de degradação da rede será evitado através da convergência de $F(x)$ para zero, como se o bloco não existisse. Figura extraída de [2]

Na arquitetura construída por mim através do PyTorch, fiz uso de quatro blocos convolucionais no total. Adicionei também blocos convolucionais do tipo comum (tal como no VGGNet), com camadas de convolução seguidas de *max pooling*. Ademais, fiz uso de *batch normalization* a cada convolução e de *dropout* (de 30%) na camada densa. A função de ativação que escolhi foi a *ReLU* para evitar problemas com gradiente evanescente. Adicionei comentários em inglês mostrando as dimensões do *input* e do *output* de cada bloco. É esperado como *input* uma imagem de dimensões 128x128x3. Optei por esse número em específico porque minha intenção foi facilitar as contas das dimensões dos *inputs* e *outputs* a cada bloco. Foi por isso também que todas as convoluções que apliquei apresentam padding = 1 (pois as dimensões do *output* ficam iguais as do *input*).

O código tal como escrevi está mostrado logo abaixo:

```
import torch.nn as nn

# standard convolutional block
def conv_max_pool_block(in_chan, out_chan):
    return nn.Sequential(
            nn.Conv2d(in_chan, out_chan, (3, 3), padding=1),
            nn.BatchNorm2d(out_chan),
            nn.ReLU(),
            nn.MaxPool2d(2))

# residual block
def res_block(in_chan):
    return nn.Sequential(
            nn.Conv2d(in_chan, in_chan, (3, 3), padding=1),
            nn.BatchNorm2d(in_chan),
            nn.ReLU(),
            nn.Conv2d(in_chan, in_chan, (3, 3), padding=1),
            nn.BatchNorm2d(in_chan),
            nn.ReLU())

# ResNet class
class ResNet(nn.Module):
    def __init__(self):
        super().__init__()

        # 1: conv + max pool layer
        # in: 128x128x3
        self.layer1 = conv_max_pool_block(3, 32) # out: 64x64x32

        # 2: res layer
        # in: 64x64x32
        self.layer2 = res_block(32) # out: 32x32x64

        # 3: conv + max pool layer
        # in: 64x64x32
        self.layer3 = conv_max_pool_block(32, 64) # out: 32x32x64

        # 4: res layer
        # in: 32x32x64
        self.layer4 = res_block(64) # out: 32x32x64

        # 5: conv + max pool layer
```

```
        # in: 32x32x64
        self.layer5 = conv_max_pool_block(64, 128) # out: 16x16x128

        # 6: res layer
        # in: 16x16x128
        self.layer6 = res_block(128) # out: 16x16x128

        # 7: conv + max pool layer
        # in: 16x16x128
        self.layer7 = conv_max_pool_block(128, 256) # out: 8x8x256

        # 8: res layer
        # in: 8x8x256
        self.layer8 = res_block(256) # out: 8x8x256

        # 9: conv + max pool layer
        # in: 8x8x256
        self.layer9 = conv_max_pool_block(256, 512) # out: 4x4x512

        # classifier layer
        # in: 4x4x512
        self.classifier = nn.Sequential(
            nn.MaxPool2d(4),
            nn.Flatten(),
            nn.Dropout(.3),
            nn.Linear(512, 100)
        ) # out: 100

    def forward(self, x):

        # 1: conv + max pool layer
        y = self.layer1(x)

        # 2: res layer
        y = self.layer2(y) + y

        # 3: conv + max pool layer
        y = self.layer3(y)

        # 4: res layer
        y = self.layer4(y) + y

        # 5: conv + max pool layer
```

```
86          y = self.layer5(y)
87
88          # 6: res layer
89          y = self.layer6(y) + y
90
91          # 7: conv + max pool layer
92          y = self.layer7(y)
93
94          # 8: res layer
95          y = self.layer8(y) + y
96
97          # 9: conv + max pool layer
98          y = self.layer9(y)
99
100         # classifier layer
101         y = self.classifier(y)
102
103         return y
```

2.3. Arquitetura do modelo 2: ResNet simplificada com blocos inception

No contexto de redes neurais convolucionais, é sempre difícil decidir entre um tamanho de *kernel* e outro. Assim, com a nova arquitetura GoogLeNet, o bloco *inception*, partindo também do princípio da opcionalidade, surgiu para deixar com que o processo de otimização decida qual tamanho de *kernel* ou operação convolucional é a melhor (com a opção de utilizar todos os tamanhos), sem intervenção subjetiva. Isso só é possível se utilizarmos de operações paralelas numa mesma camada, conforme é mostrado na figura 7. A versão do bloco inception da figura corresponde à primeira de todas. As versões posteriores consideram diversos outros fatores relacionados a melhorias em termos de performance e tempo de processamento. Para mais detalhes, considere a leitura da referência [1].

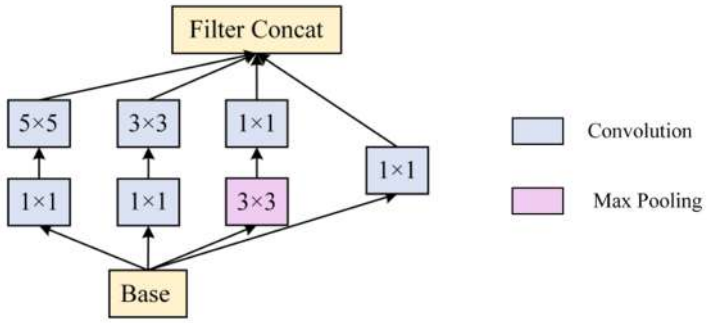

Figure 7. Ilustração da primeira versão do bloco *inception*. Esse bloco considera quatro diferentes operações convolucionais, de modo a maximizar a extração de informações úteis a partir do *input*. Note que, ao fim de tods as operações, os resultados são concatenados como *feature maps* distintos. Figura extraída de [1]

Na tentativa de fazer uso das vantagens tanto dos blocos residuais quanto dos blocos *inception*, combinei-os usando o PyTorch para posteriormente analisar se há melhora na performance sobre o *dataset* de estudo. As considerações acerca da arquitetura conforme construí são basicamente as mesmas que foram feitas no modelo 1. Ou seja, fiz uso de *batch normalization* e de *dropout* na camada densa.

O código escrito por mim está mostrado logo abaixo:

```
1  import torch
2  import torch.nn as nn
3
4  # standard convolutional block
5  def conv_max_pool_block(in_chan, out_chan):
6      return nn.Sequential(
7              nn.Conv2d(in_chan, out_chan, (3, 3), padding=1),
8              nn.BatchNorm2d(out_chan),
9              nn.ReLU(),
```

```
            nn.MaxPool2d(2))

# residual block
def res_block(in_chan):
    return nn.Sequential(
            nn.Conv2d(in_chan, in_chan, (3, 3), padding=1),
            nn.BatchNorm2d(in_chan),
            nn.ReLU(),
            nn.Conv2d(in_chan, in_chan, (3, 3), padding=1),
            nn.BatchNorm2d(in_chan),
            nn.ReLU())

# inception block, with parallel branches
class inception_block(nn.Module):
    def __init__(self, in_chan, out_chan):
        super().__init__()

        # conv 1x1 -> conv 5x5
        self.branch1 = nn.Sequential(nn.Conv2d(in_chan, (out_chan//4)//2, kernel_size=1),
                                     nn.BatchNorm2d((out_chan//4)//2),
                                     nn.ReLU(),
                                     nn.Conv2d((out_chan//4)//2, out_chan//4, kernel_size=5, padding=2),
                                     nn.BatchNorm2d(out_chan//4),
                                     nn.ReLU())

        # conv 1x1 -> conv 3x3
        self.branch2 = nn.Sequential(nn.Conv2d(in_chan, (out_chan//4)//2, kernel_size=1),
                                     nn.BatchNorm2d((out_chan//4)//2),
                                     nn.ReLU(),
                                     nn.Conv2d((out_chan//4)//2, out_chan//4, kernel_size=3, padding=1),
                                     nn.BatchNorm2d(out_chan//4),
                                     nn.ReLU())

        # max pool 3x3 -> conv 1x1
        self.branch3 = nn.Sequential(nn.MaxPool2d(kernel_size=3, padding=1, stride=1),
                                     nn.Conv2d(in_chan, out_chan//4, kernel_size=1),
                                     nn.BatchNorm2d(out_chan//4),
                                     nn.ReLU())

        # conv 1x1
        self.branch4 = nn.Sequential(nn.Conv2d(in_chan, out_chan//4, kernel_size=1),
                                     nn.BatchNorm2d(out_chan//4),
                                     nn.ReLU())
```

```
53
54      def forward(self, x):
55          return torch.cat([self.branch1(x),
56                            self.branch2(x),
57                            self.branch3(x),
58                            self.branch4(x)], dim=1)
59
60  class InceptionResNet(nn.Module):
61      def __init__(self):
62          super().__init__()
63
64          # 1: conv + max pool layer
65          # in: 128x128x3
66          self.layer1 = conv_max_pool_block(3, 32) # out: 64x64x32
67
68          # 2: res layer
69          # in: 64x64x32
70          self.layer2 = res_block(32) # out: 64x64x32
71
72          # 3: inception + max pool layer
73          # in: 64x64x32
74          self.layer3 = inception_block(32, 64) # out: 64x64x64
75          self.max_pool3 = nn.MaxPool2d(2) # out: 32x32x64
76
77          # 4: res layer
78          # in: 32x32x64
79          self.layer4 = res_block(64) # out: 32x32x64
80
81          # 5: inception + max pool layer
82          # in: 32x32x64
83          self.layer5 = inception_block(64, 128) # out: 32x32x128
84          self.max_pool5 = nn.MaxPool2d(2) # out: 16x16x128
85
86          # 6: res layer
87          # in: 16x16x128
88          self.layer6 = res_block(128) # out: 16x16x128
89
90          # 7: inception + max pool layer
91          # in: 16x16x128
92          self.layer7 = inception_block(128, 256) # out: 16x16x256
93          self.max_pool7 = nn.MaxPool2d(2) # out: 8x8x256
94
95          # 8: res layer
```

```
96          # in: 8x8x256
97          self.layer8 = res_block(256) # out: 8x8x256
98
99          # 9: inception layer
100         # in: 8x8x256
101         self.layer9 = inception_block(256, 512) # out: 8x8x512
102
103         # 10: conv + max pool layer
104         # in: 8x8x512
105         self.layer10 = conv_max_pool_block(512, 512) # out: 4x4x512
106
107         # classifier layer
108         # in: 4x4x512
109         self.classifier = nn.Sequential(
110             nn.MaxPool2d(4),
111             nn.Flatten(),
112             nn.Dropout(.3),
113             nn.Linear(512, 100)
114         ) # out: 100
115
116     def forward(self, x):
117
118         # 1: conv + max pool layer
119         y = self.layer1(x)
120
121         # 2: res layer
122         y = self.layer2(y) + y
123
124         # 3: inception layer + max pool layer
125         y = self.layer3(y)
126         y = self.max_pool3(y)
127
128         # 4: res layer
129         y = self.layer4(y) + y
130
131         # 5: inception layer + max pool layer
132         y = self.layer5(y)
133         y = self.max_pool5(y)
134
135         # 6: res layer
136         y = self.layer6(y) + y
137
138         # 7: inception layer + max pool layer
```

```
139         y = self.layer7(y)
140         y = self.max_pool7(y)
141 
142         # 8: res layer
143         y = self.layer8(y) + y
144 
145         # 9: inception layer
146         y = self.layer9(y)
147 
148         # 10: conv + max pool block layer
149         y = self.layer10(y)
150 
151         # classifier layer
152         y = self.classifier(y)
153 
154         return y
```

2.4. Um terceiro modelo: finetuning da ResNet-18 (transfer learning)

O modelo ResNet-18 pode ser facilmente carregado através do submódulo "*models*" do PyTorch. O ResNet-18 é aproximadamente o mesmo modelo (comparando-se com os pesos originais) que foi treinado para o artigo da referência [2]. Ele foi treinado sobre os dados do ImageNet, que contém cerca de 1 milhão de imagens e mil categorias diferentes. Neste link há todas as informações pertinentes sobre a ResNet-18, incluindo as métricas de performance e as transformações que devem ser aplicadas sobre os novos dados para *finetuning*. Não segui as recomendações à risca, e também não foi necessário, como ficará evidente na seção de resultados. Entretanto, continua sendo obrigatório que as dimensões das imagens de input sejam 224x224x3.

O código escrito para o download do modelo pré-treinado e para a troca da camada densa está mostrado logo abaixo. A mudança de estrutura na camada densa é necessária porque o modelo original tem como saída 1000 classes diferentes enquanto nosso problema neste estudo contém 100 classes.

```
1 import torch.nn as nn
2 from torchvision import models
3
4 model = models.resnet18(weights="IMAGENET1K_V1")
5 num_features = model.fc.in_features
6 model.fc = nn.Sequential(nn.Linear(num_features, num_features),
7                           nn.Linear(num_features, 100))
```

2.5. Pipeline de treinamento e de data augmentation

Para treinar um modelo de redes neurais convolucionais em PyTorch, é necessário carregar a classe *Dataset* num *DataLoader* com um *batch_size* específico. No meu caso, escolhi um *batch_size* de 32 para que não houvesse muita intervenção de ruído na atualização dos pesos. Não pude escolher um maior devido a limitações computacionais (fiz toda a programação pelo Kaggle e não posso estourar os limites impostos pela plataforma).

A quantidade total de épocas *num_epochs* foi variável, pois fiquei monitorando se havia diminuição considerável na *loss function* ou aumento significativo na acurácia total. Com um *learning rate* inicial de 10^{-4}, assim que observei uma estabilização na curva da *loss function* ou da acurácia total, alterei o learning rate para o valor de 10^{-5}. Minha intuição por trás dessa abordagem é que reajustes menores nos valores dos pesos nesse instante específico do aprendizado do modelo poderiam levar a uma melhora na performance.

O otimizador que escolhi foi o *Adam*, por ele levar em consideração a matriz Hessiana (de segundas derivadas), por fazer reajustes individuais nos pesos e por manter uma "memória" das iterações anteriores. Para mais informações sobre esse otimizador, clique aqui.

Já a loss function utilizada para otimização foi a "CrossEntropyLoss", por ser aplicável a problemas de múltiplas classes. Para mais informações sobre essa loss function, por favor, clique aqui.

Em relação às transformações que serão aplicadas sobre as imagens, as principais são a de *data augmentation* e a que transforma as imagens em tensores. No caso:

- Se o treinamento for ocorrer sem *data augmentation*, use somente o primeiro conjunto de transformações;
- Se o treinamento for ocorrer com data augmentation, use somente o segundo conjunto de transformações;
- Se o treinamento corresponder ao finetuning da ResNet-18, use o terceiro conjunto de transformações;

Note que a operação de *data augmentation* foi implementada usando-se o método de "*AutoAugment*". A política de *data augmentation* selecionada foi a default, visto que são aplicadas múltiplas transformações aleatórias pertinentes, incluindo rotações e mudanças nas cores, por exemplo. Para mais informações a respeito das transformações, por favor, consulte a documentação referente ao método "*AutoAugment*" (clicando aqui) e os exemplos de transformações (clicando aqui).

O código escrito aplicando todas as configurações citadas acima é mostrado logo abaixo.

```
1 import torch
2 from torch.utils.data import DataLoader
```

```
from torchvision import transforms, models
import torch.nn as nn
from tqdm import tqdm
from torch.optim import Adam
import joblib as jb # for saving the history dict
import os

# configs
device = "cuda" if torch.cuda.is_available() else "cpu"
batch_size = 32
num_epochs = 30 # 30 epochs in total by default. It can be different
e_0 = 1 # initial epoch

######## creating/loading model

# if you want to load the simplified version of resnet...
# model = ResNet().to(device)

# if you want to load my crazy mixture of residual and inception blocks...
# model = InceptionResNet().to(device)

# or if you want to apply finetuning over the resnet-18...
# model = models.resnet18(weights="IMAGENET1K_V1")
# num_features = model.fc.in_features
# model.fc = nn.Sequential(nn.Linear(num_features, num_features),
#                          nn.Linear(num_features, 100))
# model.to(device)

######## creating/loading history
history = {"train_loss": [],
           "val_loss": [],
           "train_acc": [],
           "val_acc": []}

# loss function
loss = nn.CrossEntropyLoss()

# optimizer
optimizer = Adam(model.parameters(), lr = .0001)

####### first set of transformations
# transforms_train = transforms.Compose([transforms.ToPILImage(),
#                                        transforms.Resize(128),
```

```
#                                   transforms.ToTensor()])
# transforms_test = transforms.Compose([transforms.ToPILImage(),
#                                   transforms.Resize(128),
#                                   transforms.ToTensor()])

####### second set of transformations
# transforms_train = transforms.Compose([transforms.ToPILImage(),
#                                   transforms.Resize(128),
#                                   transforms.AutoAugment(),
#                                   transforms.ToTensor()])
# transforms_test = transforms.Compose([transforms.ToPILImage(),
#                                   transforms.Resize(128),
#                                   transforms.ToTensor()])

####### third set of transformations
# transforms_train = transforms.Compose([transforms.ToPILImage(),
#                                   transforms.AutoAugment(),
#                                   transforms.ToTensor()])
# transforms_test = transforms.Compose([transforms.ToPILImage(),
#                                   transforms.ToTensor()])

# parent_path for SportsDataset class
train_path = "/kaggle/input/projeto1/Sports/train"
val_path = "/kaggle/input/projeto1/Sports/valid"
test_path = "/kaggle/input/projeto1/Sports/test"

####### initializing datasets
train_DS = SportsDataset(train_path, transforms_train)
val_DS = SportsDataset(val_path, transforms_test)
test_DS = SportsDataset(test_path, transforms_test)

####### initializing data loaders
train_loader = DataLoader(train_DS, batch_size=batch_size, shuffle=True,
                          pin_memory=True if device=="cuda" else False,
                          num_workers=os.cpu_count())

val_loader = DataLoader(val_DS, batch_size=batch_size,
                          pin_memory=True if device=="cuda" else False,
                          num_workers=os.cpu_count())

test_loader = DataLoader(test_DS, batch_size=batch_size,
                          pin_memory=True if device=="cuda" else False,
                          num_workers=os.cpu_count())
```

89

Para efetivamente treinar o modelo de classificação, rode o código abaixo. Atente-se ao fato de que é possível alterar o *learning rate* para 10^{-5} descomentando o código do início e alterando os valores de *e_0* e *num_epochs*. Obviamente o novo valor de *e_0* deve ser a última época na qual foi aplicada a *learning rate* de 10^{-4}, enquanto que *num_epochs* deve ser *e_0* mais a quantidade de épocas que você quer rodar adicionalmente. Ademais, note que, a cada época de treino, são armazenados num dicionário os resultados referentes ao valor médio da loss function nos dados de treino e de validação assim como a acurácia total. Esse dicionário é salvo posteriormente para que sejam construídos os gráficos do histórico de aprendizado dos modelos.

```
###### when the learning curves stabilize, uncomment the following code and
# run all the cell again
# e_0 = initial epoch after stabilization of learning curves
# num_epochs = new num_epochs
# optimizer = Adam(model.parameters(), lr = 1e-5)

for e in tqdm(range(e_0, num_epochs + 1)):

    # initialize total training and validation loss
    train_loss = 0
    val_loss = 0

    # initializing training and validation accuracy
    train_acc = 0
    val_acc = 0

    # set the model in training mode
    model.train()

    train_steps = 0
    for X, y in train_loader:

        X, y = X.to(device), y.to(device)

        # perform a forward pass and calculate the training loss
        pred = model.forward(X)
        loss_value = loss(pred, y)

        # zero out any previously accumulated gradients, then ,
        optimizer.zero_grad()
        # perform backpropagation
```

```
32          loss_value.backward()
33          # update model parameters
34          optimizer.step()
35
36          # add the loss_value to the total training loss so far
37          train_loss += loss_value
38
39          # computing training accuracy
40          train_acc += (torch.argmax(pred, 1) == torch.argmax(y, 1)).float().sum()
41
42          train_steps += len(y)
43
44      # switch off autograd
45      with torch.no_grad():
46
47          # set the model in evaluation mode
48          model.eval()
49
50          val_steps = 0
51          for X, y in val_loader:
52
53              X, y = X.to(device), y.to(device)
54
55              # make the predictions and calculate the validation loss
56              pred = model.forward(X)
57              loss_value = loss(pred, y)
58              val_loss += loss_value
59              val_steps += len(y)
60
61              # calculating validation accuracy
62              val_acc += (torch.argmax(pred, 1) == torch.argmax(y, 1)).float().sum()
63
64      # update training history
65      history["train_loss"].append((train_loss/train_steps).cpu().detach().numpy())
66      history["val_loss"].append((val_loss/val_steps).cpu().detach().numpy())
67      history["train_acc"].append((train_acc/train_steps).cpu().detach().numpy()*100)
68      history["val_acc"].append((val_acc/val_steps).cpu().detach().numpy()*100)
69
70      # print the model training and validation information
71      print("\nEPOCH: {}/{}".format(e, num_epochs))
72      print("\nTrain loss: {:.6f}, Validation loss: {:.6f}".format(
73          history["train_loss"][e-1], history["val_loss"][e-1]))
74      print("\nTrain acc: {:.2f}%, Validation acc: {:.2f}%".format(
```

```
75          history["train_acc"][e-1], history["val_acc"][e-1]))
76
77      jb.dump(history, f"/kaggle/working/model_history({e}).pkl")
78      model_scripted = torch.jit.script(model) # Export to TorchScript
79      model_scripted.save(f"/kaggle/working/model_state({e}).pt")
```

3. Resultados e discussão

3.1. Histórico de treinamento

A função que construí para uma melhor visualização do histórico de aprendizado dos modelos de redes neurais convolucionais que serão testados está mostrada logo abaixo. Posteriormente, serão mostrados e discutidos os resultados da aplicação dessa função sobre cada modelo.

```
import matplotlib.pyplot as plt
import joblib as jb

# plotting function for the learning history of each CNN
def plot_history(jb_path, output_path=None, title=None):

    # loading history dict
    history = jb.load(jb_path)

    # separatind important variables and processing it
    train_loss = history["train_loss"]
    train_loss = [float(value) for value in train_loss]

    val_loss = history["val_loss"]
    val_loss = [float(value) for value in val_loss]

    train_acc = history["train_acc"]
    val_acc = history["val_acc"]

    epochs = range(1, len(train_loss) + 1)

    # generating plot
    fig, axs = plt.subplots(2, 1, sharex=True, figsize=(7, 5))

    axs[0].plot(epochs, train_loss,
                linestyle="--", marker="o", markersize=3,
                color="gray")
    axs[0].plot(epochs, val_loss,
                linestyle="--", marker="o", markersize=3,
                color="black")
    axs[0].set_ylabel("Valor da função de perda")

    axs[1].plot(epochs, train_acc, label="Treino",
                linestyle="--", marker="o", markersize=3,
```

```
35                 color="gray")
36     axs[1].plot(epochs, val_acc, label="Validação",
37                 linestyle="--", marker="o", markersize=3,
38                 color="black")
39     axs[1].set_xlabel("Época")
40     axs[1].set_ylabel("Acurácia total (%)")
41     axs[1].set_ylim(0, 110)
42     axs[1].legend()
43
44     if title is not None:
45         axs[0].set_title(title)
46
47     # if necessary, the figure will be saved as history.png
48     if output_path is not None:
49         plt.tight_layout()
50         plt.savefig(output_path + "/history.png", dpi=400)
51
52     plt.show()
```

3.1.1. Modelo 1: ResNet simplificada SEM data augmentation Sem data augmentation, foram necessárias 32 épocas de treinamento no total, como pode ser visto na figura 8. Nas primeiras 30 épocas foi usado um learning rate de 10^{-4} enquanto que nas últimas 2 foi usado um learning rate de 10^{-5}. Note que houve um ganho de performance após mudar o learning rate, como já seria de se esperar devido ao menor reajuste dos pesos. No fim, o melhor modelo apresentou uma acurácia total de treino de aproximadamente 100%, enquanto que nos dados de validação a acurácia total foi de 76.40%.

Na figura 8, é possível observar que a curva de treino saturou de forma suave nas 32 épocas, enquanto que a curva de validação flutuou muito e ficou consideravelmente distante da curva de treino. Isso sugere que o modelo de classificação não conseguiu se generalizar bem para dados fora do conjunto de treinamento.

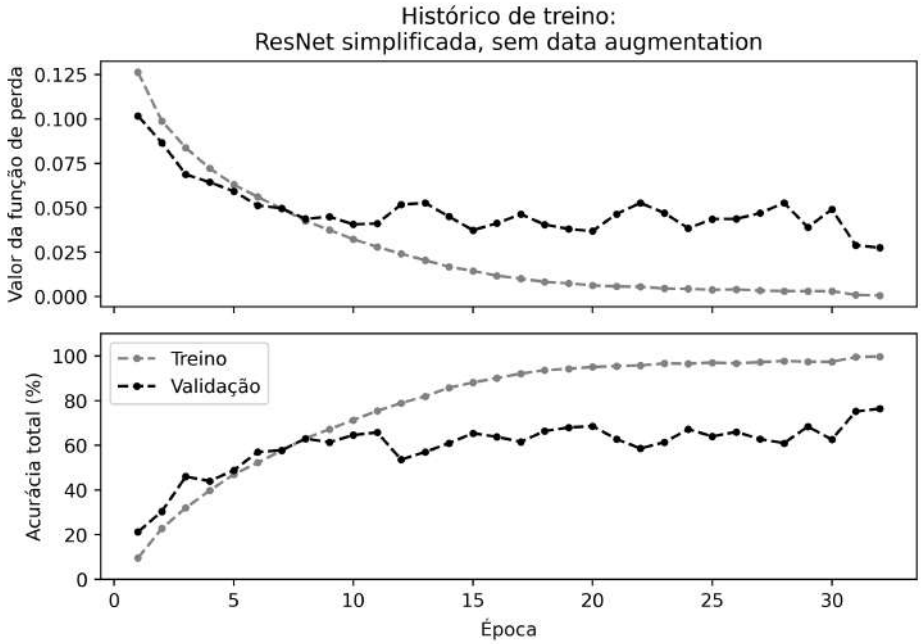

Figure 8. Histórico de treinamento do modelo 1 sem data augmentation.

3.1.2. Modelo 1: ResNet simplificada COM data augmentation Com *data augmentation*, a quantidade de iterações para a saturação de aprendizado do modelo foi maior. Foram necessárias 50 épocas com *learning rate* de 10^{-4} e mais 10 épocas com *learning rate* de 10^{-5} (totalizando 60 épocas no total). Em compensação, o poder de generalização do modelo foi melhor. Para o melhor modelo, a acurácia total nos dados de treino foi de 99.99%, enquanto que nos dados de validação essa mesma métrica apresentou o valor de 84.00%, valor consideravelmente maior do que sem *data augmentation*.

É perceptível a diferença no comportamento das curvas com e sem *data augmentation*. Pelo gráfico da figura 9, é possível notar que as curvas de treino e validação saturam praticamente juntas, sem tanta flutuação na curva de validação. A performance do modelo poderia ser bem melhor caso fossem fornecidas mais imagens para cada uma das 100 classes, tal como ocorre do *dataset* ImageNet.

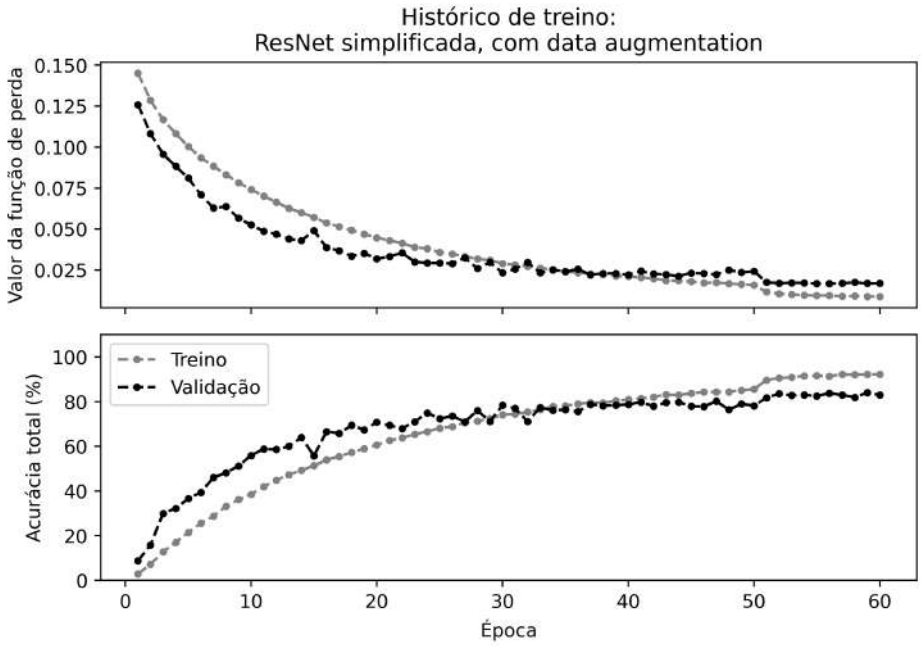

Figure 9. Histórico de treinamento do modelo 1 com data augmentation.

3.1.3. Modelo 2: ResNet simplificada com blocos inception, SEM data augmentation O aprendizado do modelo 2 sem data augmentation foi parecido com o que ocorreu para o modelo 1 no mesmo contexto. Foram necessárias 35 épocas no total para o treino do modelo, onde as 30 primeiras épocas foram com learning rate de 10^{-4} e as restantes com learning rate de 10^{-5}. A acurácia total de treino para o melhor modelo foi de 100% enquanto que nos dados de validação a acurácia total foi de 73.40%. Aparentemente a adição de blocos inception para este *dataset* não resultou numa melhora da performance do modelo de classificação.

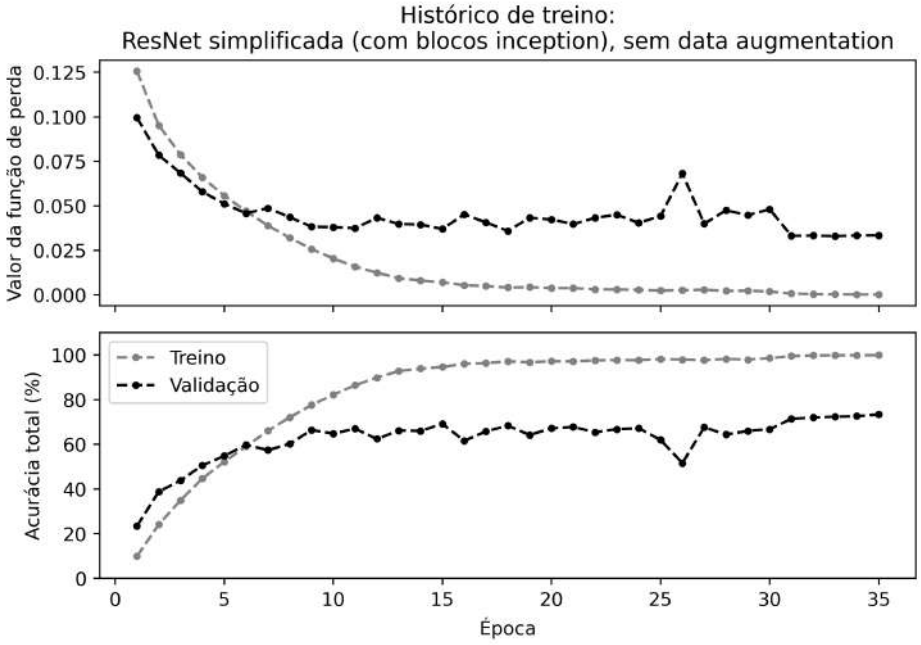

Figure 10. Histórico de treinamento do modelo 2 sem data augmentation.

3.1.4. Modelo 2: ResNet simplificada com blocos inception, COM data augmentation
Foram necessárias 50 épocas no total para o treino do modelo, com as 40 primeiras tendo um learning rate de 10^{-4} e as restantes um learning rate de 10^{-5}. O uso de *data augmentation* novamente se demonstrou útil para uma melhor generalização das redes neurais convolucionais. Veja pela figura 11 que as curvas de treino e de validação ficaram próximas, com o melhor modelo apresentando uma acurácia total nos dados de treino de 99.99% enquanto que nos dados de validação essa mesma métrica foi de 82.40%. Comparado ao modelo 1 no mesmo contexto, a adição de blocos *inception* não resultou numa melhora de performance. Isso obviamente vale para este *dataset* em específico e pode não ser verdadeiro em outros *datasets*.

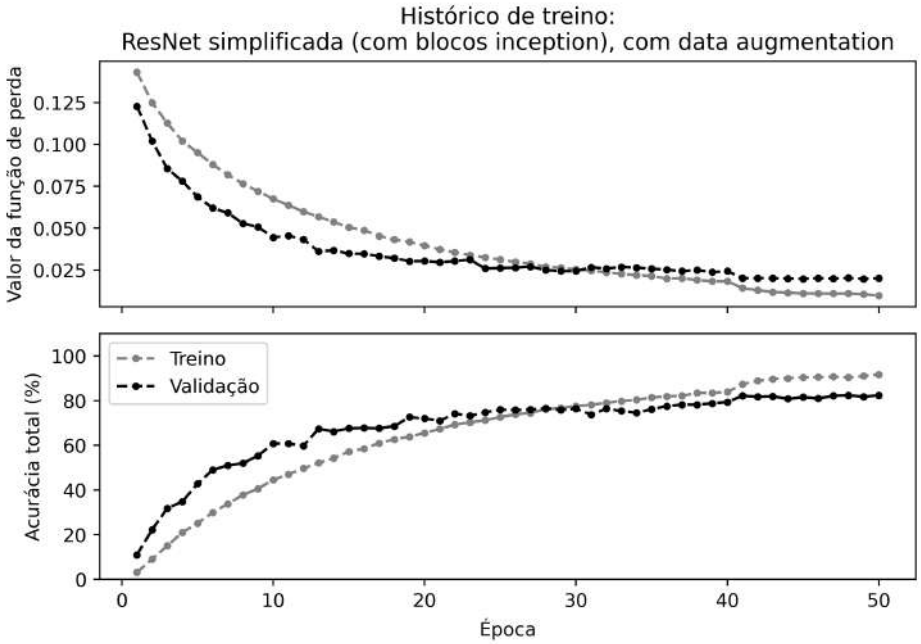

Figure 11. Histórico de treinamento do modelo 2 com data augmentation.

3.1.5. Um terceiro modelo: finetuning da ResNet-18 (transfer learning) No finetuning da ResNet-18, foram necessárias poucas épocas: 10 no total, 7 das quais com learning rate de 10^{-4} e as restantes com learning rate de 10^{-5}. A diferença no comportamento das curvas de aprendizado comparado com os gráficos anteriores é grande. Além da rápida saturação das curvas, o modelo apresentou uma generalização maior do que todos as redes testadas anteriormente, com o melhor deles tendo uma acurácia total de treino de 99.99% e de validação igual a 96.80%. Isso deixa claro que a inicialização dos pesos faz toda a diferença no treinamento e na capacidade de generalização de redes neurais convolucionais. Obviamente, em relação aos outros modelos testados aqui, existe a diferença da quantidade de camadas utilizadas e entre outros detalhes. Entretanto, a chance de *overfitting* no treinamento *from scratch* de uma rede tão grande quanto a ResNet-18 é significativa considerando que o *dataset* de esportes é muito menor que o do ImageNet.

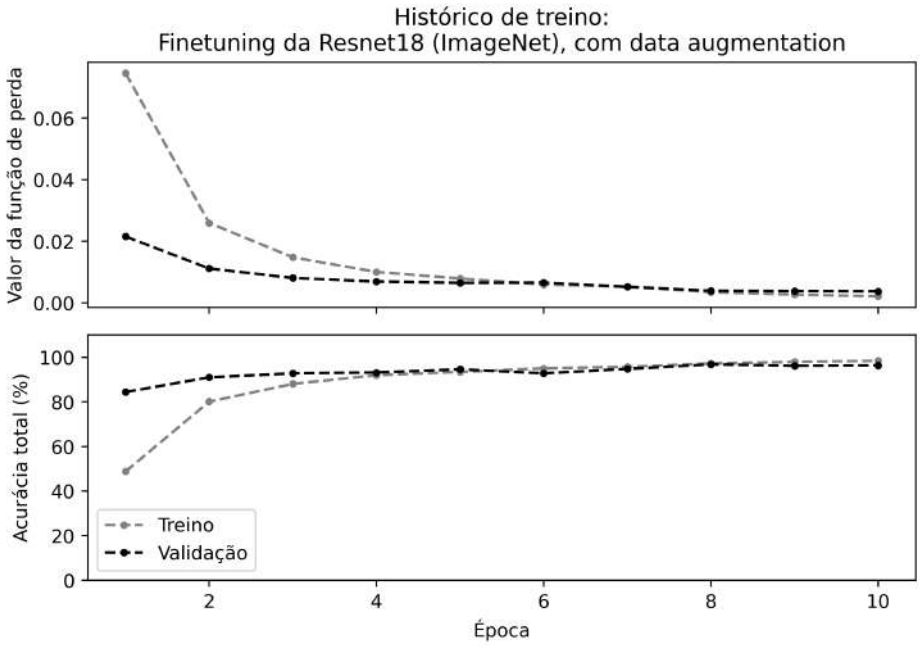

Figure 12. Histórico de *finetuning* da ResNet-18 com data augmentation.

3.2. Resultados sobre os dados de teste

Para averiguar de forma mais específica a performance dos modelos treinados, precisamos aplicá-los sobre os dados de teste e analisar os resultados de predição com métricas mais minuciosas que a acurácia total. Apesar de a acurácia ser uma métrica que indica a porcentagem de acertos dos modelos de classificação, ela possui interpretabilidade limitada quando utilizada em problemas de múltiplas classes, principalmente quando há desbalanceamento dos dados. Este não é o nosso caso, mas pensemos juntos: o que significa uma acurácia total, por exemplo, de 84.00% num dataset com 100 classes aproximadamente balanceadas? O modelo pode estar se saindo bem em 84 classes mas tendo uma performance péssima nas restantes. Ou ele pode estar acertando 84.00% dos dados de cada classe. Enfim, como deu para perceber, é difícil resumir a performance em um único número.

Nesse sentido, faremos uso de três métricas diferentes: sensibilidade (*recall*), precisão (*precision*) e *F1-score* (média harmônica entre sensibilidade e precisão). Não é da minha intenção explicar tudo sobre cada uma dessas métricas (você pode saber mais clicando aqui), mas farei uma breve interpretação delas aqui:

- Sensibilidade (*recall*): métrica que mede a quantidade de acertos em uma classe considerando o total de instâncias dessa mesma classe;
- Precisão (*precision*): métrica que mede a quantidade de acertos em uma classe considerando o total de vezes que a classe foi predita;
- *F1-score*: "balanço" entre sensibilidade e precisão (já que o *F1-score* é a média harmônica entre os dois números);

Ou seja, tais métricas são métricas específicas para cada classe. Mas temos 100 no total! Eu deveria, então, criar uma tabela e saturar sua percepção visual com 300 números diferentes? Não, essa não é minha intenção. Por isso criei uma função que facilitará nossa visualização dos resultados sobre cada classe, adicionando cores que facilitarão a leitura de tudo.

Primeiro precisamos gerar as predições sobre os dados de teste. Para um determinado modelo e dados de teste, a função abaixo que criei retornará os labels de referência e os labels preditos pelo modelo:

```
import torch

def eval_model(model, test_loader):

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

    model.to(device)
    model.eval()

```

```
    y_pred = []
    y_test = []
    test_acc = 0
    test_steps = 0

    # switch off autograd
    with torch.no_grad():

        # set the model in evaluation mode
        for X, y in test_loader:

            X, y = X.to(device), y.to(device)

            # make the predictions
            pred = model.forward(X)
            test_steps += len(y)
            # saving predictions and reference labels
            y_pred.extend(list(torch.argmax(pred, 1).cpu().numpy()))
            y_test.extend(list(torch.argmax(y, 1).cpu().numpy()))

            # calculating validation accuracy for reference
            test_acc += (torch.argmax(pred, 1) == torch.argmax(y, 1)).float().sum()
    print("Total accuracy over test set:", float((test_acc/test_steps).cpu().numpy())*100)

    return y_test, y_pred
```

A função que tomará como input as predições e os labels de referência para gerar uma tabela colorida com os valores de sensibilidade, precisão e F1-score para cada uma das classes do *dataset* está mostrada logo abaixo. Note que há também outros argumentos relacionados à estética da tabela. Cabe ao leitor/usuário do código fazer os próprios testes e ver o que serve a ele.

```

from sklearn.metrics import classification_report
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np

def plot_classification_report(y_test, y_pred, labels, target_names,
                               vmin, vmax, annot_size, x_size, y_size,
                              title, output_path):
    fig, ax = plt.subplots(figsize=(4, 15))
```

```
12      clf_report = classification_report(y_test,
13                                         y_pred,
14                                         labels=labels,
15                                         target_names=target_names,
16                                         output_dict=True)
17      df = pd.DataFrame(clf_report).T
18      df.index = [list(df.index)[i] + " (" + str(int(support)) + ")" if list(df.index)[i] != "accuracy" el
19
20      sns.heatmap(df.iloc[51:100, :-1], annot=True, ax=ax, vmin=.6,
21                  vmax=1.25, cbar=False, annot_kws={"fontsize":annot_size}, cmap="PiYG")
22      ax.set_xticklabels(ax.get_xmajorticklabels(), fontsize = x_size)
23      ax.set_yticklabels(ax.get_ymajorticklabels(), fontsize = y_size)
24      ax.xaxis.tick_top()
25
26      if title is not None:
27          ax.set_title(title)
28
29      if output_path is not None:
30          plt.tight_layout()
31          plt.savefig(output_path + "/class_report.png", dpi=400)
32
33      plt.show()
34
```

Nas páginas seguintes, serão mostrados os resultados sobre os dados de teste para cada um dos modelos treinados. Por fim, haverá uma subseção de discussão e comparação para um melhor entendimento das tabelas.

3.2.1. Modelo 1: ResNet simplificada SEM data augmentation

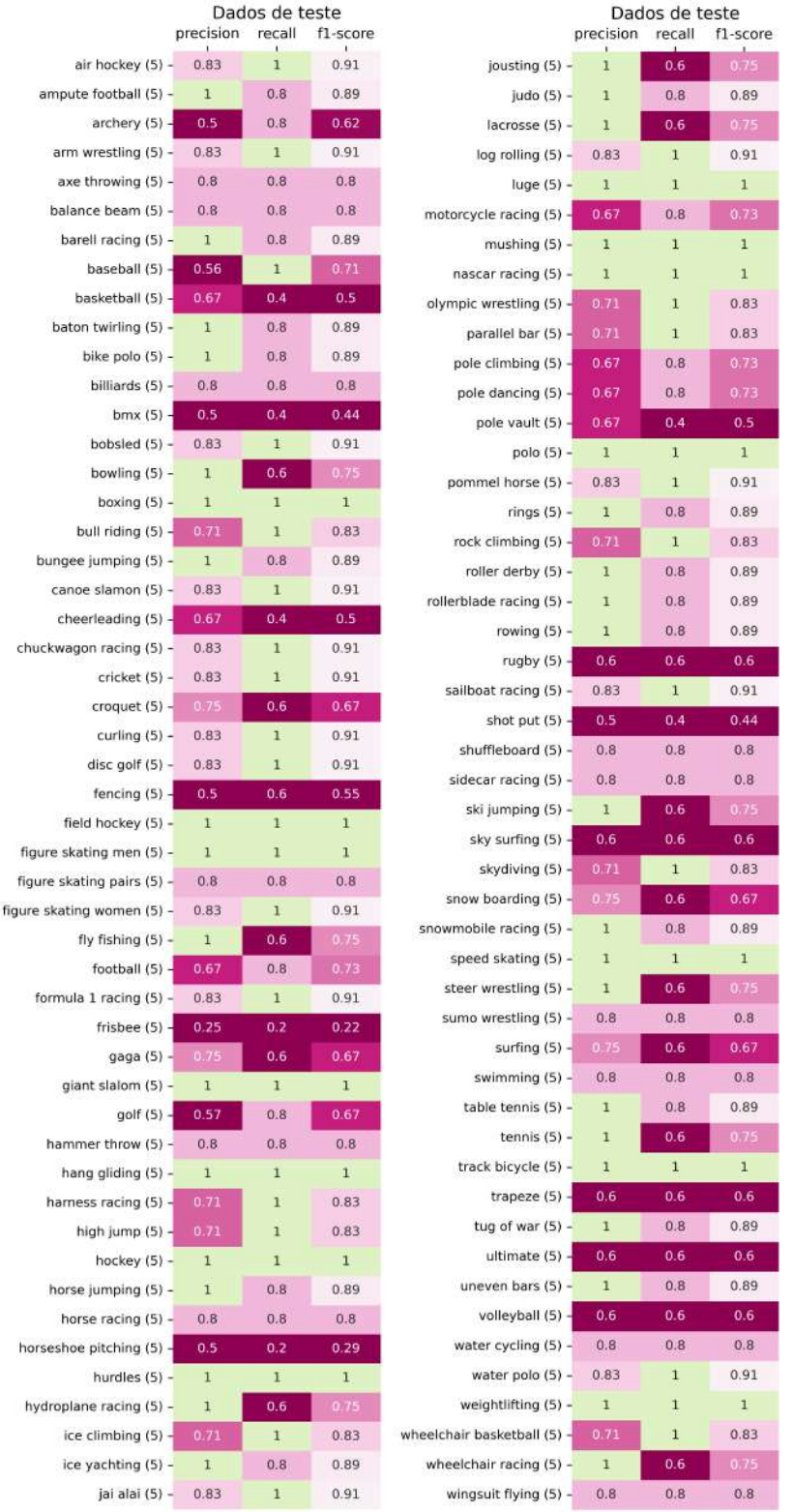

Figure 13. Performance sobre os dados de teste do modelo 1 treinado sem data augmentation.

3.2.2. Modelo 1: ResNet simplificada COM data augmentation

Figure 14. Performance sobre os dados de teste do modelo 1 treinado com data augmentation.

3.2.3. Modelo 2: ResNet simplificada com blocos inception, SEM data augmentation

Figure 15. Performance sobre os dados de teste do modelo 2 treinado sem data augmentation.

3.2.4. Modelo 2: ResNet simplificada com blocos inception, COM data augmentation

Figure 16. Performance sobre os dados de teste do modelo 2 treinado com data augmentation.

3.2.5. Um terceiro modelo: finetuning da ResNet-18 (transfer learning)

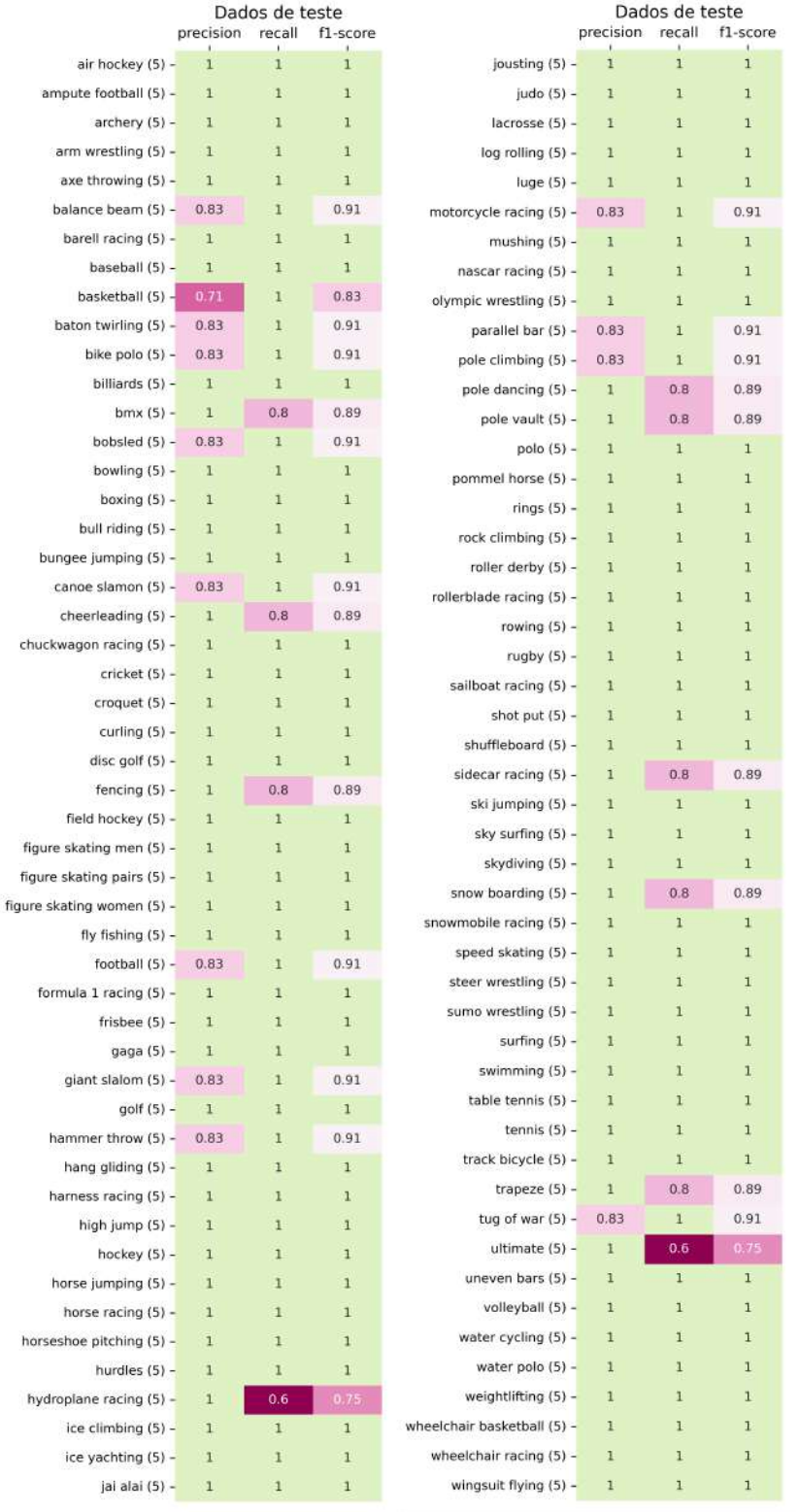

Figure 17. Performance sobre os dados de teste do modelo ResNet-18 treinado via finetuning e com data augmentation.

3.2.6. Discussão das tabelas de performance É evidente o auxílio das cores na interpretação das tabelas de performance: quanto maior o excesso de vermelho/rosa escuro, pior é a performance da rede neural convolucional. Não foi necessário conhecer o valor exato de nenhuma das métricas.

É possível abstrair facilmente, por exemplo, o quanto a técnica de *data augmentation* melhora a generalidade dos modelos (basta comparar sucessivamente a "quantidade de vermelho" entre as tabelas). A técnica de *data augmentation* foi capaz de melhorar significativamente a sensibilidade e a precisão de várias classes ao mesmo tempo para um mesmo modelo, e isso nos dados de teste.

Outro fato simples de perceber é o quanto a técnica de *finetuning* da ResNet-18 foi melhor que as outras tentativas de modelo. Na tabela de performance, praticamente não há a cor vermelha, indicando que os valores de sensiblidade e precisão nos dados de teste são ótimos para várias classes diferentes.

Uma forma resumida e numérica de comparar as tabelas de performance entre todos os modelos testados é através de uma tabela com as médias de sensibilidade, precisão e F1-score entre todas as classes. Ademais, é interessante incluir os valores de acurácia total. Essa tabela está mostrada adiante, na tabela 1

Table 1. Tabela com a média entre as classes das métricas Recall, Precision e F1-score. Adicionalmente, está mostrada a acurácia total. Os resultados da tabela são todos provenientes dos dados de teste. Legenda: Rec. (Recall médio entre as classes); Prec. (Precision médio entre as classes); F1 (F1-score médio entre as classes); Acc. (Acurácia total); DA (data augmentation)

	Rec. (%)	**Prec. (%)**	**F1 (%)**	**Acc. (%)**
Modelo 1 (sem DA)	82.57	80.80	80.34	80.80
Modelo 1 (com DA)	88.79	87.40	87.26	87.40
Modelo 2 (sem DA)	77.61	76.60	75.56	76.60
Modelo 2 (com DA)	86.24	84.60	84.25	84.60
ResNet-18 (finetuning)	97.71	97.20	97.10	97.20

Note que novamente a melhora na performance devido ao *data augmentation* fica evidente comparando-se usando o mesmo modelo. Note também que, de longe, o finetuning da ResNet-18 foi o modelo que mais se generalizou bem para os dados de teste.

Como não usamos os dados de teste em nenhuma ocasião durante o treinamento do modelo, logo todas as métricas calculadas até aqui servem como indicativo de como os modelos que treinei se comportariam na vida real. Obviamente, para que isso ocorresse à risca, a mesma política de coleta de dados deve ser mantido como ocorreu com os dados de treino, validação e teste.

4. Conclusão

O impacto da técnica de data augmentation foi evidente. Nos dados de teste, ela foi capaz de aumentar a performance de todos os modelos que treinei. Apesar de também ter aumentado a quantidade de épocas para treino, é um sacrifício pequeno perto do ganho em performance que foi demonstrado neste pequeno estudo.

Ademais, ficou mais do que claro a importância dos valores iniciais dos pesos das redes neurais convolucionais. Eles impactam principalmente durante o treinamento e facilitam a otimização dos pesos, com benefícios tanto em velocidade de treino quanto em performance e generalização. Assim, em datasets consideravelmente pequenos (pelo menos quando comparados com o ImageNet), é extremamente pertinente aplicar o *finetuning* de redes convolucionais pré-treinadas, como a ResNet-18.

5. Materiais suplementares

O código utilizado para gerar as figuras seguintes está mostrado logo a seguir:

```
1
2  # function that evaluates an image and plot the prediction
3  def eval_and_plot(model, image_path, y_true, class_dict, ax, transforms=None):
4
5      device = "cuda" if torch.cuda.is_available() else "cpu"
6
7      image = cv2.imread(image_path)
8      image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
9
10     if transforms is not None:
11         X = transforms(image)
12
13     model.to(device)
14     model.eval()
15     X = torch.unsqueeze(X, 0).to(device)
16     model_output = model.forward(X)
17     y_pred = torch.argmax(model_output, 1).cpu().numpy()
18     prob = nn.functional.sigmoid(model_output).cpu().detach().numpy()
19     prob = prob[0, y_pred][0]
20     y_pred = class_dict[int(y_pred)]
21
22     ax.imshow(image)
23     ax.set_xticks([])
24     ax.set_yticks([])
25     ax.set_title(f"""
26     y_true: {y_true}\n
27     y_pred: {y_pred}, prob = {np.round(prob*100, 4)}%""")
28
29     return y_pred
30
31 # so we can automate a full figure with multiple subplots, you must
32 # run the following (of course, you will need to edit as you want)
33
34 import random
35
36 # for example, for this model
37 model = torch.jit.load('/kaggle/input/resnet/model_state(32)_76_40.pt')
38
39 fig, axs = plt.subplots(4, 3, figsize=(12, 16))
```

```
fig.suptitle('ResNet simplificada: exemplos de predições', fontsize=16)

for i in range(4):
    for j in range(3):
        idx = random.sample(range(len(test_DS)), k=1)[0]
        image_path = test_DS.image_paths[idx]
        label = test_DS.labels[idx]
        eval_and_plot(model,
                            image_path,
                            label,
                            train_DS.class_dict,
                      axs[i,j],
                            transforms.Compose([transforms.ToPILImage(),
                                                transforms.Resize(128),
                                                transforms.ToTensor()]))

plt.tight_layout()
# then save the plot
plt.savefig("/kaggle/working/predictions_resnet.png", dpi=400)

plt.show()
```


ResNet simplificada: exemplos de predições
y_true: figure skating women
y_pred: figure skating women, prob = 99.9914%
y_true: bungee jumping
y_pred: bungee jumping, prob = 99.9527%
y_true: wheelchair basketball
y_pred: wheelchair basketball, prob = 99.9336%
y_true: uneven bars
y_pred: uneven bars, prob = 99.9747%
y_true: horse jumping
y_pred: horse jumping, prob = 99.0228%
y_true: jai alai
y_pred: jai alai, prob = 99.9861%
y_true: horse racing
y_pred: harness racing, prob = 96.2601%
y_true: cheerleading
y_pred: basketball, prob = 99.9634%
y_true: ampute football
y_pred: ampute football, prob = 87.4149%
y_true: roller derby
y_pred: wheelchair basketball, prob = 99.9616%
y_true: hammer throw
y_pred: shot put, prob = 99.1562%
y_true: field hockey
y_pred: field hockey, prob = 100.0%

Figure 18.

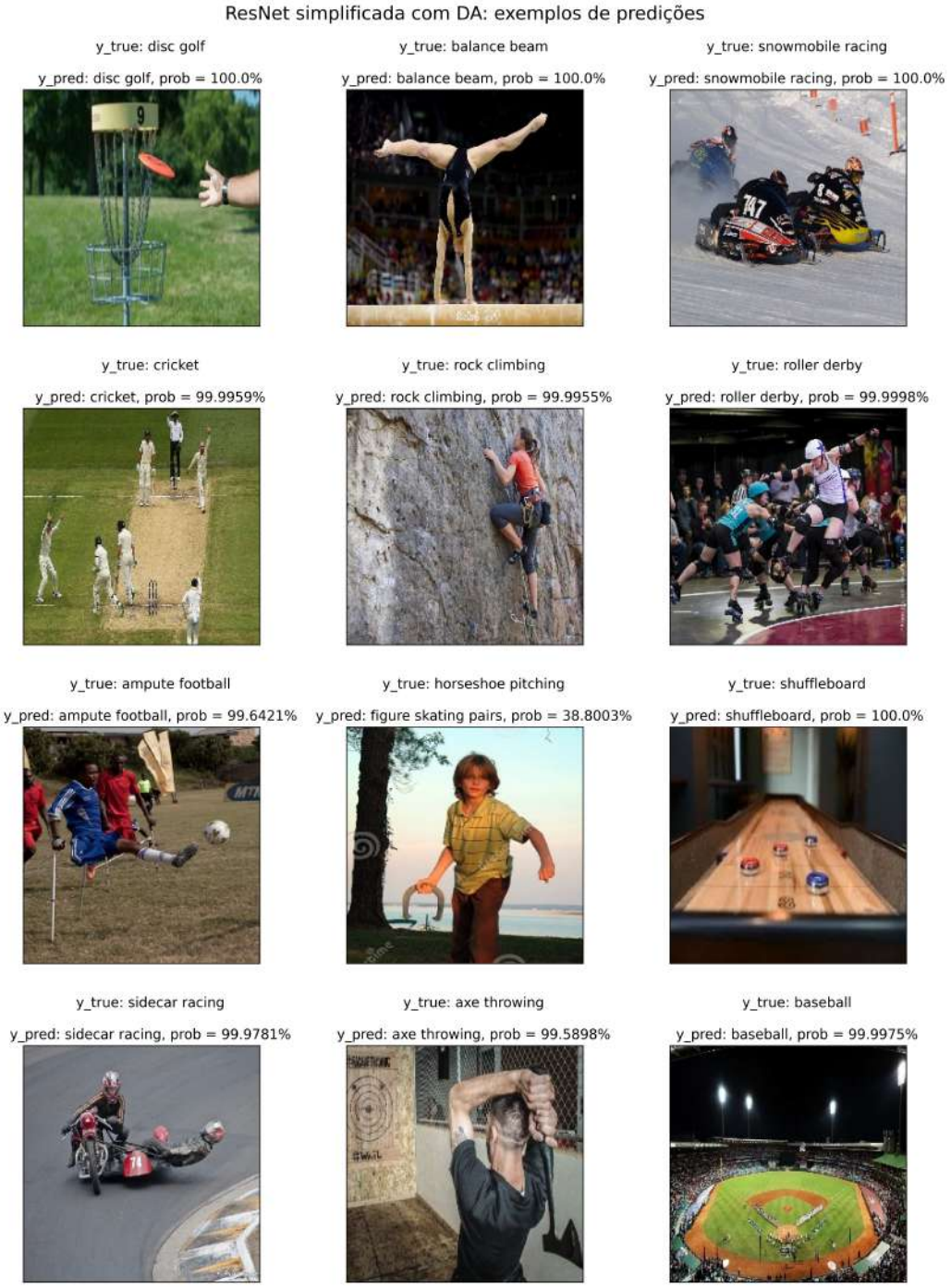
ResNet simplificada com DA: exemplos de predições
y_true: disc golf
y_pred: disc golf, prob = 100.0%
y_true: balance beam
y_pred: balance beam, prob = 100.0%
y_true: snowmobile racing
y_pred: snowmobile racing, prob = 100.0%
y_true: cricket
y_pred: cricket, prob = 99.9959%
y_true: rock climbing
y_pred: rock climbing, prob = 99.9955%
y_true: roller derby
y_pred: roller derby, prob = 99.9998%
y_true: ampute football
y_pred: ampute football, prob = 99.6421%
y_true: horseshoe pitching
y_pred: figure skating pairs, prob = 38.8003%
y_true: shuffleboard
y_pred: shuffleboard, prob = 100.0%
y_true: sidecar racing
y_pred: sidecar racing, prob = 99.9781%
y_true: axe throwing
y_pred: axe throwing, prob = 99.5898%
y_true: baseball
y_pred: baseball, prob = 99.9975%

Figure 19.

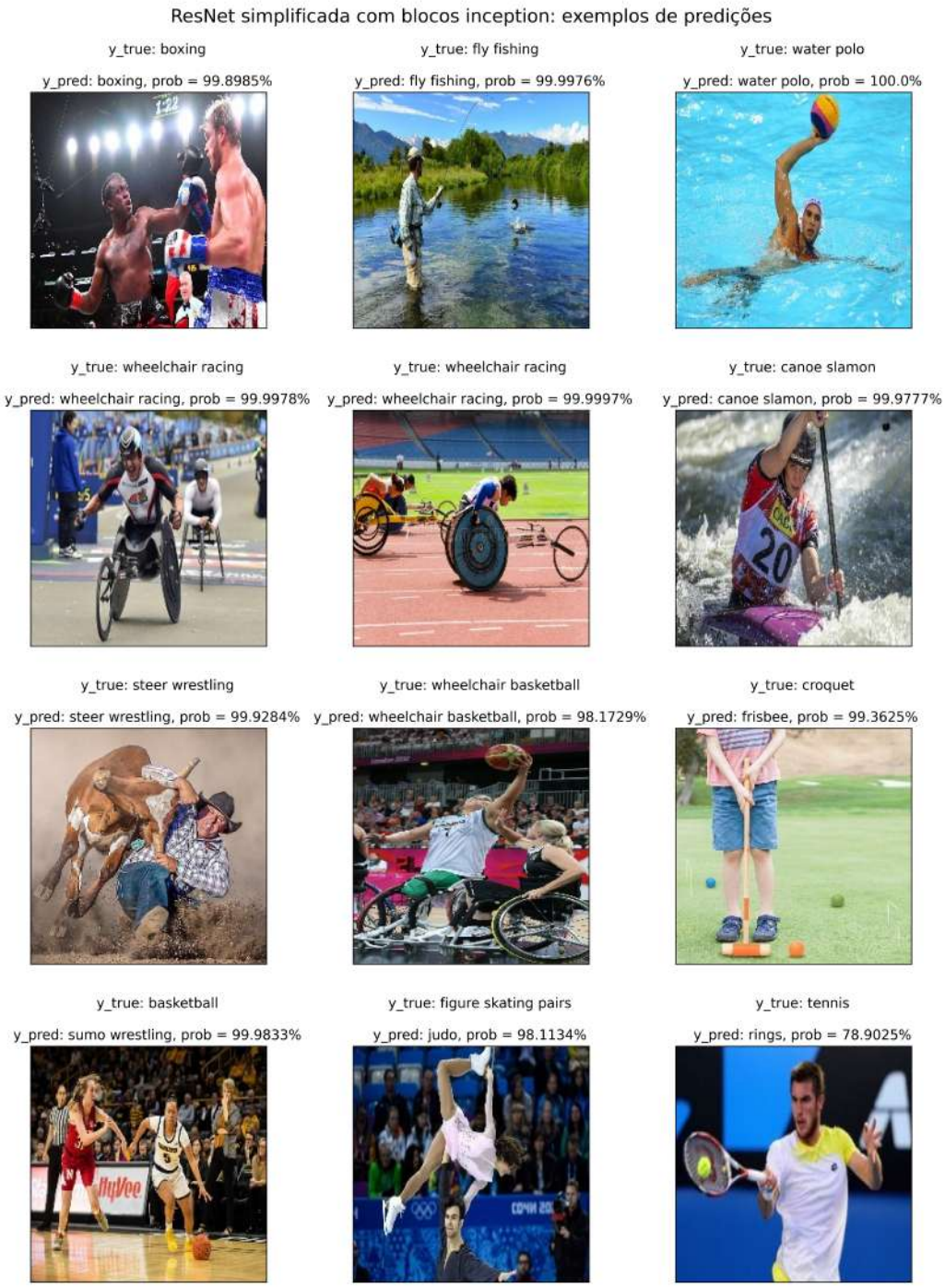
ResNet simplificada com blocos inception: exemplos de predições
y_true: boxing
y_pred: boxing, prob = 99.8985%
y_true: fly fishing
y_pred: fly fishing, prob = 99.9976%
y_true: water polo
y_pred: water polo, prob = 100.0%
y_true: wheelchair racing
y_pred: wheelchair racing, prob = 99.9978%
y_true: wheelchair racing
y_pred: wheelchair racing, prob = 99.9997%
y_true: canoe slamon
y_pred: canoe slamon, prob = 99.9777%
y_true: steer wrestling
y_pred: steer wrestling, prob = 99.9284%
y_true: wheelchair basketball
y_pred: wheelchair basketball, prob = 98.1729%
y_true: croquet
y_pred: frisbee, prob = 99.3625%
y_true: basketball
y_pred: sumo wrestling, prob = 99.9833%
y_true: figure skating pairs
y_pred: judo, prob = 98.1134%
y_true: tennis
y_pred: rings, prob = 78.9025%

Figure 20.


ResNet simplificada com blocos inception e DA: exemplos de predições
y_true: disc golf
y_pred: disc golf, prob = 100.0%
y_true: motorcycle racing
y_pred: motorcycle racing, prob = 100.0%
y_true: log rolling
y_pred: log rolling, prob = 99.7324%
y_true: ice yachting
y_pred: ice yachting, prob = 99.7875%
y_true: rowing
y_pred: rowing, prob = 99.9999%
y_true: fly fishing
y_pred: fly fishing, prob = 99.9998%
y_true: cheerleading
y_pred: cheerleading, prob = 83.9114%
y_true: ice yachting
y_pred: ice yachting, prob = 100.0%
y_true: cricket
y_pred: cricket, prob = 99.9975%
y_true: disc golf
y_pred: disc golf, prob = 99.9996%
y_true: ampute football
y_pred: ampute football, prob = 89.6978%
y_true: water cycling
y_pred: water cycling, prob = 99.9627%

Figure 21.


Finetuning da ResNet-18: exemplos de predições
y_true: bungee jumping
y_pred: bungee jumping, prob = 100.0%
y_true: ultimate
y_pred: ultimate, prob = 100.0%
y_true: ampute football
y_pred: ampute football, prob = 100.0%
y_true: table tennis
y_pred: table tennis, prob = 100.0%
y_true: weightlifting
y_pred: weightlifting, prob = 100.0%
y_true: figure skating pairs
y_pred: figure skating pairs, prob = 100.0%
y_true: roller derby
y_pred: roller derby, prob = 100.0%
y_true: bull riding
y_pred: bull riding, prob = 100.0%
y_true: tennis
y_pred: tennis, prob = 100.0%
y_true: fencing
y_pred: fencing, prob = 99.9999%
y_true: air hockey
y_pred: air hockey, prob = 100.0%
y_true: boxing
y_pred: boxing, prob = 100.0%

Figure 22.

Referências

[1] Chen L, Li S, Bai Q, Yang J, Jiang S and Miao Y 2021 *Remote Sensing* **13** ISSN 2072-4292 URL https://www.mdpi.com/2072-4292/13/22/4712

[2] He K, Zhang X, Ren S and Sun J 2015 Deep residual learning for image recognition (*Preprint* 1512.03385)

[3] Zhang A, Lipton Z C, Li M and Smola A J 2023 Dive into deep learning (*Preprint* 2106.11342)