Programa de Pós-Graduação em Engenharia de Computação e Sistemas - PECS/UEMA

Atividade III - DenseNet-121

Aluno: Adrielson Ferreira Justino

Professor: Omar Andres Carmona Cortes

- Treinamento e a comparação de duas arquiteturas de redes neurais convolucionais (CNN) para classificação de imagens, usando conjunto de dados **CIFAR-10.**

- O objetivo é treinar duas arquiteturas de modelos diferentes **(model1 e model2)** e comparar suas performances usando testes estatísticos.

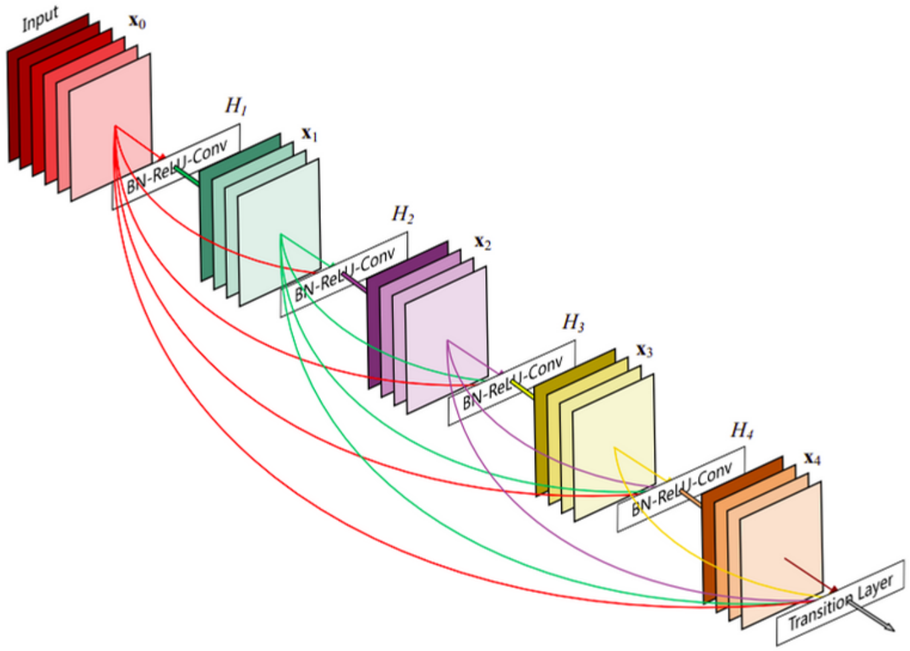
DenseNet-121

- Arquitetura apresentada em 2016 por Gao Huang et al. em seu artigo da “Densely Connected Convolutional Networks”.

- Redes Convolucionais Densamente Conectadas (DenseNet) é uma arquitetura de rede neural convolucional (CNN) *feed-forward* que conecta cada camada a todas as outras camadas.

- Cada camada obtém informações de todas as camadas anteriores e passa seus próprios mapas de características para todas as camadas que virão depois dela.

- O modelo DenseNet surgiu com o objetivo de solucionar o problema do **desaparecimento de gradiente**, para arquiteturas muito profundas.



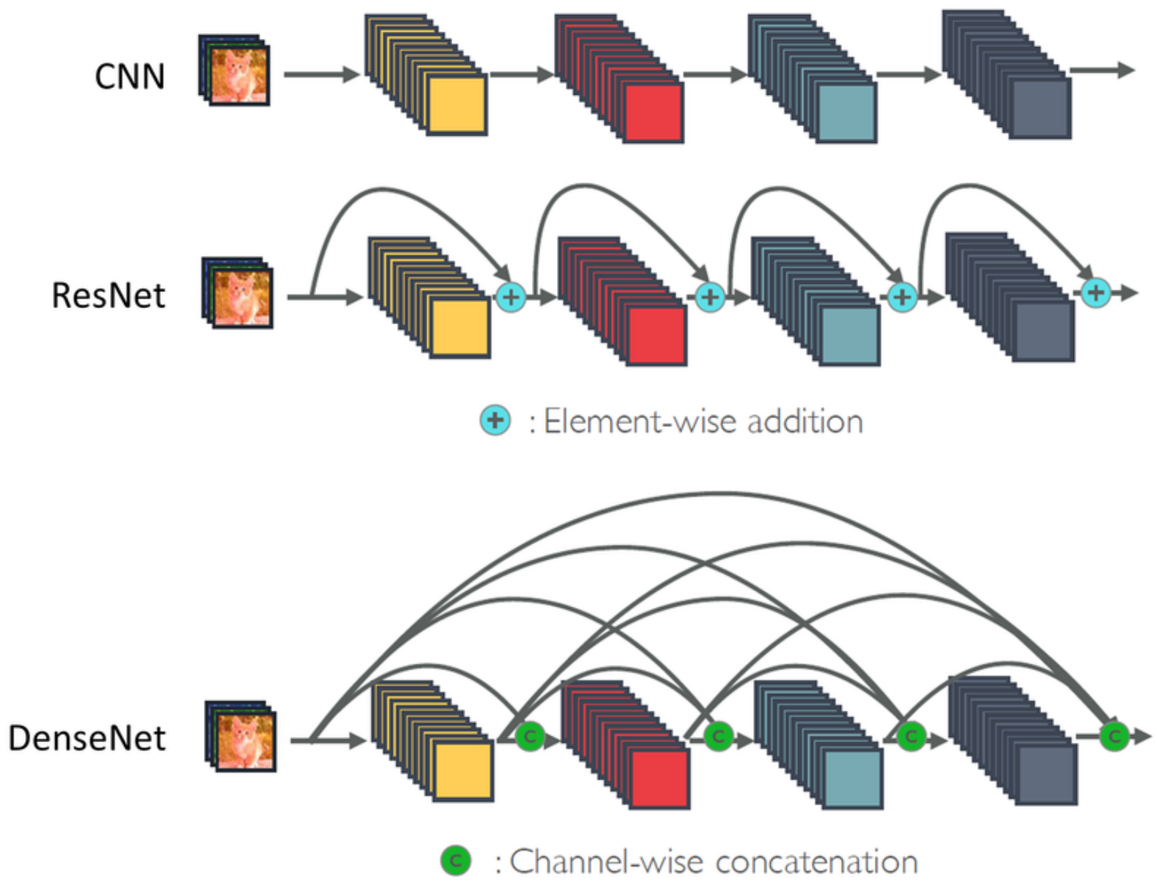
Gao Huang et al. (2016)

Rede Neural Convolucional

- Em uma CNN *feed-forward* tradicional, cada camada convolucional, exceto a primeira, recebe a saída da camada convolucional anterior;

- Produz um mapa de características de saída que é então passado para a próxima camada convolucional;

- Portanto, para camadas 'L', existem conexões diretas 'L'.

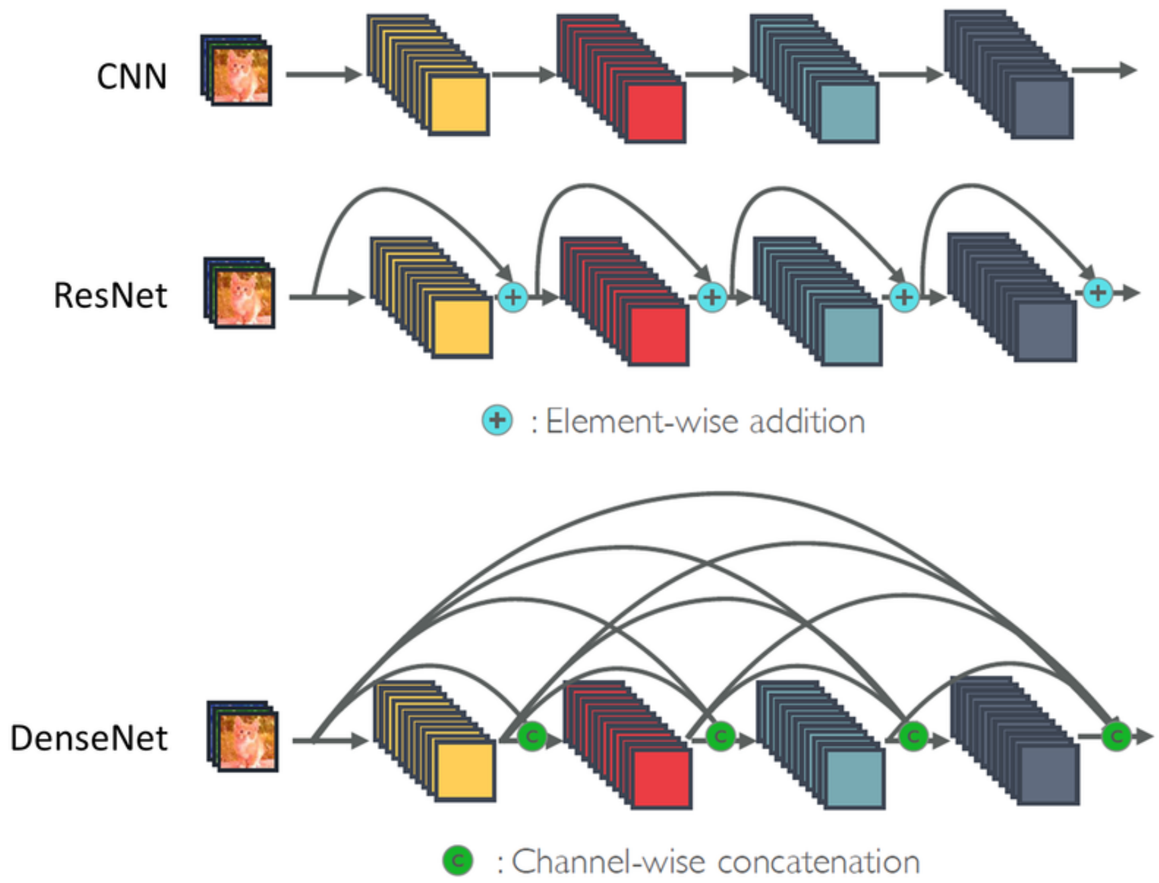


Rede Neural Convolucional

- À medida que o número de camadas na CNN aumenta, surge o problema do **“desaparecimento de gradiente”**.

- Conforme o caminho das informações das camadas de entrada para as camadas de saída aumenta, certas informações podem 'desaparecer' ou se perder;

- Reduz a capacidade da rede de treinar de forma eficaz.



Conectividade

- A ideia chave empregada pela DenseNet é, em cada camada, os mapas de características são passados como entrada não só para a camada subsequente, mas para todas as camadas até o final da rede, em uma estrutura denominada **bloco dense**

- Em cada camada, os mapas de características de todas as camadas anteriores não são somados, mas concatenados e usados como entradas.

- Conexões **L(L+1)/2** na rede, em vez de apenas conexões **L** como nas arquiteturas tradicionais de aprendizagem profunda.

- DenseNets requerem menos parâmetros do que uma CNN tradicional equivalente, e isso permite a reutilização de recursos à medida que mapas de características redundantes são descartados.

Arquitetura (Implementação)

- DenseNet consiste em dois blocos importantes além das camadas convolucionais e de pooling básicas:
  - **Blocos Densos**
  - **Camadas de Transição**

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	[1 × 1 conv, 3 × 3 conv] × 24	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 48	[1 × 1 conv, 3 × 3 conv] × 64
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	[1 × 1 conv, 3 × 3 conv] × 16	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 48
Classification Layer	1 × 1	7 × 7 global average pool			
		1000D fully-connected, softmax			

Implementação: https://www.kaggle.com/adrielson/comparativo-cnn-densenet-121

Arquitetura (Implementação)

- DenseNet começa com uma convolução básica com 64 filtros de tamanho 7X7 e stride de 2;
- Seguido por uma camada MaxPooling com pooling máximo de 3x3 e um avanço de 2.

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	[1 × 1 conv, 3 × 3 conv] × 24	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 48	[1 × 1 conv, 3 × 3 conv] × 64
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	[1 × 1 conv, 3 × 3 conv] × 16	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 48
Classification Layer	1 × 1	7 × 7 global average pool			
		1000D fully-connected, softmax			

```
input = Input(input_shape)
x = Conv2D(64, 7, strides=2, padding='same')(input)
x = MaxPooling2D(3, strides=2, padding='same')(x)
```

Arquitetura (Implementação)

Blocos densos

- Cada bloco denso tem duas convoluções, com núcleos de tamanho 1x1 e 3x3.
- No bloco denso 1, isso é repetido 6 vezes, no bloco denso 2 é repetido 12 vezes, no bloco denso 3, 24 vezes e finalmente no bloco denso 4, 16 vezes.

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	[1 × 1 conv, 3 × 3 conv] × 24	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 48	[1 × 1 conv, 3 × 3 conv] × 64
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	[1 × 1 conv, 3 × 3 conv] × 16	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 48
Classification Layer	1 × 1	7 × 7 global average pool			
		1000D fully-connected, softmax			

Arquitetura (Implementação)

- Cada bloco convolucional após a entrada tem a seguinte sequência: BatchNormalization, Ativação do ReLU, Camada Conv2D real.

```
def densenet(input_shape, n_classes):
    def bn_rl_conv(x, filters, kernel=1, strides=1):
        x = BatchNormalization()(x)
        x = ReLU()(x)
        x = Conv2D(filters, kernel, strides=strides, padding='same')(x)
        return x
```

- No bloco denso, cada uma das convoluções 1x1 possui 4 vezes o número de filtros (4*), mas os filtros 3x3 estão presentes apenas uma vez. Além disso, é feita a concatenação da entrada com o tensor de saída. O loop executa cada bloco em 6,12,24,16 repetições respectivamente.

```
    def dense_block(x, repetition):
        for _ in range(repetition):
            y = bn_rl_conv(x, 4 * growth_rate)
            y = bn_rl_conv(y, growth_rate, 3)
            x = Concatenate()([y, x])
        return x
```

Arquitetura (Implementação)

Camada de transição

- Há uma camada convolucional 1x1
- Um camada de pooling média 2x2 com um avanço de 2.

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6
Transition Layer (1)	56 × 56 28 × 28	1 × 1 conv 2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12
Transition Layer (2)	28 × 28 14 × 14	1 × 1 conv 2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	[1 × 1 conv, 3 × 3 conv] × 24	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 48	[1 × 1 conv, 3 × 3 conv] × 64
Transition Layer (3)	14 × 14 7 × 7	1 × 1 conv 2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	[1 × 1 conv, 3 × 3 conv] × 16	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 48
Classification Layer	1 × 1	7 × 7 global average pool 1000D fully-connected, softmax			

```
def transition_layer(x):
    x = bn_rl_conv(x, K.int_shape(x)[-1] // 2)
    x = AveragePooling2D(2, strides=2, padding='same')(x)
    return x
```

Arquitetura (Implementação)

- Concluído a definição dos blocos densos e das camadas de transição. É preciso empilhar os blocos densos e as camadas de transição. Por meio do *loop for* para percorrer as 6,12,24,16 repetições.

```
    for repetition in [6, 12, 24, 16]:
        x = dense_block(x, repetition)
        x = transition_layer(x)
```

Arquitetura (Implementação)

Camada de classificação

- **Global Average Pooling** - aceita todos os mapas de recursos da rede para realizar a classificação
- Camada de saída **fully-connected**

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	[1 × 1 conv, 3 × 3 conv] × 24	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 48	[1 × 1 conv, 3 × 3 conv] × 64
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	[1 × 1 conv, 3 × 3 conv] × 16	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 48
Classification Layer	1 × 1	7 × 7 global average pool			
		1000D fully-connected, softmax			

```
x = GlobalAveragePooling2D()(x)
output = Dense(n_classes, activation='softmax')(x)
```

Arquitetura (Implementação)

- Portanto, DenseNet-121 possui as seguintes camadas:
  - **1 Convolução 7x7**
  - **58 Convolução 3x3**
  - **61 Convolução 1x1**
  - **4Pool médio**
  - **1 camada totalmente conectada**

```
#DenseNet-121
def densenet(input_shape, n_classes):
    def bn_rl_conv(x, filters, kernel=1, strides=1):
        x = BatchNormalization()(x)
        x = ReLU()(x)
        x = Conv2D(filters, kernel, strides=strides, padding='same')(x)
        return x

    def dense_block(x, repetition):
        for _ in range(repetition):
            y = bn_rl_conv(x, 4 * growth_rate)
            y = bn_rl_conv(y, growth_rate, 3)
            x = Concatenate()([y, x])
        return x

    def transition_layer(x):
        x = bn_rl_conv(x, K.int_shape(x)[-1] // 2)
        x = AveragePooling2D(2, strides=2, padding='same')(x)
        return x

    growth_rate = 32

    input = Input(input_shape)
    x = Conv2D(64, 7, strides=2, padding='same')(input)
    x = MaxPooling2D(3, strides=2, padding='same')(x)

    for repetition in [6, 12, 24, 16]:
        x = dense_block(x, repetition)
        x = transition_layer(x)

    x = GlobalAveragePooling2D()(x)
    output = Dense(n_classes, activation='softmax')(x)
```

Comparativo estatístico

Pré-processamento:

- O conjunto de dados CIFAR-10 foi reduzido para 30000 imagens
- 25000 para conjuto de treino
- 5000 para conjunto de teste

Experimento:

- Cada modelo foi executado 5 vezes;
- Foram coletadas as medidas de desempenho por época (10 épocas);
- Foi utilizado os valores das acurácias para realização do teste de normalidade (**Shapiro Wilk**);
- Se os grupos seguem distribuição normal;
  - Aplica-se **ANOVA**
- Senão
  - Aplica-se **Kruskal Wallis**

Modelo 1

Epoch 1/10
274/274 [==============================] - 222s 806ms/step - loss: 1.6815 - accuracy: 0.4215 - val_loss: 1.3237 - val_accuracy: 0.5136
Epoch 2/10
274/274 [==============================] - 248s 906ms/step - loss: 1.2181 - accuracy: 0.5714 - val_loss: 1.1896 - val_accuracy: 0.5807
Epoch 3/10
274/274 [==============================] - 215s 784ms/step - loss: 1.0350 - accuracy: 0.6375 - val_loss: 1.1318 - val_accuracy: 0.6016
Epoch 4/10
274/274 [==============================] - 215s 785ms/step - loss: 0.8434 - accuracy: 0.7013 - val_loss: 1.1060 - val_accuracy: 0.6200
Epoch 5/10
274/274 [==============================] - 245s 895ms/step - loss: 0.6476 - accuracy: 0.7722 - val_loss: 1.1472 - val_accuracy: 0.6295
Epoch 6/10
274/274 [==============================] - 218s 797ms/step - loss: 0.4357 - accuracy: 0.8499 - val_loss: 1.3541 - val_accuracy: 0.6052
Epoch 7/10
274/274 [==============================] - 224s 817ms/step - loss: 0.2558 - accuracy: 0.9124 - val_loss: 1.6341 - val_accuracy: 0.6208
Epoch 8/10
274/274 [==============================] - 241s 878ms/step - loss: 0.1466 - accuracy: 0.9528 - val_loss: 1.8855 - val_accuracy: 0.6059
Epoch 9/10
274/274 [==============================] - 216s 789ms/step - loss: 0.1122 - accuracy: 0.9634 - val_loss: 1.9957 - val_accuracy: 0.6108
Epoch 10/10
274/274 [==============================] - 246s 898ms/step - loss: 0.0932 - accuracy: 0.9683 - val_loss: 2.2559 - val_accuracy: 0.6176

Epoch 1/10
274/274 [==============================] - 225s 816ms/step - loss: 1.8406 - accuracy: 0.3536 - val_loss: 1.4775 - val_accuracy: 0.4737
Epoch 2/10
274/274 [==============================] - 230s 841ms/step - loss: 1.3441 - accuracy: 0.5146 - val_loss: 1.2573 - val_accuracy: 0.5539
Epoch 3/10
274/274 [==============================] - 230s 839ms/step - loss: 1.1374 - accuracy: 0.5977 - val_loss: 1.1318 - val_accuracy: 0.6001
Epoch 4/10
274/274 [==============================] - 229s 835ms/step - loss: 0.9275 - accuracy: 0.6735 - val_loss: 1.1049 - val_accuracy: 0.6228
Epoch 5/10
274/274 [==============================] - 225s 820ms/step - loss: 0.6767 - accuracy: 0.7649 - val_loss: 1.1638 - val_accuracy: 0.6244
Epoch 6/10
274/274 [==============================] - 225s 822ms/step - loss: 0.4231 - accuracy: 0.8540 - val_loss: 1.2510 - val_accuracy: 0.6444
Epoch 7/10
274/274 [==============================] - 225s 823ms/step - loss: 0.2030 - accuracy: 0.9311 - val_loss: 1.5814 - val_accuracy: 0.6351
Epoch 8/10
274/274 [==============================] - 229s 835ms/step - loss: 0.1088 - accuracy: 0.9630 - val_loss: 1.9119 - val_accuracy: 0.6257
Epoch 9/10
274/274 [==============================] - 225s 822ms/step - loss: 0.1094 - accuracy: 0.9640 - val_loss: 2.0876 - val_accuracy: 0.6183
Epoch 10/10
274/274 [==============================] - 224s 816ms/step - loss: 0.0800 - accuracy: 0.9741 - val_loss: 2.2660 - val_accuracy: 0.6347

Modelo 1

```
Epoch 1/10
274/274 [==============================] - 230s 836ms/step - loss: 1.7649 - accuracy: 0.4177 - val_loss: 1.3456 - val_accuracy: 0.5227
Epoch 2/10
274/274 [==============================] - 233s 851ms/step - loss: 1.2208 - accuracy: 0.5666 - val_loss: 1.1537 - val_accuracy: 0.5955
Epoch 3/10
274/274 [==============================] - 229s 836ms/step - loss: 1.0031 - accuracy: 0.6457 - val_loss: 1.0745 - val_accuracy: 0.6212
Epoch 4/10
274/274 [==============================] - 228s 831ms/step - loss: 0.8070 - accuracy: 0.7161 - val_loss: 1.0773 - val_accuracy: 0.6397
Epoch 5/10
274/274 [==============================] - 227s 830ms/step - loss: 0.5976 - accuracy: 0.7904 - val_loss: 1.1736 - val_accuracy: 0.6272
Epoch 6/10
274/274 [==============================] - 227s 829ms/step - loss: 0.3882 - accuracy: 0.8658 - val_loss: 1.3102 - val_accuracy: 0.6325
Epoch 7/10
274/274 [==============================] - 226s 826ms/step - loss: 0.2079 - accuracy: 0.9294 - val_loss: 1.6133 - val_accuracy: 0.6373
Epoch 8/10
274/274 [==============================] - 226s 825ms/step - loss: 0.1273 - accuracy: 0.9595 - val_loss: 1.6977 - val_accuracy: 0.6420
Epoch 9/10
274/274 [==============================] - 228s 832ms/step - loss: 0.1044 - accuracy: 0.9641 - val_loss: 2.0992 - val_accuracy: 0.6188
Epoch 10/10
274/274 [==============================] - 226s 824ms/step - loss: 0.0795 - accuracy: 0.9736 - val_loss: 2.0661 - val_accuracy: 0.6351
```

```
Epoch 1/10
274/274 [==============================] - 228s 825ms/step - loss: 1.6698 - accuracy: 0.4097 - val_loss: 1.3569 - val_accuracy: 0.5132
Epoch 2/10
274/274 [==============================] - 223s 815ms/step - loss: 1.2298 - accuracy: 0.5608 - val_loss: 1.2127 - val_accuracy: 0.5737
Epoch 3/10
274/274 [==============================] - 227s 830ms/step - loss: 1.0397 - accuracy: 0.6356 - val_loss: 1.1597 - val_accuracy: 0.5944
Epoch 4/10
274/274 [==============================] - 224s 818ms/step - loss: 0.8872 - accuracy: 0.6884 - val_loss: 1.0549 - val_accuracy: 0.6263
Epoch 5/10
274/274 [==============================] - 223s 815ms/step - loss: 0.7446 - accuracy: 0.7379 - val_loss: 1.1160 - val_accuracy: 0.6337
Epoch 6/10
274/274 [==============================] - 224s 818ms/step - loss: 0.5862 - accuracy: 0.7942 - val_loss: 1.1677 - val_accuracy: 0.6412
Epoch 7/10
274/274 [==============================] - 225s 822ms/step - loss: 0.4494 - accuracy: 0.8426 - val_loss: 1.2359 - val_accuracy: 0.6343
Epoch 8/10
274/274 [==============================] - 223s 813ms/step - loss: 0.3341 - accuracy: 0.8826 - val_loss: 1.4352 - val_accuracy: 0.6244
Epoch 9/10
274/274 [==============================] - 223s 813ms/step - loss: 0.2212 - accuracy: 0.9221 - val_loss: 1.6103 - val_accuracy: 0.6385
Epoch 10/10
274/274 [==============================] - 225s 820ms/step - loss: 0.1483 - accuracy: 0.9489 - val_loss: 1.8815 - val_accuracy: 0.6295
```

Modelo 1

Epoch 1/10
274/274 [==============================] - 224s 814ms/step - loss: 1.7183 - accuracy: 0.3919 - val_loss: 1.4353 - val_accuracy: 0.4805
Epoch 2/10
274/274 [==============================] - 223s 813ms/step - loss: 1.2901 - accuracy: 0.5405 - val_loss: 1.2693 - val_accuracy: 0.5556
Epoch 3/10
274/274 [==============================] - 226s 826ms/step - loss: 1.0431 - accuracy: 0.6295 - val_loss: 1.1777 - val_accuracy: 0.5996
Epoch 4/10
274/274 [==============================] - 221s 806ms/step - loss: 0.8139 - accuracy: 0.7119 - val_loss: 1.1252 - val_accuracy: 0.6132
Epoch 5/10
274/274 [==============================] - 221s 807ms/step - loss: 0.5689 - accuracy: 0.8008 - val_loss: 1.3326 - val_accuracy: 0.5991
Epoch 6/10
274/274 [==============================] - 220s 805ms/step - loss: 0.3395 - accuracy: 0.8811 - val_loss: 1.5138 - val_accuracy: 0.5948
Epoch 7/10
274/274 [==============================] - 220s 804ms/step - loss: 0.1768 - accuracy: 0.9414 - val_loss: 1.9347 - val_accuracy: 0.5940
Epoch 8/10
274/274 [==============================] - 222s 809ms/step - loss: 0.1258 - accuracy: 0.9589 - val_loss: 2.3176 - val_accuracy: 0.5779
Epoch 9/10
274/274 [==============================] - 221s 807ms/step - loss: 0.0877 - accuracy: 0.9709 - val_loss: 2.5251 - val_accuracy: 0.5964
Epoch 10/10
274/274 [==============================] - 221s 805ms/step - loss: 0.0749 - accuracy: 0.9743 - val_loss: 2.8353 - val_accuracy: 0.5912

Modelo 2

Epoch 1/10
274/274 [==============================] - 350s 1s/step - loss: 2.1668 - accuracy: 0.3307 - val_loss: 113.4545 - val_accuracy: 0.1065
Epoch 2/10
274/274 [==============================] - 279s 1s/step - loss: 1.7943 - accuracy: 0.3943 - val_loss: 1.7003 - val_accuracy: 0.3700
Epoch 3/10
274/274 [==============================] - 278s 1s/step - loss: 1.4978 - accuracy: 0.4678 - val_loss: 15.4587 - val_accuracy: 0.3839
Epoch 4/10
274/274 [==============================] - 272s 995ms/step - loss: 1.2239 - accuracy: 0.5535 - val_loss: 1.2321 - val_accuracy: 0.5521
Epoch 5/10
274/274 [==============================] - 273s 998ms/step - loss: 1.1162 - accuracy: 0.5957 - val_loss: 1.1829 - val_accuracy: 0.5713
Epoch 6/10
274/274 [==============================] - 274s 1s/step - loss: 1.0004 - accuracy: 0.6382 - val_loss: 1.1842 - val_accuracy: 0.5797
Epoch 7/10
274/274 [==============================] - 293s 1s/step - loss: 0.9814 - accuracy: 0.6485 - val_loss: 1.2093 - val_accuracy: 0.5817
Epoch 8/10
274/274 [==============================] - 279s 1s/step - loss: 0.9620 - accuracy: 0.6573 - val_loss: 1.3028 - val_accuracy: 0.5840
Epoch 9/10
274/274 [==============================] - 281s 1s/step - loss: 0.9423 - accuracy: 0.6621 - val_loss: 1.3587 - val_accuracy: 0.5819
Epoch 10/10
274/274 [==============================] - 278s 1s/step - loss: 0.9285 - accuracy: 0.6655 - val_loss: 1.2672 - val_accuracy: 0.5860

Epoch 1/10
274/274 [==============================] - 332s 1s/step - loss: 2.1830 - accuracy: 0.3201 - val_loss: 2.6523 - val_accuracy: 0.3123
Epoch 2/10
274/274 [==============================] - 279s 1s/step - loss: 1.7215 - accuracy: 0.4103 - val_loss: 1.6061 - val_accuracy: 0.4125
Epoch 3/10
274/274 [==============================] - 280s 1s/step - loss: 1.5701 - accuracy: 0.4483 - val_loss: 1.4984 - val_accuracy: 0.4583
Epoch 4/10
274/274 [==============================] - 296s 1s/step - loss: 1.2664 - accuracy: 0.5403 - val_loss: 1.2424 - val_accuracy: 0.5488
Epoch 5/10
274/274 [==============================] - 297s 1s/step - loss: 1.1816 - accuracy: 0.5710 - val_loss: 1.2166 - val_accuracy: 0.5591
Epoch 6/10
274/274 [==============================] - 277s 1s/step - loss: 1.1052 - accuracy: 0.6037 - val_loss: 1.1766 - val_accuracy: 0.5675
Epoch 7/10
274/274 [==============================] - 296s 1s/step - loss: 1.0865 - accuracy: 0.6045 - val_loss: 1.1766 - val_accuracy: 0.5719
Epoch 8/10
274/274 [==============================] - 297s 1s/step - loss: 1.0802 - accuracy: 0.6099 - val_loss: 1.1733 - val_accuracy: 0.5735
Epoch 9/10
274/274 [==============================] - 280s 1s/step - loss: 1.0671 - accuracy: 0.6117 - val_loss: 1.1874 - val_accuracy: 0.5731
Epoch 10/10
274/274 [==============================] - 277s 1s/step - loss: 1.0502 - accuracy: 0.6182 - val_loss: 1.1842 - val_accuracy: 0.5781

Modelo 2

```
Epoch 1/10
274/274 [==============================] - 356s 1s/step - loss: 2.3006 - accuracy: 0.2886 - val_loss: 256.6780 - val_accuracy: 0.2067
Epoch 2/10
274/274 [==============================] - 302s 1s/step - loss: 1.8895 - accuracy: 0.3447 - val_loss: 2.1960 - val_accuracy: 0.3467
Epoch 3/10
274/274 [==============================] - 298s 1s/step - loss: 1.6157 - accuracy: 0.4143 - val_loss: 1.5341 - val_accuracy: 0.4355
Epoch 4/10
274/274 [==============================] - 285s 1s/step - loss: 1.3594 - accuracy: 0.4957 - val_loss: 1.3666 - val_accuracy: 0.5087
Epoch 5/10
274/274 [==============================] - 280s 1s/step - loss: 1.2952 - accuracy: 0.5231 - val_loss: 1.6204 - val_accuracy: 0.5180
Epoch 6/10
274/274 [==============================] - 277s 1s/step - loss: 1.2310 - accuracy: 0.5489 - val_loss: 2.3097 - val_accuracy: 0.5279
Epoch 7/10
274/274 [==============================] - 296s 1s/step - loss: 1.2125 - accuracy: 0.5577 - val_loss: 2.0799 - val_accuracy: 0.5344
Epoch 8/10
274/274 [==============================] - 280s 1s/step - loss: 1.1990 - accuracy: 0.5584 - val_loss: 2.8046 - val_accuracy: 0.5367
Epoch 9/10
274/274 [==============================] - 283s 1s/step - loss: 1.1888 - accuracy: 0.5691 - val_loss: 2.6460 - val_accuracy: 0.5381
Epoch 10/10
274/274 [==============================] - 281s 1s/step - loss: 1.1735 - accuracy: 0.5701 - val_loss: 3.1591 - val_accuracy: 0.5383
```

```
Epoch 1/10
274/274 [==============================] - 337s 1s/step - loss: 2.1126 - accuracy: 0.3327 - val_loss: 32.3894 - val_accuracy: 0.2860
Epoch 2/10
274/274 [==============================] - 301s 1s/step - loss: 1.7585 - accuracy: 0.4025 - val_loss: 3.3528 - val_accuracy: 0.3583
Epoch 3/10
274/274 [==============================] - 297s 1s/step - loss: 1.5468 - accuracy: 0.4598 - val_loss: 2.3835 - val_accuracy: 0.3945
Epoch 4/10
274/274 [==============================] - 297s 1s/step - loss: 1.3148 - accuracy: 0.5281 - val_loss: 1.2867 - val_accuracy: 0.5300
Epoch 5/10
274/274 [==============================] - 299s 1s/step - loss: 1.1790 - accuracy: 0.5726 - val_loss: 1.4993 - val_accuracy: 0.5561
Epoch 6/10
274/274 [==============================] - 295s 1s/step - loss: 1.0818 - accuracy: 0.6061 - val_loss: 1.9240 - val_accuracy: 0.5695
Epoch 7/10
274/274 [==============================] - 280s 1s/step - loss: 1.0668 - accuracy: 0.6134 - val_loss: 2.9937 - val_accuracy: 0.5719
Epoch 8/10
274/274 [==============================] - 301s 1s/step - loss: 1.0551 - accuracy: 0.6195 - val_loss: 2.5214 - val_accuracy: 0.5779
Epoch 9/10
274/274 [==============================] - 278s 1s/step - loss: 1.0488 - accuracy: 0.6215 - val_loss: 3.3527 - val_accuracy: 0.5769
Epoch 10/10
274/274 [==============================] - 280s 1s/step - loss: 1.0310 - accuracy: 0.6262 - val_loss: 4.2157 - val_accuracy: 0.5773
```

Modelo 2

Epoch 1/10
274/274 [==============================] - 335s 1s/step - loss: 2.3385 - accuracy: 0.2913 - val_loss: 3.3433 - val_accuracy: 0.1520
Epoch 2/10
274/274 [==============================] - 279s 1s/step - loss: 1.7744 - accuracy: 0.3766 - val_loss: 1.9469 - val_accuracy: 0.4059
Epoch 3/10
274/274 [==============================] - 278s 1s/step - loss: 1.6189 - accuracy: 0.4212 - val_loss: 1.5741 - val_accuracy: 0.4256
Epoch 4/10
274/274 [==============================] - 297s 1s/step - loss: 1.3689 - accuracy: 0.5032 - val_loss: 1.8000 - val_accuracy: 0.4893
Epoch 5/10
274/274 [==============================] - 301s 1s/step - loss: 1.2955 - accuracy: 0.5296 - val_loss: 2.0851 - val_accuracy: 0.5081
Epoch 6/10
274/274 [==============================] - 275s 1s/step - loss: 1.2193 - accuracy: 0.5593 - val_loss: 5.0103 - val_accuracy: 0.5201
Epoch 7/10
274/274 [==============================] - 277s 1s/step - loss: 1.2019 - accuracy: 0.5637 - val_loss: 6.7005 - val_accuracy: 0.5248
Epoch 8/10
274/274 [==============================] - 275s 1s/step - loss: 1.1869 - accuracy: 0.5685 - val_loss: 10.5695 - val_accuracy: 0.5263
Epoch 9/10
274/274 [==============================] - 277s 1s/step - loss: 1.1770 - accuracy: 0.5723 - val_loss: 10.5251 - val_accuracy: 0.5277
Epoch 10/10
274/274 [==============================] - 276s 1s/step - loss: 1.1639 - accuracy: 0.5782 - val_loss: 20.1516 - val_accuracy: 0.5297

Comparativo estatístico

Resultado

```
Pelo menos um dos grupos não segue uma distribuição normal.
Como base no teste de Kruskal-Wallis, podemos inferir que:
statistic: 57.4999
p_value: 0.0000
Portanto:
Há diferenças estatisticamente significativas entre os grupos.
```

	épocas	Melhor Acurácia
CNN	10	0.6444
Densenet121	10	0.5860
CNN	20	0.6355
Densenet121	20	**0.6721**

Implementação: https://www.kaggle.com/adrielson/comparativo-cnn-densenet-121

Referências

1. HUANG, Gao et al. Densely connected convolutional networks. In: Proceedings of the IEEE conference on computer vision and pattern recognition. 2017. p. 4700-4708.

1. BRAGA, Matheus Oliveira. Uma avaliação elaborada dos principais modelos de referência para classificação de imagens. 2021.

Programa de Pós-Graduação em Engenharia de Computação e Sistemas - PECS/UEMA

Atividade III - DenseNet-121

Aluno: Adrielson Ferreira Justino
Professor: Omar Andres Carmona Cortes