Aula 5
Transferência de Aprendizado

Eduardo L. L. Cabral

Objetivos

- Apresentar RNAs clássicas.
- Apresentar formas de transferir aprendizado de uma RNA para outra.
- Apresentar formas de utilizar uma RNA existente para criar um nova RNA.
- Apresentar formas de treinar uma nova RNA criada usando uma rede já treinada.

RNAs clássicas

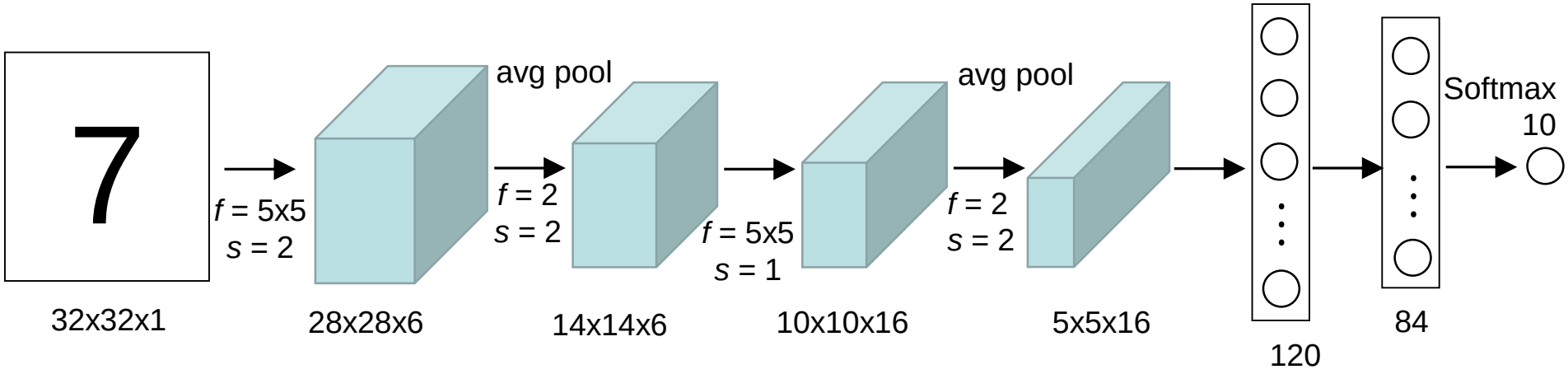
- É importante conhecer RNAs já desenvolvidas para:
  - Ganhar intuição para desenvolver novas RNAs;
  - Conhecer arquiteturas de RNAs diferentes;
  - Para usar a técnica de transferência de aprendizado.
- As RNAs clássicas são as seguintes:
  - LeNet-5;
  - AlexNet;
  - VGG16.
- Essas RNAs clássicas consistem na fundação da visão computacional moderna.
- Essas RNAs também forneceram muitas idéias importantes para outras aplicações fora da visão computacional.

RNAs clássicas – LeNet-5

- Foi desenvolvida em 1998 para reconhecer dígitos.
- Foi a primeira RNA convolucional.
- Referência ⇒ LeCun et al., Gradient-based learning applied to document recognition, 1998.
- Foi desenvolvida para reconhecer dígitos de 0 a 9 em imagens tons de cinza.
- Possui cerca de 60 mil parâmetros para imagens em tons de cinza de dimensão 32x32x1.
- Usava funções de ativação tangente hiperbólica e sigmóide.
- Originalmente usada um filtro diferente para cada canal das camadas convolucionais.

RNAs clássicas – LeNet-5

- Realizava “average-pooling”.
- Esquema da arquitetura da LeNet-5:

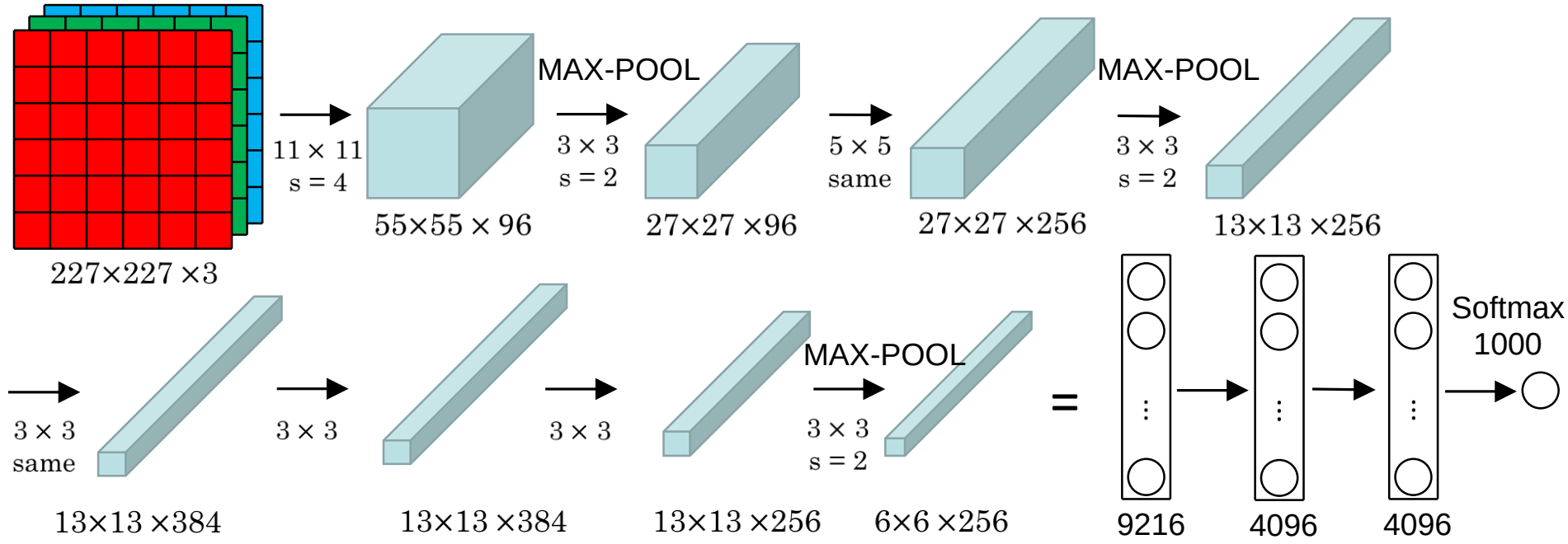


RNAs clássicas – AlexNet

- Revolucionou a área de RNAs convolucionais.
- Referência ⇒ Krizhevsky et al., ImageNet classification with deep convolutional neural networks, 2012.
- Usada para classificação de múltiplas classes com 1.000 classes ⇒ reconhecimento de objetos em imagens.
- Arquitetura similar à LeNet-5, mas muito maior.
- Possui cerca de 60 milhões de parâmetros.
- Foi a primeira RNA a usar essa quantidade de parâmetros.
- Usa função de ativação ReLu.
- Usava uma camada chamada “Local Response Normalization” que não se usa mais.

RNAs clássicas – AlexNet

- Arquitetura da AlexNet:



RNAs clássicas – VGG16

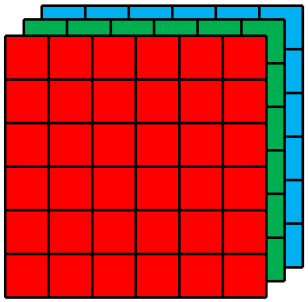
- Referência ⇒ Simonyan & Zisserman, Very deep convolutional networks for large-scale image recognition, 2015.
- Usada para classificação de múltiplas classes com 1.000 classes ⇒ reconhecimento de objetos em imagens.
- Arquitetura muito simples ⇒ composta pela repetição de camadas convolucionais simples:
  - Filtros 3x3, *s* = 1, “same convolution”;
  - Max-pooling, janela 2x2, *s* =2;
  - Duas/três camadas de convolução seguidas de uma camada de max-pooling;
  - Mantém o mesmo padrão em todas as camadas e dobra número de filtros a cada duas/três camadas convolucional.
  - Vantagem da simplicidade.

RNAs clássicas – VGG16

- Possui cerca de 138 milhões de parâmetros.
- O nome VGG16 é porque possui 16 camadas com parâmetros treináveis.
- Existe também a VGG19 $\Rightarrow$ versão maior com 19 camadas.
- VGG16 e VGG19 realizam a mesma tarefa $\Rightarrow$ melhor usar a VGG16 que é menor.

RNAs clássicas – VGG16

- Arquitetura da VGG16:



224×224 ×3

→ [CONV 64] ×2 → 224×224×64 → POOL → 112×112 ×64 → [CONV 128] ×2 → 112×112 ×128 → POOL → 56×56 ×128

→ [CONV 256] ×3 → 56×56 ×256 → POOL → 28×28 ×256 → [CONV 512] ×3 → 28×28 ×512 → POOL → 14×14×512

→ [CONV 512] ×3 → 14×14 ×512 → POOL → 7×7×512 → FC 4096 → FC 4096 → Softmax 1000

CONV = 3×3 filter, s = 1, same MAX-POOL = 2×2 , s = 2

Transferência de treinamento

- É difícil sintonizar uma RNA convolucional, ou seja, ajustar os seus hiperparâmetros, para obter um desempenho ótimo.
- Uma abordagem comum e altamente eficiente na área de RNAs é usar uma rede convolucional previamente treinada para iniciar o desenvolvimento de uma nova aplicação $\Rightarrow$ isso é usado principalmente se o banco de dados disponível for pequeno.
- As RNAs convolucionais de referência, que são usadas como base, foram treinadas usando um grande banco de dados para realizarem tarefas tipicamente de classificação de imagens.
- Se o banco de dados utilizado no treinamento da RNA convolucional de referência é realmente grande e bem geral, então, essa RNA é capaz de detectar características que podem ser efetivamente utilizadas como um modelo genérico para qualquer tipo de imagem.

Transferência de treinamento

- As características aprendidas pelas RNAs convolucionais de referência podem ser usadas para diversos problemas de visão computacional, mesmo que esses problemas sejam bem diferentes do problema original da RNA de referência.
- Por exemplo, pode-se treinar uma RNA com o banco de imagens da ImageNet, onde se tem imagens de animais e objetos comuns, e depois usar essa RNA pré-treinada para identificar itens de mobiliário.
- Essa portabilidade de treinamento é uma grande vantagem das RNAs convolucionais deep-learning comparadas com as RNAS rasas $\Rightarrow$ isso faz com que deep learning seja muito efetivo para banco de dados com poucas imagens.

Transferência de treinamento

- Existem alguns princípios que se forem seguidos a chance de sucesso no desenvolvimento de uma nova aplicação é maior:
  - Praticamente todas as RNAs desenvolvidas estão disponíveis no GitHub;
  - Para desenvolver uma nova aplicação de visão computacional usando RNAs o melhor é escolher uma RNA convolucional de referência adequada e começar o trabalho a partir dessa RNA;
  - Usar uma RNA convolucional pré-treinada como base, facilita muito o trabalho de desenvolvimento de uma nova aplicação;
  - Se for usar uma RNA já desenvolvida é melhor obter o código e os parâmetros originais dessa RNA do que tentar refazer o trabalho original, pois existem muitos detalhes a serem considerados;
  - Usar uma RNA convolucional pré-treinada evita um grande trabalho de sintonia de hiperparâmetros e de tempo de treinamento.

Transferência de treinamento

- RNAS convolucionais são compostas em geral por duas partes:
  - Uma parte inicial composta por camadas convolucionais;
  - Uma parte final composta por camadas densas e um classificador.
- Pode-se usar as camadas densas de uma RNA convolucional pré-treinada?
  - Isso deve ser evitado;
  - Parte densa foi treinada para a tarefa específica para a qual a RNA foi treinada e para imagens com uma determinada dimensão;
  - Informação contida nas camadas densas são específicas do problema;
  - A parte convolucional da RNA é treinada para detectar características que são comuns a qualquer imagem;
  - Parte convolucional de RNAs de referência consistem em extratores de características de imagens muito eficientes.

Transferência de treinamento

- Ao usar uma RNA convolucional pré-treinada deve-se antes extrair as suas camadas densas e usar somente a parte convolucional.

- O nível de generalidade e, portanto, de reusabilidade de camadas convolucionais de RNAs treinadas depende da localização da camada na rede:
  - Quanto mais no início a camada estiver mais características genéricas são detectadas, por exemplo, bordas, cores, texturas, cantos;
  - Quanto mais no final a camada estiver mais características abstratas são detectadas, por exemplo, olhos, narizes, bocas etc;
  - Se as imagens da nova tarefa diferem muito das imagens usadas no treinamento da RNA de referência, então, talvez seja melhor reusar somente as primeiras camadas.

RNAs disponíveis no Keras

- O Keras fornece muitas RNAs já treinadas que podem ser importadas facilmente do módulo keras.applications.
- Todas as RNAs convolucionais disponíveis no Keras foram treinadas para tarefas de classificação usando a base de dados ImageNet (http://www.image-net.org/).
- Algumas RNAs disponíveis atualmente no Keras são as seguintes;

- Xception	- VGG16
- VGG19	- ResNet
- ResNet v2	- ResNeXt
- Inception v3	- Inception-ResNet v2
- MobileNet v1	- MobileNet v2
- DenseNet	- NASNet

RNAs disponíveis no Keras

- Características das RNAs disponíveis no Keras.
- A exatidão se refere ao desempenho da RNA no banco de dados ImageNet.

Modelo	Tamanho	Exatidão	Parâmetros	Profundidade
Xception	88 MB	0.945	22,910,480	126
VGG16	528 MB	0.901	138,357,544	23
VGG19	549 MB	0.900	143,667,240	26
ResNet50	98 MB	0.921	25,636,712	-
ResNet101	171 MB	0.928	44,707,176	-
ResNet152	232 MB	0.931	60,419,944	-
ResNet50V2	98 MB	0.930	25,613,800	-
ResNet101V2	171 MB	0.938	44,675,560	-
ResNet152V2	232 MB	0.942	60,380,648	-
ResNeXt50	96 MB	0.938	25,097,128	-
ResNeXt101	170 MB	0.943	44,315,560	-
InceptionV3	92 MB	0.937	23,851,784	159
InceptionResNetV2	215 MB	0.953	55,873,736	572
MobileNet	16 MB	0.895	4,253,864	88
MobileNetV2	14 MB	0.901	3,538,984	88
DenseNet121	33 MB	0.923	8,062,504	121
DenseNet169	57 MB	0.932	14,307,880	169
DenseNet201	80 MB	0.936	20,242,984	201
NASNetMobile	23 MB	0.919	5,326,716	-
NASNetLarge	343 MB	0.960	88,949,818	-

Transferência de treinamento

- **Exemplo** ⇒ uso da VGG16 como RNA de referência para uma tarefa de verificar se trabalhadores em uma obra estão ou não usando capacete de proteção a partir de imagens de câmeras localizadas na obra.

  Observações:

  – VGG16 foi treinada no banco de dados ImageNet com imagens de animais e objetos do cotidiano para classificar objetos nas imagens;

  – VGG16 foi treinada inicialmente para identificar 1.000 classes de objetos em imagens de dimensão 224x224x3;

  – Como as classes da VGG16 são específicas para o problema que ela foi treinada, temos que retirar as camadas densas antes de usá-la no nosso problema.

Transferência de treinamento

- O código para carregar a VGG16 fornecida pelo Keras é o seguinte:

```
from keras.applications import VGG16
rna_base = VGG16(weights='imagenet', include_top=False,
                 input_shape=(150, 150, 3))
rna_base.summary()
```

  - São passados 3 argumentos ao carregar uma RNA:
    - `weights` ⇒ especifica os parâmetros desejados para inicializar a RNA;
    - `include_top` ⇒ refere a incluir, ou não, as camadas densas do classificador que ficam no final da RNA (essas camadas densas correspondem ao classificador das 1.000 classes da ImageNet);
    - Queremos classificar somente duas classes ⇒ se trabalhador usa ou não capacete, não incluímos essas camadas;
    - `input_shape` ⇒ é a dimensão das imagens que iremos usar; esse argumento é opcional, se não incluirmos, a RNA será capaz de processar imagens de qualquer dimensão.

Transferência de treinamento

```
Layer (type)                 Output Shape              Param #
=================================================================
input_1       (InputLayer)   (None, 150, 150, 3)   0
block1_conv1 (Convolution2D) (None, 150, 150, 64) 1792
block1_conv2 (Convolution2D) (None, 150, 150, 64) 36928
block1_pool  (MaxPooling2D)  (None, 75, 75, 64)    0
block2_conv1 (Convolution2D) (None, 75, 75, 128)  73856
block2_conv2 (Convolution2D) (None, 75, 75, 128)  147584
block2_pool  (MaxPooling2D)  (None, 37, 37, 128)  0
block3_conv1 (Convolution2D) (None, 37, 37, 256)  295168
block3_conv2 (Convolution2D) (None, 37, 37, 256)  590080
block3_conv3 (Convolution2D) (None, 37, 37, 256)  590080
block3_pool  (MaxPooling2D)  (None, 18, 18, 256)  0
block4_conv1 (Convolution2D) (None, 18, 18, 512)  1180160
block4_conv2 (Convolution2D) (None, 18, 18, 512)  2359808
block4_conv3 (Convolution2D) (None, 18, 18, 512)  2359808
block4_pool  (MaxPooling2D)  (None, 9, 9, 512)    0
block5_conv1 (Convolution2D) (None, 9, 9, 512)    2359808
block5_conv2 (Convolution2D) (None, 9, 9, 512)    2359808
block5_conv3 (Convolution2D) (None, 9, 9, 512)    2359808
block5_pool  (MaxPooling2D)  (None, 4, 4, 512)    0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
```

Transferência de treinamento

- Como escolhemos a dimensão da simagens de entrada, a dimensão do tensor de saída da rna_base fica definida, sendo (4, 4, 512) $\Rightarrow$ esse tensor será usado como entrada para o classificador do nosso exemplo.
- Existem duas formas de continuar:
  - Extração de características $\Rightarrow$ consiste em executar a rna_base convolucional com as novas imagens e guardar os resultados em um tensor numpy e usar esse tensor como dado de entrada de uma RNA densa para classificação;
  - Treinamento parcial $\Rightarrow$ consiste em estender a rna_base adicionando camadas densas no final e treinar parcialmente essa nova RNA para as novas imagens.

Extração de características

- Esse método consiste em usar a rna_base para somente extrair características das imagens da nova aplicação.
- Essas características são então processadas por uma nova RNA, que possui somente camadas do tipo densas, que é treinada do zero para realizar a nova tarefa.
- Essa solução é rápida e computacionalmente simples, porque somente exige executar a rna_base uma única vez para cada imagem.
- Somente exige o treinamento da RNA com camadas densas da nova tarefa (classificação).

Extração de características

- Após carregar a rna_base, a próxima etapa é executar essa rna_base para as imagens de treinamento, validação e teste para extrair as características dessas imagens e colocá-las em tensores.
- Fazemos isso usando o método `predict` com a rna_base:

```
train_features = rna_base.predict(train_imagens)
val_features = rna_base.predict(val_imagens)
test_features = rna_base.predict(test_imagens)
```

- As características extraídas tem dimensão (exemplos, 4, 4, 512), que é a dimensão do tensor de saída da rna_base.

Extração de características

- Como o classificador possui camadas densas, então temos que redimensionar os dados de entrada para transformá-los em um vetor.
- O código abaixo apresenta como fazer esse redimensionamento usando a função `reshape` da biblioteca numpy:

```
import numpy as np
train_features = np.reshape(train_features, (5000, 4*4*512))
val_features = np.reshape(val_features, (1000, 4*4*512))
test_features = np.reshape(test_features, (1000, 4*4*512))
```

  - Nesse código foi assumido que o número de exemplos de treinamento é 5.000 e de validação e teste é 1.000 cada um.
- Agora precisamos configurar a RNA com camadas densas para realizar a tarefa de classificação e depois compilá-la e treiná-la.

Extração de características

- A configuração, compilação e treinamento dessa RNA para classificação é feita como em casos anteriores.
- O código abaixo apresenta um exemplo de como configurar a RNA para classificação:

```
# Importa modelos e camadas
from keras import models
from keras import layers

# Configura RNA para classificação binária
rna = models.Sequential()
rna.add(layers.Dense(256, activation='relu', input_dim=4*4*512))
rna.add(layers.Dropout(0.5))
rna.add(layers.Dense(1, activation='sigmoid'))
```

  - Note que foi adicionado dropout na primeira camada densa para diminuir problemas de overfitting.

Extração de características

- O código abaixo apresenta um exemplo de como compilar e treinar a RNA para classificação:

```
# Importa otimizadores
from keras import optimizers

# Comiplação da RNA
rna.compile(optimizer=optimizers.RMSprop(lr=0.001),
            loss='binary_crossentropy', metrics=['acc'])

# Treinamento da RNA
history = rna.fit(train_features, train_labels, epochs=100,
                  batch_size=64,
                  validation_data=(val_features, val_labels))
```

  - Note que foi utilizado o método de otimização RMSprop para o treinamento da RNA.
- O treinamento dessa rede será rápido porque temos somente duas camadas densas.

Treinamento parcial

- O segundo método de realizar transferência de treinamento é mais demorado e computacionalmente mais exigente, mas os resultados em geral são melhores.
- Esse método consiste em estender a rna_base adicionando as camadas densas para classificação e treinar parcialmente a RNA resultante.
- A rna_base é adicionada como se fosse uma camada de uma RNA sequencial da mesma forma como adicionamos qualquer tipo de camada.
- Esse método permite criar uma nova RNA completa e, assim, obter resultados melhores do que somente usar a rna_base para extrair características.

Treinamento parcial

- O código abaixo apresenta um exemplo de como configurar a nova RNA tendo como camadas convolucionais iniciais a rna_base:

```
# Importa modelos e camadas
from keras import models
from keras import layers

# Define uma RNA sequencial
rna = models.Sequential()

# Inicia RNA como a rna_base e adiciona camadas de flattenning e
densas
rna.add(rna_base)
rna.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

# Visualização da arquitetura da rede
rna.summary()
```

Treinamento parcial

- A arquitetura da nova RNA será a seguinte:

```
Layer      (type)       Output Shape        Param #
=================================================================
vgg16      (Model)      (None, 4, 4, 512)   14714688
_________________________________________________________________
flatten_1 (Flatten)     (None, 8192)        0
_________________________________________________________________
dense_1    (Dense)      (None, 256)         2097408
_________________________________________________________________
dense_2    (Dense)      (None, 1)           257
=================================================================
Total params: 16,812,353
Trainable params: 16,812,353
Non-trainable params: 0
```

  - Nota-se que a nova RNA possui 16.812.353 parâmetros, sendo que 14.714.688 parâmetros são da VGG16 e 2.097.665 são das camadas densas que incluímos, ou seja, essa rede é muito grande.

Treinamento parcial

- Antes de compilar e treinar essa nova RNA é muito importante “congelar” os parâmetros da RNA convolucional usada como base (rna_base).
- “Congelar” uma camada, ou um conjunto de camadas significa impedir que os seus parâmetros sejam atualizados durante o treinamento.
- Se isso não for realizado o aprendizado da RNA convolucional usada como base será modificado durante o treinamento.
- Na medida em que as camadas densas são inicializadas aleatoriamente, erros grandes na saída da RNA serão obtidos no início do treinamento gerando gradientes de grande amplitude, que são propagados na RNA destruindo o aprendizado prévio.

Treinamento parcial

- No Keras congelamos os parâmetros de uma RNA definindo o seu atributo `trainable` igual a `False`, como mostrado no código a seguir:

```
# Número de parâmetros a serem treinados antes do congelamento
print('Número de parâmetros treináveis antes do congelamento =',
      len(model.trainable_weights))

# Congelamento dos parâmetros da rna_base
rna_base.trainable = False

# Número de parâmetros a serem treinados após o congelamento
print('Número de parâmetros treináveis após o congelamento =',
      len(model.trainable_weights))

# Impressão do resultado
Número de parâmetros treináveis antes do congelamento = 30
Número de parâmetros treináveis após o congelamento = 4
```

  - Note que os números resultantes representam o número de tensores a serem treinados e não o número total de parâmetros.

Treinamento parcial

- Com esse arranjo somente os parâmetros das duas camadas densas adicionadas no final da rna_base são treinados.
- O numero de parâmetros treinados são 4 tensores $\Rightarrow$ dois tensores por camada (matriz de pesos das ligações e vetor de vieses).
- A compilação e treinamento dessa nova RNA devem ser realizados após o congelamento dos parâmetros e é feita normalmente como nos casos já vistos.
- Seguindo esse procedimento o resultado que se pode alcançar é o mesmo que o obtido pelo método de extração de características, pois a RNA base convolucional é a mesma e, portanto, as características utilizadas pela parte densa da rede para realizar a classificação são as mesmas.

Treinamento parcial com sintonia fina

- Para obter melhores resultados quando se usa uma RNA pré-treinada é estender o treinamento parcial com uma “sintonia fina”.
- A técnica de realizar sintonia fina consiste em “descongelar” algumas camadas finais da rna_base e re-treinar essas camadas junto com as camadas densas adicionadas.
- Essa técnica se chama sintonia fina porque realiza pequenos ajustes nos parâmetros das camadas finas da RNA convolucional base.
- Lembre que as camadas convolucionais finais extraem representações abstratas, que estão mais relacionadas com a tarefa realizada $\Rightarrow$ a sintonia fina ajusta essas representações para o novo problema.

Treinamento parcial com sintonia fina

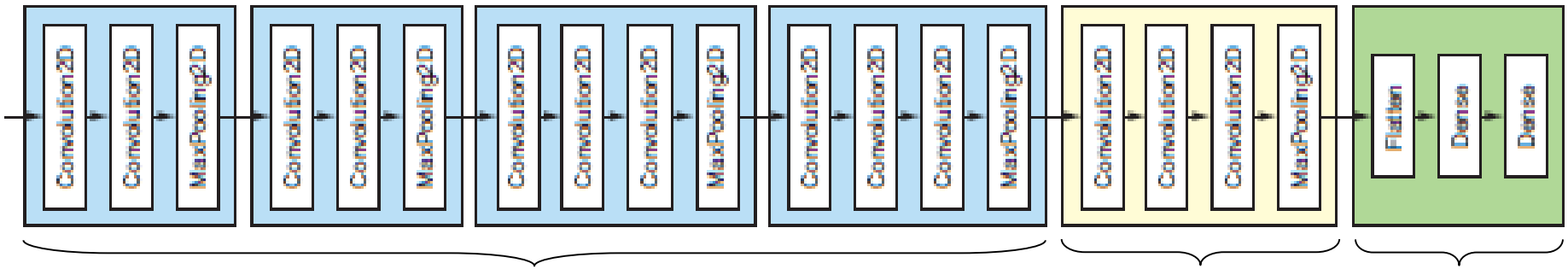
- Como visto, é necessário congelar a RNA convolucional base para treinar uma RNA de classificação adicionada à essa.
- Mas somente é possível melhorar as últimas camadas da RNA convolucional base depois que a RNA de classificação já foi treinada.
- Se o classificador não foi treinado o erro propagado através da rna_base durante o treinamento será muito grande e o aprendizado da RNA convolucional será destruído.
- Existe um procedimento para realizar a sintonia fina da nova RNA e, assim, melhorar o treinamento.

Treinamento parcial com sintonia fina

- As etapas para realizar a sintonia fina de uma RNA são as seguintes:
  1. Adicionar a RNA de classificação no final da RNA convolucional base;
  2. Congelar os parâmetros da RNA base;
  3. Treinar a RNA de classificação;
  4. Descongelar algumas camadas da RNA convolucional base;
  5. Re-treinar em conjunto as camadas convolucionais descongeladas da RNA base e as camadas densas adicionadas.
- As etapas 1 a 3 são as mesmas utilizadas na técnica de treinamento parcial, vista anteriormente.

Treinamento parcial com sintonia fina

- Com exemplo da etapa 4, vamos descongelar toda a rna_base e depois congelar algumas camadas.
- Para facilitar a visualização do que será realizado, um esquema da VGG16 com as camadas densas adicionadas é mostrado abaixo:



Treinamento parcial com sintonia fina

- Nesse exemplo iremos realizar a sintonia fina das últimas 3 camadas convolucionais, o que significa que todas as camadas até o 4º bloco (blocos em azul) são congeladas e as camadas do 5º bloco (bloco em amarelo) serão re-treinadas.
- Uma pergunta que pode surgir $\Rightarrow$ Porque não realizar a sintonia fina de mais camadas, ou mesmo de toda a RNA convolucional base?

  É claro que podemos re-treinar tudo que quisermos, mas devemos considerar o seguinte:
  - Camadas iniciais de RNA convolucionais profundas codificam características genéricas presentes em qualquer imagem e, portanto, essas camadas são reutilizáveis;
  - Camadas convolucionais finais codificam características mais especializadas do problema específico para o qual foi treinada;

Treinamento parcial com sintonia fina

- É mais útil re-treinar somente as camadas convolucionias mais especializadas, porque essas são as camadas que devem ser ajustadas para o novo problema;
- Não haveria nenhum ganho, ou quase nenhum, em re-treinar as camadas iniciais da RNA base;
- Quanto mais parâmetros forem treinados maiores os problemas de overfitting;
- Nesse exemplo, a RNA convolucional base tem cerca de 14,7 milhões de parâmetros $\Rightarrow$ seria um problema enorme treinar uma RNA tão grande, principalmente se tivermos poucos exemplos.

➢ Sempre a melhor estratégia é realizar a sintonia fina somente das camadas convolucionais finais da RNA base.

Treinamento parcial com sintonia fina

- O código abaixo apresenta como descongelar as camadas que queremos ajustar da RNA convolucional base. Note que para isso precisamos saber os “nomes” das várias camadas da RNA.
- A arquitetura da rna_base está representada no próximo slide para facilitar a compreensão.

```
# Descongela todas as camadas da rna_base
rna_base.trainable = True
set_trainable = False

# Percorre camadas da rna_base procurando pelo 5° bloco
for layer in rna_base.layers:
    if layer.name == 'block5_conv1':
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False
```

Treinamento parcial com sintonia fina

```
Layer         (type)              Output Shape          Param #
=================================================================
input_1       (InputLayer)        (None, 150, 150, 3)   0
block1_conv1  (Convolution2D)     (None, 150, 150, 64)  1792
block1_conv2  (Convolution2D)     (None, 150, 150, 64)  36928
block1_pool   (MaxPooling2D)      (None, 75, 75, 64)    0
block2_conv1  (Convolution2D)     (None, 75, 75, 128)   73856
block2_conv2  (Convolution2D)     (None, 75, 75, 128)   147584
block2_pool   (MaxPooling2D)      (None, 37, 37, 128)   0
block3_conv1  (Convolution2D)     (None, 37, 37, 256)   295168
block3_conv2  (Convolution2D)     (None, 37, 37, 256)   590080
block3_conv3  (Convolution2D)     (None, 37, 37, 256)   590080
block3_pool   (MaxPooling2D)      (None, 18, 18, 256)   0
block4_conv1  (Convolution2D)     (None, 18, 18, 512)   1180160
block4_conv2  (Convolution2D)     (None, 18, 18, 512)   2359808
block4_conv3  (Convolution2D)     (None, 18, 18, 512)   2359808
block4_pool   (MaxPooling2D)      (None, 9, 9, 512)     0
block5_conv1  (Convolution2D)     (None, 9, 9, 512)     2359808
block5_conv2  (Convolution2D)     (None, 9, 9, 512)     2359808
block5_conv3  (Convolution2D)     (None, 9, 9, 512)     2359808
block5_pool   (MaxPooling2D)      (None, 4, 4, 512)     0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
```

Treinamento parcial com sintonia fina

- Agora a RNA está pronta para ser re-treinada e sintonizada para o novo problema.
- A sintonia fina deve ser realizada com uma taxa de aprendizado muito pequena, porque se deseja limitar o valor das modificações das camadas convolucionais que estão sendo ajustadas ⇒ atualizações muito grande dos parâmetros podem destruir completamente o treinamento original.

```
# Importa modelos e camadas
rna.compile(loss='binary_crossentropy',
            optimizer=optimizers.RMSprop(lr=1e-5)
            metrics=['acc'])
history = rna.fit(train_imagens,
                  train_labels, epochs=100,
                  batch_size=64,
                  validation_data=(val_imagens, val_labels))
```

Treinamento parcial com sintonia fina

Conclusão $\Rightarrow$ transferência de treinamento realizada com a técnica de sintonia fina é extremamente eficiente para obter sistemas com alto desempenho, mesmo quando o conjunto de dados é pequeno.