

[PSI5790 aula 9, Início]
Segmentação semântica

1. Introdução

Segmentação semântica é a tarefa de classificar cada um dos pixels de uma imagem em classes. Veja o exemplo na figura 1. Segmentação é diferente de classificação, pois segmentação atribui uma classe para cada pixel da imagem, enquanto que classificação atribui uma única classe para a imagem inteira. Isto é, a saída da segmentação é uma imagem, enquanto que na classificação é um número.

Nota: Na apostila aprendizagem-ead, foram apresentadas algumas técnicas clássicas de aprendizado de máquina que podem ser usadas para segmentar imagens. Segmentamos feijões e mapas pela textura. Seria interessante resolver novamente aquelas aplicações usando aprendizado profundo.

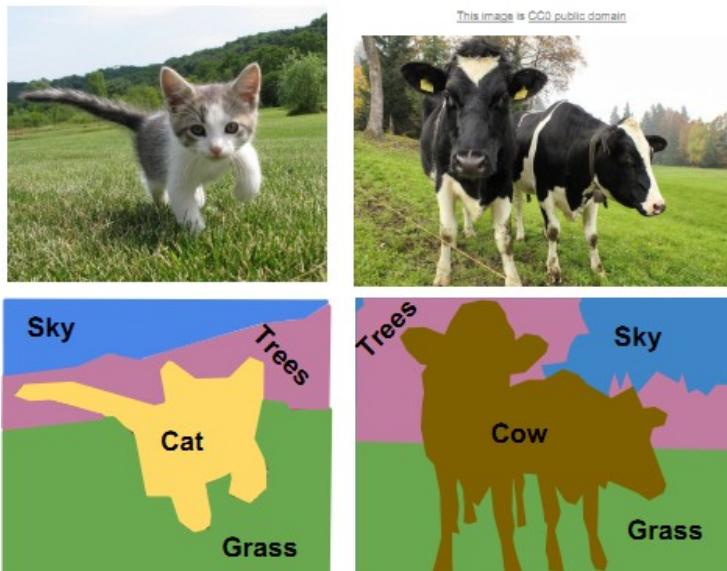


Figura 1: Segmentação semântica (de [Li2017-lecture11]).

Existem outros problemas, além da segmentação semântica, onde a entrada é uma imagem e a saída também é uma imagem. Um deles é a colorização, onde a entrada é uma imagem em níveis de cinzas e a saída é uma imagem colorida [Zeger2021]. O EP de 2022 desta disciplina era sobre colorir faces (figura F1).

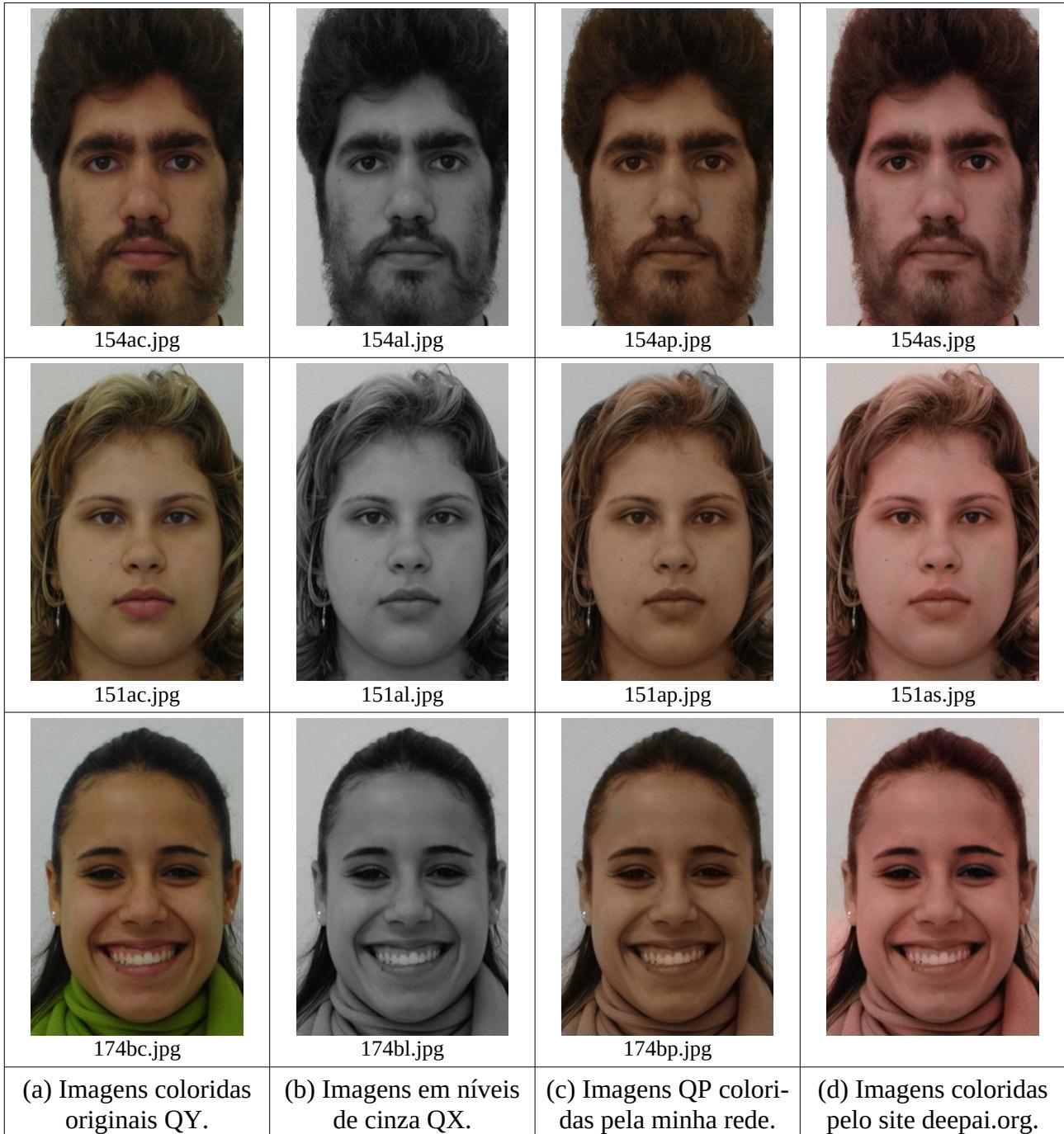


Figura F1: (a) Imagens coloridas originais; (b) conversão para níveis de cinza; (c) imagens coloridas pela rede; e (d) imagens coloridas pelo site deepai.org/machine-learning-model/colorizer.

Um outro problema onde tanto a entrada e a saída são imagens é a super-resolução. Super-resolução (ou aumento de resolução) recebe uma imagem em baixa resolução e gera uma imagem em alta-resolução. O EP de 2018 desta disciplina pedia para aumentar a resolução das imagens de faces (figura F2). Super-resolução é usada hoje em dia em televisores modernos para gerar vídeos de altíssima resolução (4k ou 8k) a partir de vídeos de qualidade inferior.



Figura F2: Da esquerda para direta: Imagem original 8×8 ; interpolação bilinear para 32×32 ; interpolação bicúbica para 32×32 ; super-resolução com rede neural convolucional; saída ideal 32×32 .

As arquiteturas de redes que veremos nesta apostila podem ser utilizadas para colorir e aumentar a resolução das imagens, embora seja possível obter resultados melhores utilizando as arquiteturas de redes especialmente projetadas para essas tarefas.

1.1 Elipses e retângulos ruidosos

Para testar as técnicas de segmentação semântica, sem perder muito tempo treinando redes neurais, criei um pequeno conjunto de imagens denominado de *eliret*. Nesse conjunto, há imagens de elipses e retângulos ruidosos com tamanhos, formas e nível de cinza variados (figura 2), sempre centralizadas no meio da imagem. Todas as imagens têm dimensão 32×32 e vem em pares (X, Y): imagem de entrada e respectiva saída com a segmentação ideal.

Há 100 imagens de elipses (000 a 099) e 100 de retângulos (100 a 199) com as respectivas saídas. Esses 200 pares de imagens estão divididos em 100 pares para treino (listadas no arquivo *treino.csv*), 50 para validação (*valida.csv*) e 50 para teste (*teste.csv*). O problema é treinar uma rede com imagens *treino.csv* e *valida.csv* de forma que, processando as imagens de entrada de *teste.csv* (QX), obter saídas QP mais parecidas possível com as segmentações ideais QY . Esse BD está em:

http://www.lps.usp.br/hae/apostila/segm_eliret.zip (compactado) OU

<https://drive.google.com/drive/folders/1qtOVdWNaxnro-SDUBpjFJFB-PcPop7t?usp=sharing> (não compactado)

Nota: O problema acabou ficando fácil demais para ser resolvido por aprendizagem profunda. Deveria ter criado um problema um pouco mais difícil. Mesmo assim, podemos tirar algumas conclusões deste exemplo.

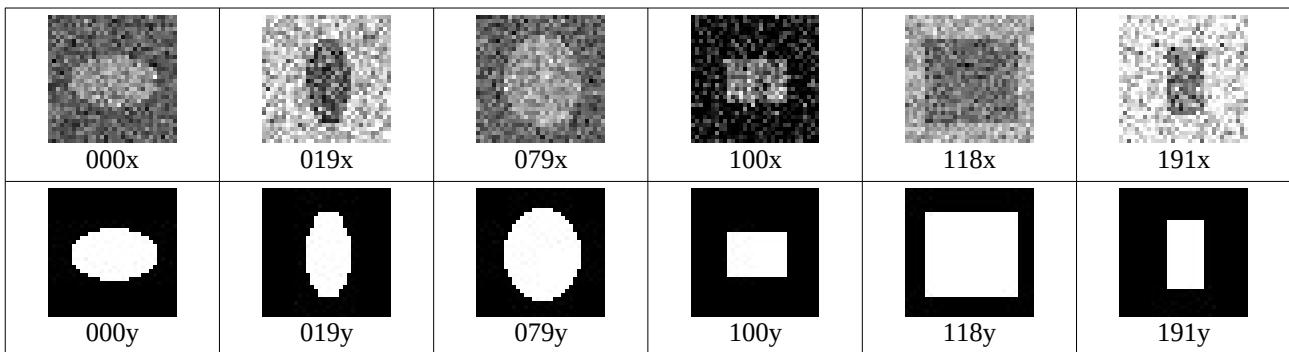


Figura 2: Banco de imagens *eliret* para testar técnicas de segmentação semântica.

Copio abaixo algumas linhas dos 3 arquivos .csv:

Treino.csv:

000x.png;000y.png
100x.png;100y.png
...
049x.png;049y.png
149x.png;149y.png

Valida.csv:

050x.png;050y.png
150x.png;150y.png
...
074x.png;074y.png
174x.png;174y.png

teste.csv:

075x.png;075y.png
175x.png;175y.png
...
099x.png;099y.png
199x.png;199y.png

Os arquivos de elipses estão numerados de 000 a 099 e os arquivos de retângulos de 100 a 199. Os arquivos de treino vão de 000 a 049 e 100 a 149; os arquivos de validação vão de 050 a 074 e 150 a 174; os arquivos de teste vão de 075 a 099 e 175 a 199. Estou supondo que os arquivos .csv e as imagens ficam no diretório default.

1.2 Fully Convolutional Network (FCN)

Fully convolutional network (FCN) é uma rede neural convolucional com duas partes: *downsampling* e *upsampling* (figura 3).

https://pt.d2l.ai/chapter_computer-vision/fcn.html

https://d2l.ai/chapter_computer-vision/fcn.html

<https://paperswithcode.com/paper/fully-convolutional-networks-for-segmentation>

A parte de *downsampling* é semelhante à rede “tipo LeNet” que já usamos para classificar os dígitos de MNIST: consiste de camadas convolucionais com função de ativação (normalmente ReLU) seguidas por subamostragem (normalmente max-pooling). Como vimos, na parte final da rede LeNet há camadas densas (fully connected) que classificam a imagem. Aqui, em vez de camadas densas, vamos colocar camadas que aumentam a resolução da imagem (*upsampling*), até atingir a resolução da imagem original.

Qual é a intuição de fazer *downsampling* seguida de *upsampling* para fazer segmentação?

A primeira observação é que a classe de pixel não fica mudando rapidamente. Não é possível um pixel ser “gramo”, o pixel seguinte ser “boi” e o pixel seguinte ser “céu”. A classe é constante dentro de uma certa região da imagem. Assim, a imagem em baixa resolução é adequada para representar a classe de uma região da imagem.

A segunda observação é que, para decidir a classe de um pixel, é necessário olhar uma vizinhança relativamente grande em torno do pixel. Um pixel verde pode tanto ser grama como copa de árvore - não é possível decidir a classificação sem olhar a vizinhança. Cada pixel nas imagens de baixa resolução “enxerga” uma vizinhança grande na imagem de entrada, permitindo classificar a classe do pixel. A parte de *upsampling* tenta extrapolar a classificação presente em baixa resolução para alta resolução.

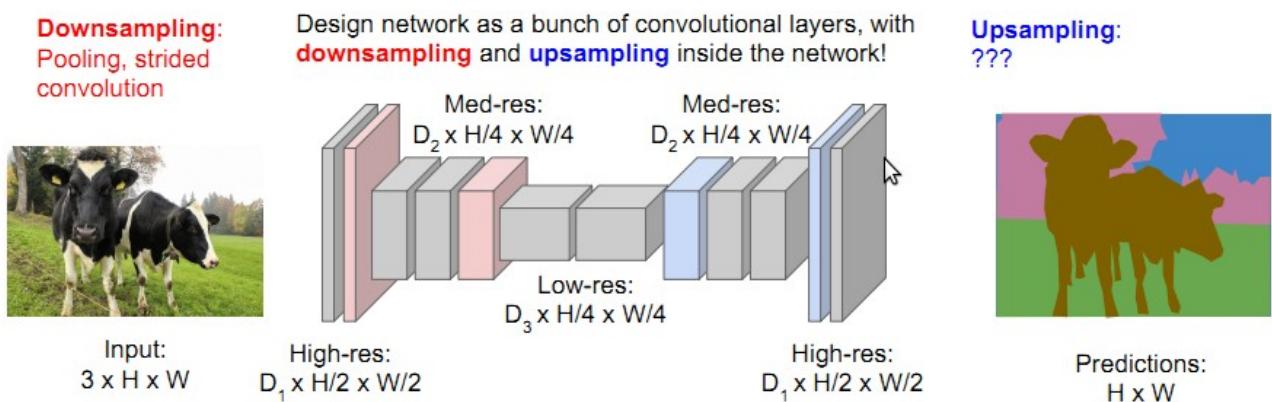


Figura 3: Estrutura de uma rede FCN para segmentação semântica (de [Li2017-lecture11]).

Para fazer a segmentação semântica, devemos fazer *upsampling* e para isso precisamos de camadas que aumentam resolução. As camadas que vimos até agora somente conseguem manter ou diminuir a resolução. Há (pelo menos) duas formas de aumentar resolução:

1.3 Unpooling: o “contrário” de max-pooling ou average-pooling.

Em Keras, há camada que faz o contrário de *pooling*, denominada *UpSampling2D* (figura 4). Se usar o bloco 2×2 (por exemplo), *UpSampling2D* replica o valor de cada pixel 4 vezes. Evidentemente, é possível usar outros tamanhos de bloco, como 3×3 . Também é possível escolher diferentes tipos de interpolação diferentes de “nearest”, entre eles “area”, “bicubic”, “bilinear”, etc.

Classe UpSampling2D:

```
tf.keras.layers.UpSampling2D(  
    size=(2, 2), data_format=None, interpolation="nearest", **kwargs  
)
```

https://keras.io/api/layers/reshaping_layers/up_sampling2d/

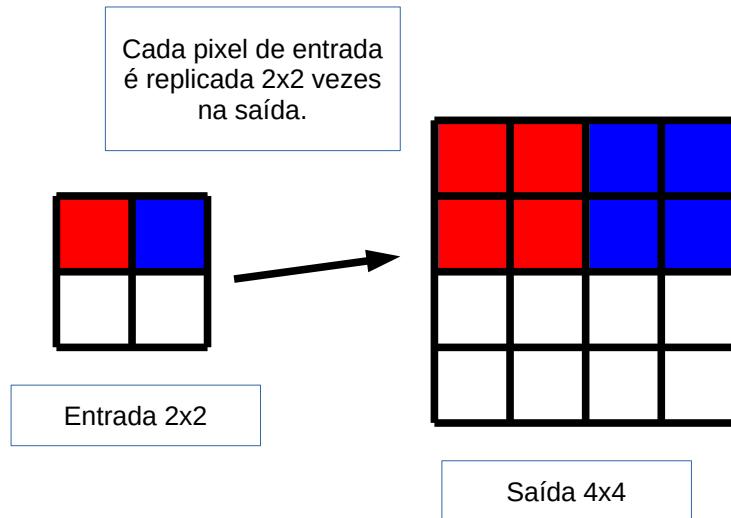


Figura 4: UpSampling2D de Keras.

1.4 Convolução transposta (ou deconvolução): o “contrário” da convolução.

Camada convolucional que vimos até agora normalmente anda com passo (strides) 1. Isto é, a janela move de 1 em 1 pixel e a saída terá o mesmo tamanho que a entrada (usando modo “same”). Aumentando o passo para 2, a camada convolucional irá reduzir a resolução da imagem por 2 (modo “same”), pois janela irá mover de 2 em 2 pixels. Concluindo, é possível usar camada convolucional para diminuir a resolução da imagem, bastando para isso usar passo maior que 1.

Na convolução transposta (transposed convolution ou deconvolution, figuras 5 e 6), a janela percorre imagem de saída em vez de percorrer a imagem de entrada. Se executar convolução transposta com passo maior que 1, a resolução da imagem de saída ficará maior que a entrada (modo “same”).

Tanto a convolução como a convolução transposta podem ser feitas no modo “valid” (que só considera os pixels onde a janela cabe inteiramente dentro do domínio da imagem) ou “same” (que supõe que os pixels fora do domínio da imagem são zeros). No modo “same” com passo 2, convolução e convolução transposta geram saídas respectivamente 2 vezes menor e 2 vezes maior que a imagem de entrada.

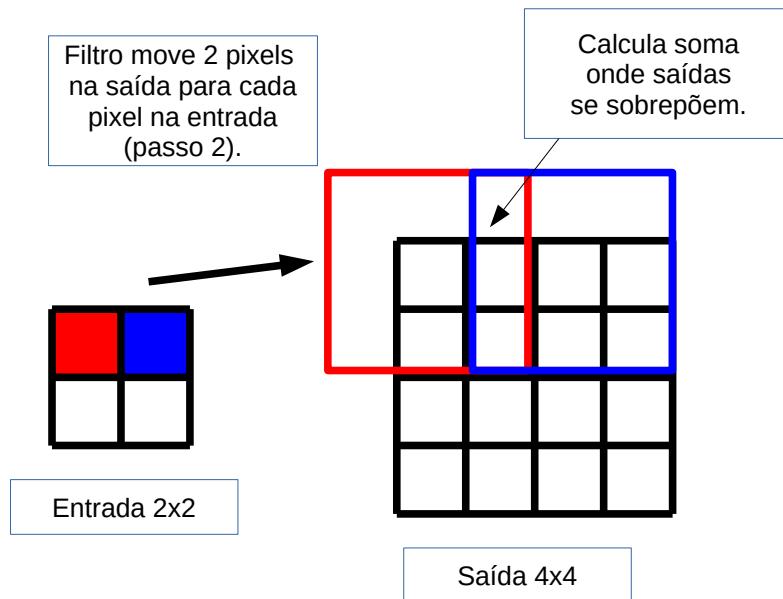


Figura 5: Convolução transposta 2D com janela 3×3 , passo 2.

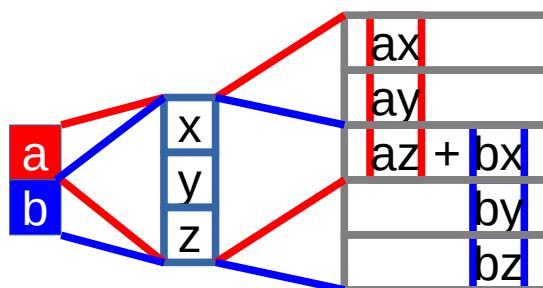


Figura 6: Convolução transposta 1D com janela de dimensão 3, passo 2.

Exercício: Faça um pequeno programa em Keras para testar as saídas de convolução e convolução transposta.

1.5 Implementação de FCN para segmentar elipses e retângulos

A célula de código abaixo baixa BD segm_eliret.zip e o descompacta no diretório default:

```

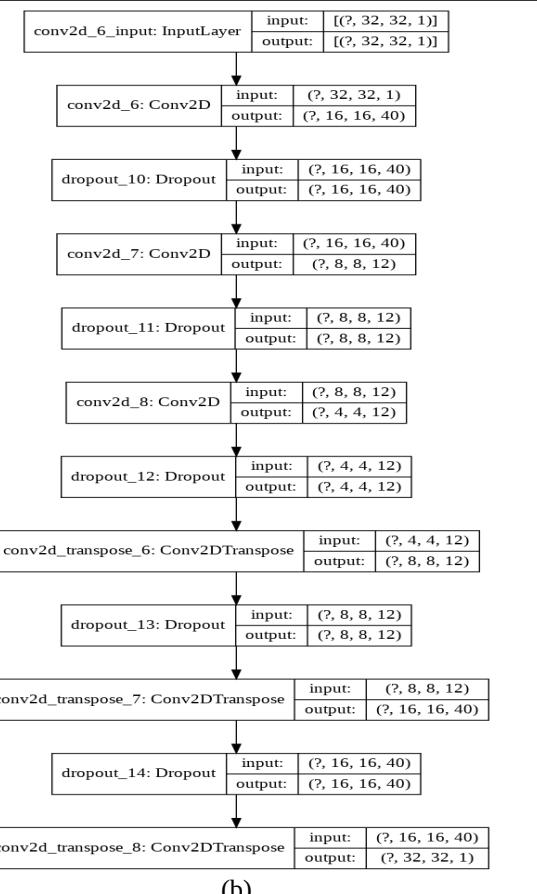
1 url='http://www.lps.usp.br/hae/apostila/segm_eliret.zip'
2 import os; nomeArq=os.path.split(url)[1]
3 if not os.path.exists(nomeArq):
4     print("Baixando o arquivo",nomeArq,"para diretorio default",os.getcwd())
5     os.system("wget -nc -U 'Firefox/50.0' "+url)
6 else:
7     print("O arquivo",nomeArq,"ja existe no diretorio default",os.getcwd())
8 print("Descompactando arquivos novos de",nomeArq)
9 os.system("unzip -u "+nomeArq)

```

O programa fcn-train1 (programa 1) treina FCN para segmentar elipses e retângulos ruidosos. A estrutura da rede está na figura 7 e os erros MSE obtidos ao longo do treino na figura 8.

| Layer (type) | Output Shape | Param # |
|-------------------------------|--------------------|---------|
| <hr/> | | |
| conv2d_6 (Conv2D) | (None, 16, 16, 40) | 1040 |
| dropout_10 (Dropout) | (None, 16, 16, 40) | 0 |
| conv2d_7 (Conv2D) | (None, 8, 8, 12) | 12012 |
| dropout_11 (Dropout) | (None, 8, 8, 12) | 0 |
| conv2d_8 (Conv2D) | (None, 4, 4, 12) | 3612 |
| dropout_12 (Dropout) | (None, 4, 4, 12) | 0 |
| conv2d_transpose_6 (Conv2DTr) | (None, 8, 8, 12) | 3612 |
| dropout_13 (Dropout) | (None, 8, 8, 12) | 0 |
| conv2d_transpose_7 (Conv2DTr) | (None, 16, 16, 40) | 12040 |
| dropout_14 (Dropout) | (None, 16, 16, 40) | 0 |
| conv2d_transpose_8 (Conv2DTr) | (None, 32, 32, 1) | 1001 |
| <hr/> | | |
| Total params: 33,317 | | |
| Trainable params: 33,317 | | |
| Non-trainable params: 0 | | |

(a)



(b)

Figura 7: Estrutura da rede FCN para segmentar elipses e retângulos.

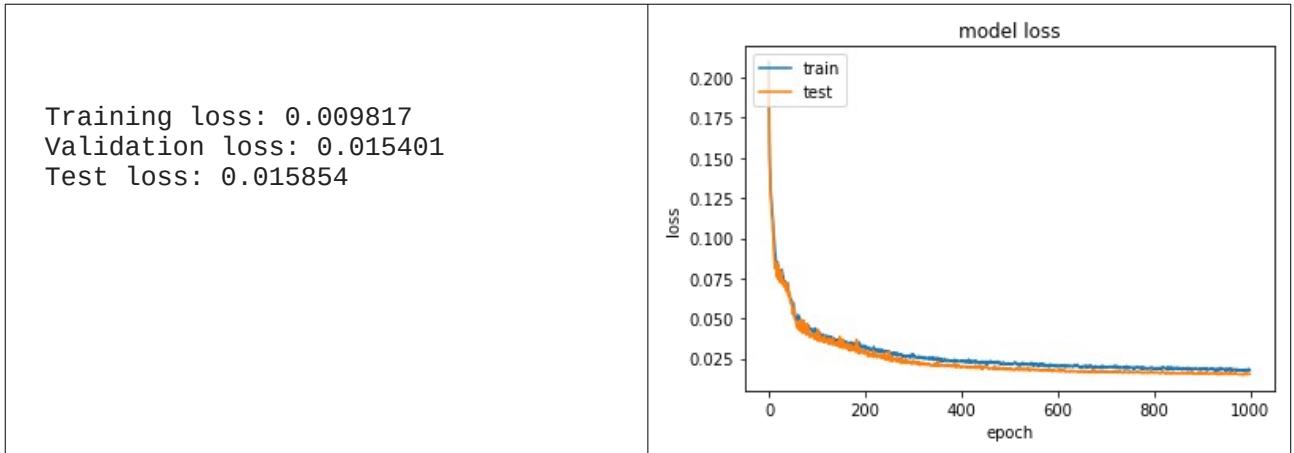


Figura 8: Erros MSE durante treino e teste.

No programa 1, as linhas 19-34 definem função a *leCsv* que lê arquivo csv (comma separated values) e lê as pares de imagens entrada-saída listadas dentro do arquivo csv. As linhas 37-40 usam essa função para ler as imagens do BD e as armazena nas matrizes de treino, validação e teste *ax*, *ay*, *vx*, *vy*, *qx* e *qy* como tensores “float32” de tamanhos $n \times 32 \times 32 \times 1$, onde n é o número de imagens. Os elementos vão de 0 a 1.

As linhas 45-59 definem a estrutura da rede. As linhas 46-52 fazem downsampling. Note as opções *strides=2* e *padding=“same”*. *Strides=2* faz a janela do filtro andar de dois em dois pixels. *Padding=“same”* faz a janela do filtro percorrer todos os pixels da imagem de entrada, mesmo que a janela fique parcialmente fora do domínio da imagem (assume valor zero para os pixels fora do domínio). Estas duas opções fazem com que a saída da camada convolucional tenha resolução duas vezes menor do que a entrada, sem a necessidade de fazer max-pooling.

As linhas 53-57 fazem upsampling. As camadas Conv2DTranspose com opções *strides=2* e *padding=“same”* fazem com que a resolução da imagem de saída seja duas vezes maior que a entrada.

Todas funções de ativação são “relu” exceto a última camada, que não possui função de ativação. As camadas Dropout foram inseridas para diminuir overfitting.

Veja a figura 9, onde as colunas *a* e *b* mostram imagens de entrada e saída ideais e as colunas *c* e *d* (amarelo) mostram as saídas de FCN. A saída original de FCN é imagem em níveis de cinza (coluna *c*). Ela foi limiarizada (coluna *d*) para facilitar a comparação com a saída ideal. Pelas saídas, pode-se afirmar que a rede FCN não aprendeu a distinguir se a entrada é uma elipse ou um retângulo. A saída de elipse ficou retangular e a saída de retângulo ficou elipsoidal. Em FCN, uma camada que aumenta resolução só possui informações fornecidas pela camada anterior de resolução duas vezes mais baixa. Uma camada que aumenta resolução trabalha sem saber se a imagem que está tentando segmentar é elipse ou retângulo, o que faz com que elipse fique retangular e retângulo fique elipsoidal.

```

1 #fcn-train1.py
2 #Treina rede fcn para segmentacao semanticas de elipses e retangulos
3 import os; os.environ['TF_CPP_MIN_LOG_LEVEL']=3
4 os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
5 import tensorflow.keras as keras
6 from tensorflow.keras.models import Sequential
7 from tensorflow.keras.layers import Dropout, Conv2D, Conv2DTranspose
8 from tensorflow.keras import optimizers
9 import sys; import cv2; import numpy as np; import matplotlib.pyplot as plt
10
11 def impHistoria(history):
12     print(history.history.keys())
13     plt.plot(history.history['loss'])
14     plt.plot(history.history['val_loss'])
15     plt.title('model loss'); plt.ylabel('loss'); plt.xlabel('epoch')
16     plt.legend(['train', 'test'], loc='upper left')
17     plt.show()
18
19 def leCsv(nomeDir,nomeArq):
20     print("Lendo: ", nomeArq); arq=open(os.path.join(nomeDir,nomeArq),"r")
21     lines=arq.readlines(); arq.close(); n=len(lines)
22
23     nl,nc = 32,32;
24     AX=np.empty((n,nl,nc),dtype='uint8'); AY=np.empty((n,nl,nc),dtype='uint8')
25     i=0;
26     for linha in lines:
27         linha=linha.strip('\n'); linha=linha.split(';')
28         AX[i]=cv2.imread(os.path.join(nomeDir,linha[0]),0)
29         AY[i]=cv2.imread(os.path.join(nomeDir,linha[1]),0)
30         i=i+1
31
32     ax= np.float32(AX)/255.0; ay= np.float32(AY)/255.0 #Entre 0 e 1
33     ax = ax.reshape(n, nl, nc, 1); ay = ay.reshape(n, nl, nc, 1)
34     return ax, ay
35
36 #<<<<<<<<< main <<<<<<<<<
37 bdDir = "."
38 ax, ay = leCsv(bdDir,"treino.csv")
39 vx, vy = leCsv(bdDir,"valida.csv")
40 qx, qy = leCsv(bdDir,"teste.csv")
41 outDir = "."; os.chdir(outDir)
42
43 nl,nc = 32,32; input_shape = (nl,nc,1); batch_size = 20; epochs = 1000
44
45 model = Sequential()
46 model.add(Conv2D(40, kernel_size=(5,5), strides=2, activation='relu',
47                  padding='same', input_shape=input_shape)) #saida 16*16*40
48 model.add(Dropout(0.25))
49 model.add(Conv2D(12, kernel_size=(5,5), strides=2, activation='relu', padding='same')) #saida 8*8*12
50 model.add(Dropout(0.25))
51 model.add(Conv2D(12, kernel_size=(5,5), strides=2, activation='relu', padding='same')) #saida 4*4*12
52 model.add(Dropout(0.25))
53 model.add(Conv2DTranspose(12, kernel_size=(5,5), strides=2, activation='relu', padding='same')) #saida 8*8*12
54 model.add(Dropout(0.25))
55 model.add(Conv2DTranspose(40, kernel_size=(5,5), strides=2, activation='relu', padding='same')) #saida 16*16*40
56 model.add(Dropout(0.25))
57 model.add(Conv2DTranspose(1, kernel_size=(5,5), strides=2, padding='same')) #saida 32*32*1
58 from tensorflow.keras.utils import plot_model; plot_model(model, to_file='fcn-train1.png', show_shapes=True)
59 model.summary()
60
61 opt=optimizers.Adam()
62 model.compile(optimizer=opt, loss='mean_squared_error')
63 history=model.fit(ax, ay, batch_size=batch_size, epochs=epochs, verbose=0, validation_data=(vx,vy))
64 impHistoria(history)
65
66 score = model.evaluate(ax, ay, verbose=0); print('Training loss:', score)
67 score = model.evaluate(vx, vy, verbose=0); print('Validation loss:', score)
68 score = model.evaluate(qx, qy, verbose=0); print('Test loss:', score)
69 model.save('fcn-train1.keras')

```

Programa 1: Treina modelo para segmentar elipses e retângulos usando FCN (sem data augmentation). https://colab.research.google.com/drive/1RUc1fE1daB1zBox7pe8-of2os_wKHqCv?usp=sharing

| | a | b | c | d | e | f |
|-----|------------------|----------------------|---------------------------|----------------------|----------------------------|-----------------------|
| | entrada teste QX | saída teste ideal QY | saída FCN níveis de cinza | saída FCN binarizada | saída UNET níveis de cinza | saída UNET binarizada |
| 076 | | | | | | |
| 078 | | | | | | |
| 176 | | | | | | |
| 178 | | | | | | |

Figura 9: Algumas imagens de teste segmentadas pela FCN (amarelo) e U-net (azul).

O programa fcn-pred1.py (programa 2) usa a rede gerada pelo programa fcn-train1.py (programa 1) para segmentar a imagem de elipse ou retângulo ruidosa.

A variável “*nome*” especifica a imagem que se deseja segmentar (“*nomex.png*”).

```

1 #fcn-pred1.py
2 #Faz segmentacao semanticas usando rede gerada pelo fcn-train1.py
3 import os; os.environ['TF_CPP_MIN_LOG_LEVEL']=3'
4 os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
5 import cv2; import numpy as np
6 import tensorflow.keras as keras
7 from tensorflow.keras.models import load_model
8 from tensorflow.keras.layers import Dropout, Conv2D, Conv2DTranspose
9 from tensorflow.keras import optimizers
10 import sys; from sys import argv
11
12 #<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
13 bdDir = "."
14 outDir = "."; os.chdir(outDir)
15 nome="077"
16 inImgX = nome+"x.png"; inImgY = nome+"y.png"
17 outImgG = nome+"g.png"; outImgB = nome+"b.png"
18 arquivoRede = "fcn-train1.keras"
19
20 model = load_model(os.path.join(outDir,arquivoRede))
21 QX=cv2.imread(os.path.join(bdDir,inImgX),0)
22 QY=cv2.imread(os.path.join(bdDir,inImgY),0)
23 nl=QX.shape[0]; nc=QX.shape[1]
24 qx=np.float32(QX)/255.0 #Entre 0 e +1
25 qx=qx.reshape(1, nl, nc, 1)
26
27 qp=model.predict(qx); qp=qp.reshape(nl,nc) # entre 0 e +1
28
29 QPG=255.0*qp; QPG=np.clip(QPG,0,255) # Entre 0 e 255
30 QPG=np.uint8(QPG); cv2.imwrite(os.path.join(outDir,outImgG),QPG)
31
32 QPB=np.zeros((nl,nc),dtype='uint8'); QPB[ qp>=0.5 ] = 255
33 cv2.imwrite(os.path.join(outDir,outImgB),QPB)
34
35 from matplotlib import pyplot as plt
36 f = plt.figure()
37 f.add_subplot(1,4,1); plt.imshow(QX,cmap="gray"); plt.axis('off')
38 f.add_subplot(1,4,2); plt.imshow(QY,cmap="gray"); plt.axis('off')
39 f.add_subplot(1,4,3); plt.imshow(QPG,cmap="gray"); plt.axis('off')
40 f.add_subplot(1,4,4); plt.imshow(QPB,cmap="gray"); plt.axis('off')
41 plt.show(block=True)

```

Programa 2: Segmenta elipse/retângulo usando modelo FCN treinada previamente (sem data augmentation).

https://colab.research.google.com/drive/1RUc1fE1daB1zBox7pe8-of2os_wKHqCv?usp=sharing

2. Redes completamente convolucionais

2.1 Segmentar imagem de teste maior do que imagens de treino

Vimos (apostila cifar-ead) que as redes convolucionais “modernas” conseguem classificar imagens maiores que as imagens de treino, sem ter que redimensionar as imagens de entrada. De fato, as redes neurais convolucionais “completamente convolucionais” (isto é, que não possuem camadas densas) podem ser alimentadas com imagens maiores do que a resolução para a qual foi treinada – tanto para segmentação quanto para classificação.

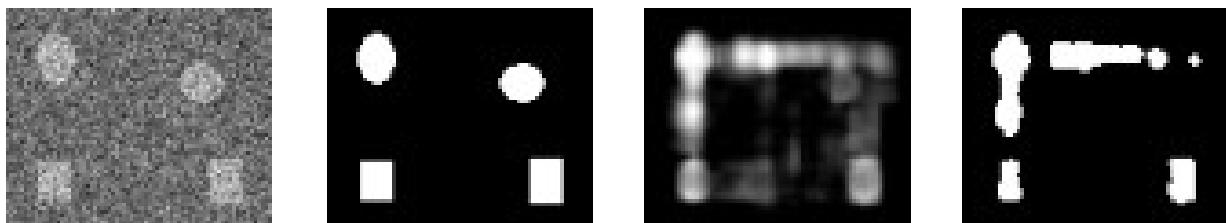
Uma convolução (por exemplo, 5×5) pode ser aplicada a qualquer imagem com tamanho maior ou igual ao tamanho do núcleo (no exemplo, $\geq 5 \times 5$). O mesmo vale para convolução transposta e para dropout. Assim, uma rede convolucional que não tem camadas densas pode receber como entrada imagens de qualquer tamanho, maiores ou iguais às imagens de treino. Se o modelo tivesse camadas densas, estas “engessariam” a resolução de entrada para um valor fixo.

Nota: Pode haver algumas outras restrições de resolução. Por exemplo, se a rede possuir uma camada que diminui resolução por 2 (além das camadas convolucionais), os números de linhas e colunas da imagem de entrada devem ser pares.

Como a nossa rede diminui a resolução da imagem por 2 três vezes, escolhi $nl \times nc = 72 \times 88$ de forma que nl e nc sejam divisíveis por $2^3 = 8$. Para poder aplicar o modelo *fcn-train1.keras* treinado em imagens 32×32 numa imagem 72×88 , devemos recriar um novo modelo com a mesma estrutura mas com a nova resolução de entrada (72×88). Este novo modelo terá a quantidade de pesos e viés exatamente igual à rede original. Depois, devemos carregar os pesos da rede treinada *fcn-train1.keras* no novo modelo usando o método:

```
model.load_weights(nome_da_rede).
```

Fazendo isso e processando a imagem *maior1x.png*, obtemos a saída da figura 10:



| | | | |
|----------------------|----------------------|------------------------|------------------|
| a – maior1x.png – qx | b – saída ideal – qy | c – saída da rede – qp | d- qp binarizada |
|----------------------|----------------------|------------------------|------------------|

Figura 10: Segmentação de *maior1x.png* usando o modelo *fcn-train1.keras* gerado pelo programa 1 (FCN) sem data augmentation.

2.2 Segmentar com data augmentation

O programa foi executado sem erro, mas a segmentação da figura 10 está evidentemente errada. Isto acontece porque o modelo foi treinado somente com elipses/retângulos centralizados no meio da imagem. É necessário que o modelo tenha sido treinado também com figuras fora do centro para poder segmentar corretamente imagens como *maior1x.png*.

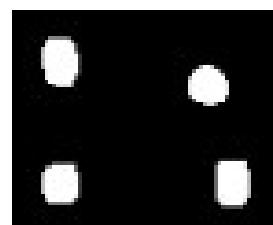
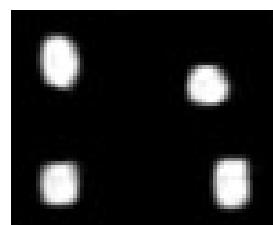
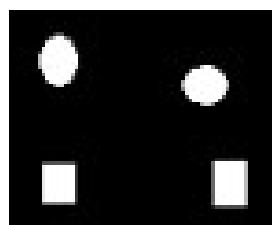
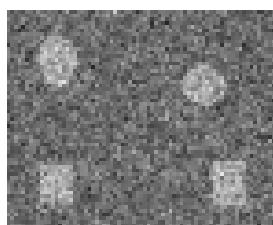
O programa 3 faz data augmentation usando *ImageDataGenerator* com deslocamentos horizontal e vertical aleatório de $\pm 70\%$ da largura/altura para gerar as amostras descentralizadas:

```
aug_dict = dict(
    width_shift_range=0.70, #float: fraction of total width, if < 1, or pixels if >= 1.
    height_shift_range=0.70, #float: fraction of total height, if < 1, or pixels if >= 1.
    horizontal_flip=True, #Boolean. Randomly flip inputs horizontally.
    fill_mode='reflect'); #One of {"constant", "nearest", "reflect" or "wrap"}.
```

O programa 4 faz inferência usando o modelo treinado no programa 3. Com isso, foi possível obter as segmentações corretas das imagens *maior1x.png* e *maior2x.png* com resolução de 72×88 e 80×88 pixels mostradas na figura 11 (apesar de que ainda está com bordas mal-definidas). Esta propriedade de redes completamente convolucionais (treinar rede completamente convolucional em imagens pequenas e aplicá-la em imagens grandes) é usada em muitas outras aplicações. Para fazer data augmentation, precisamos distorcer da mesma forma as imagens e as respectivas máscaras. Isso é feito usando a mesma semente do gerador de números pseudo-aleatórios.

Nota: Uma explicação mais detalhada desta técnica está mais adiante, na segmentação de membranas das células.

Exercício: Reescreva os programas 3 e 4 substituindo *ImageDataGenerator* por *preprocessing layers* para fazer data augmentation.

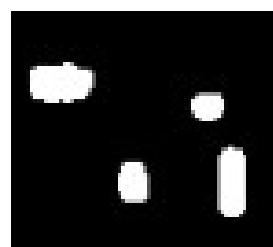
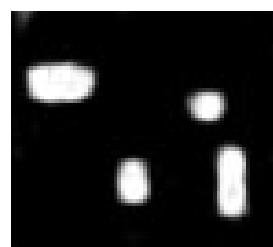
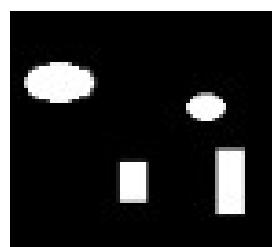
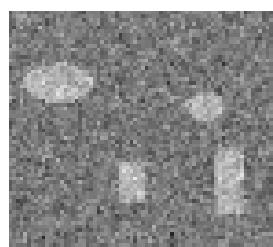


a – maior1x.png – qx

b – saída ideal – qy

c – saída da rede – qp

d- qp binarizada



a – maior2x.png – qx

b – saída ideal – qy

c – saída da rede – qp

d- qp binarizada

Figura 11: Segmentação das imagens *maior1x.png* e *maior2x.png* com resoluções 72×88 usando rede FCN treinada em imagens 32×32 com data augmentation. (a) imagem teste de entrada (qx); (b) imagem de saída ideal (qy); (c) imagem gerada pela rede (qp); (d) imagem qp binarizada.

```

#fcn_da1.py
#Treina rede fcn para segmentacao semanticas de elipses e retangulos
import os; os.environ['TF_CPP_MIN_LOG_LEVEL']=3
os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
import tensorflow.keras as keras
from tensorflow.keras.models import *
from tensorflow.keras import optimizers
from tensorflow.keras.layers import *
from tensorflow.keras.preprocessing.image import *
import sys; import cv2; import numpy as np; import matplotlib.pyplot as plt

def impHistoria(history):
    print(history.history.keys())
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title('model loss'); plt.ylabel('loss'); plt.xlabel('epoch')
    plt.legend(['train', 'test'], loc='upper left')
    plt.show()

def leCsv(nomeDir,nomeArq):
    print("Lendo: ", nomeArq); arq=open(os.path.join(nomeDir,nomeArq),"r")
    lines=arq.readlines(); arq.close(); n=len(lines)

    nl,nc = 32,32;
    AX=np.empty((n,nl,nc),dtype='uint8'); AY=np.empty((n,nl,nc),dtype='uint8')
    i=0;
    for linha in lines:
        linha=linha.strip('\n'); linha=linha.split(';')
        AX[i]=cv2.imread(os.path.join(nomeDir,linha[0]),0)
        AY[i]=cv2.imread(os.path.join(nomeDir,linha[1]),0)
        i=i+1

    ax= np.float32(AX)/255.0; ay= np.float32(AY)/255.0 #Entre 0 e +1
    ax = ax.reshape(n, nl, nc, 1); ay = ay.reshape(n, nl, nc, 1)
    return ax, ay

def fcn(input_shape = (32,32,1)):
    model = Sequential()
    model.add(Conv2D(40, kernel_size=(5,5), strides=2, activation='relu', padding='same',
                   input_shape=input_shape)) #saida 16*16*40
    model.add(Conv2D(12, kernel_size=(5,5), strides=2, activation='relu', padding='same')) #saida 8*8*12
    model.add(Dropout(0.25))
    model.add(Conv2D(12, kernel_size=(5,5), strides=2, activation='relu', padding='same')) #saida 4*4*12
    model.add(Conv2DTranspose(12, kernel_size=(5,5), strides=2, activation='relu', padding='same')) #saida 8*8*12
    model.add(Conv2DTranspose(40, kernel_size=(5,5), strides=2, activation='relu', padding='same')) #saida 16*16*40
    model.add(Dropout(0.25))
    model.add(Conv2DTranspose(1, kernel_size=(5,5), strides=2, padding='same')) #saida 32*32*1
    from tensorflow.keras.utils import plot_model;
    plot_model(model, to_file='fcn_da1.png', show_shapes=True)
    model.summary()
    return model

#<<<<<<<<<< main <<<<<<<<<<
bdDir = "."
ax, ay = leCsv(bdDir,"treino.csv")
vx, vy = leCsv(bdDir,"valida.csv")
qx, qy = leCsv(bdDir,"teste.csv")
outDir = "."; os.chdir(outDir)

nl,nc = 32,32; input_shape = (nl,nc,1)

aug_dict = dict(
    width_shift_range=0.70, #float: fraction of total width, if < 1, or pixels if >= 1.
    height_shift_range=0.70, #float: fraction of total height, if < 1, or pixels if >= 1.
    horizontal_flip=True, #Boolean. Randomly flip inputs horizontally.
    fill_mode='reflect'); #One of {"constant", "nearest", "reflect" or "wrap"}.

image_datagen = ImageDataGenerator(**aug_dict);
mask_datagen = ImageDataGenerator(**aug_dict);
seed=7; batch_size=20

image_generator = image_datagen.flow(ax, batch_size = batch_size, seed = seed);
mask_generator = mask_datagen.flow/ay, batch_size = batch_size, seed = seed);

def trainGenerator():
    train_generator=zip(image_generator,mask_generator)
    for (img,mask) in train_generator:
        mask[mask > 0.5] = 1; mask[mask <= 0.5] = 0
        yield(img,mask)

#model=fcn(input_shape)
model = load_model("fcn_da1.h5");

opt=optimizers.Adam()
model.compile(optimizer=opt, loss='mean_squared_error')
history=model.fit(trainGenerator(), steps_per_epoch=ax.shape[0]//batch_size,
                    epochs=1000, verbose=2, validation_data=(vx,vy))
impHistoria(history)

score = model.evaluate(ax, ay, verbose=0); print('Training loss:', score)
score = model.evaluate(vx, vy, verbose=0); print('Validation loss:', score)
score = model.evaluate(qx, qy, verbose=0); print('Test loss:', score)
model.save('fcn_da1.h5')

```

Programa 3: Treina modelo para segmentar elipses e retângulos usando FCN e data augmentation.

<https://colab.research.google.com/drive/1PWYJ4tKHls5WliQxv-0iT0UaS4gSPrti?usp=sharing>

Programa 4: Segmenta elipses e retângulos usando modelo FCN treinado com data augmentation.

<https://colab.research.google.com/drive/1PWyJ4tKHls5WliQxy-0iT0UaS4gSPrti?usp=sharing>

Nota (09/09/2024): Pode ser que o programa acima não rode corretamente em Tensorflow 2.17 (por causa de ImageDataGenerator). Experimente rodar em alguma versão antiga de Tensorflow, como 2.15.

```
!pip uninstall tensorflow -y  
!pip install tensorflow==2.15
```

3. U-Net

Nota para mim: Seria interessante trocar segmentação de membrana pela segmentação de vasos da retina DRIVE dataset.

<https://www.kaggle.com/datasets/andrewmvd/drive-digital-retinal-images-for-vessel-extraction>

Ou então pela segmentação do pulmão.

<https://www.kaggle.com/datasets/anasmohammedtahir/covidqu>

3.1 Introdução

FCN não conseguiu localizar de forma precisa as fronteiras entre as regiões pois, para fazer upsampling, FCN utilizou somente as informações da camada anterior com resolução duas vezes menor. Olhando somente as informações da imagem com menor resolução, é impossível localizar precisamente onde ficam as fronteiras entre as diferentes regiões na imagem com maior resolução. U-Net [Ronneberger2015] resolve este problema passando para as camadas de *upsampling* as informações das camadas de *downsampling* com a mesma resolução (figuras 12 e 13). U-Net recebe esse nome pois a sua estrutura lembra uma letra “U”, onde o braço esquerdo é a parte de *downsampling* e o braço direito é a parte de *upsampling*.

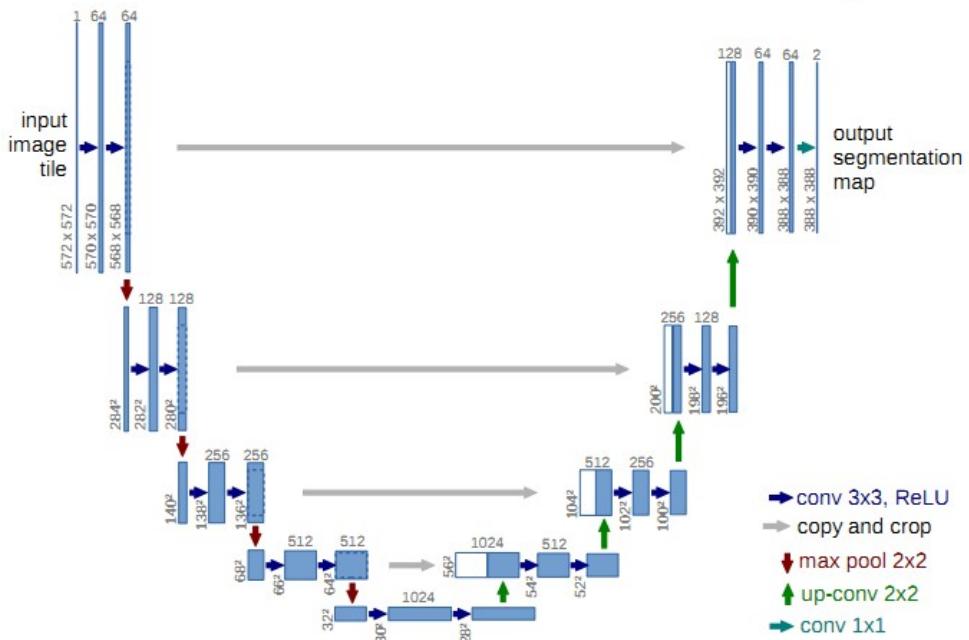


Figura 12: U-Net original.

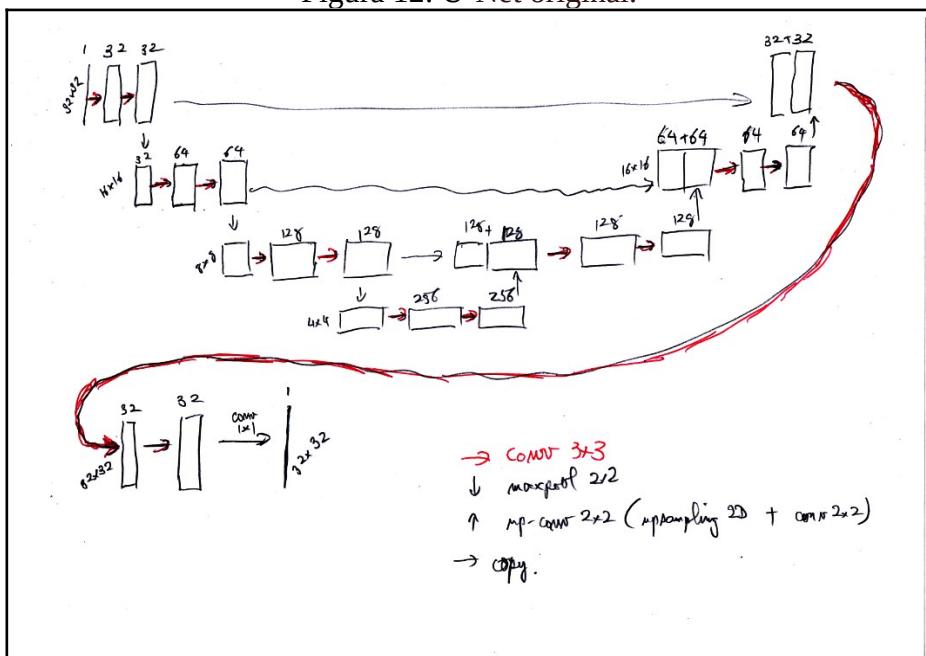


Figura 13: U-Net usada para segmentar elipses e retângulos.

U-Nets das figuras 12 e 13 possuem semelhanças com a rede FCN (figura 3): Os dois tipos de redes possuem partes de *downsampling* e *upsampling*. A principal diferença entre as duas redes é que U-Net possui ligações diretas entre as camadas de *downsampling* e *upsampling* de resoluções iguais (flechas que atravessam o vale entre os dois braços de “U”). A possibilidade de acessar informações com as mesmas resoluções permite que U-Net consiga localizar com mais precisão as fronteiras entre as diferentes regiões.

3.2 U-Net para segmentar elipses e retângulos

O programa unet-train1 (programa 5) treina U-Net para segmentar elipses e retângulos ruidosos. A estrutura da rede está mostrada na figura 14 e o erro MSE obtido, durante 1000 épocas de treinamento, na figura 15.

| Layer (type) | Output Shape | Param # | Connected to |
|--------------------------------|---------------------|---------|---------------------------------|
| input_1 (InputLayer) | (None, 32, 32, 1) | 0 | |
| conv2d_1 (Conv2D) | (None, 32, 32, 32) | 320 | input_1[0][0] |
| conv2d_2 (Conv2D) | (None, 32, 32, 32) | 9248 | conv2d_1[0][0] |
| max_pooling2d_1 (MaxPooling2D) | (None, 16, 16, 32) | 0 | conv2d_2[0][0] |
| conv2d_3 (Conv2D) | (None, 16, 16, 64) | 18496 | max_pooling2d_1[0][0] |
| conv2d_4 (Conv2D) | (None, 16, 16, 64) | 36928 | conv2d_3[0][0] |
| max_pooling2d_2 (MaxPooling2D) | (None, 8, 8, 64) | 0 | conv2d_4[0][0] |
| conv2d_5 (Conv2D) | (None, 8, 8, 128) | 73856 | max_pooling2d_2[0][0] |
| conv2d_6 (Conv2D) | (None, 8, 8, 128) | 147584 | conv2d_5[0][0] |
| max_pooling2d_3 (MaxPooling2D) | (None, 4, 4, 128) | 0 | conv2d_6[0][0] |
| conv2d_7 (Conv2D) | (None, 4, 4, 256) | 295168 | max_pooling2d_3[0][0] |
| conv2d_8 (Conv2D) | (None, 4, 4, 256) | 590080 | conv2d_7[0][0] |
| up_sampling2d_1 (UpSampling2D) | (None, 8, 8, 256) | 0 | conv2d_8[0][0] |
| conv2d_9 (Conv2D) | (None, 8, 8, 128) | 131200 | up_sampling2d_1[0][0] |
| concatenate_1 (Concatenate) | (None, 8, 8, 256) | 0 | conv2d_6[0][0]; conv2d_9[0][0] |
| conv2d_10 (Conv2D) | (None, 8, 8, 128) | 295040 | concatenate_1[0][0] |
| conv2d_11 (Conv2D) | (None, 8, 8, 128) | 147584 | conv2d_10[0][0] |
| up_sampling2d_2 (UpSampling2D) | (None, 16, 16, 128) | 0 | conv2d_11[0][0] |
| conv2d_12 (Conv2D) | (None, 16, 16, 64) | 32832 | up_sampling2d_2[0][0] |
| concatenate_2 (Concatenate) | (None, 16, 16, 128) | 0 | conv2d_4[0][0]; conv2d_12[0][0] |
| conv2d_13 (Conv2D) | (None, 16, 16, 64) | 73792 | concatenate_2[0][0] |
| conv2d_14 (Conv2D) | (None, 16, 16, 64) | 36928 | conv2d_13[0][0] |
| up_sampling2d_3 (UpSampling2D) | (None, 32, 32, 64) | 0 | conv2d_14[0][0] |
| conv2d_15 (Conv2D) | (None, 32, 32, 32) | 8224 | up_sampling2d_3[0][0] |
| concatenate_3 (Concatenate) | (None, 32, 32, 64) | 0 | conv2d_2[0][0]; conv2d_15[0][0] |
| conv2d_16 (Conv2D) | (None, 32, 32, 32) | 18464 | concatenate_3[0][0] |
| conv2d_17 (Conv2D) | (None, 32, 32, 32) | 9248 | conv2d_16[0][0] |
| conv2d_18 (Conv2D) | (None, 32, 32, 1) | 33 | conv2d_17[0][0] |
| <hr/> | | | |
| Total params: | 1,925,025 | | |
| Trainable params: | 1,925,025 | | |
| Non-trainable params: | 0 | | |

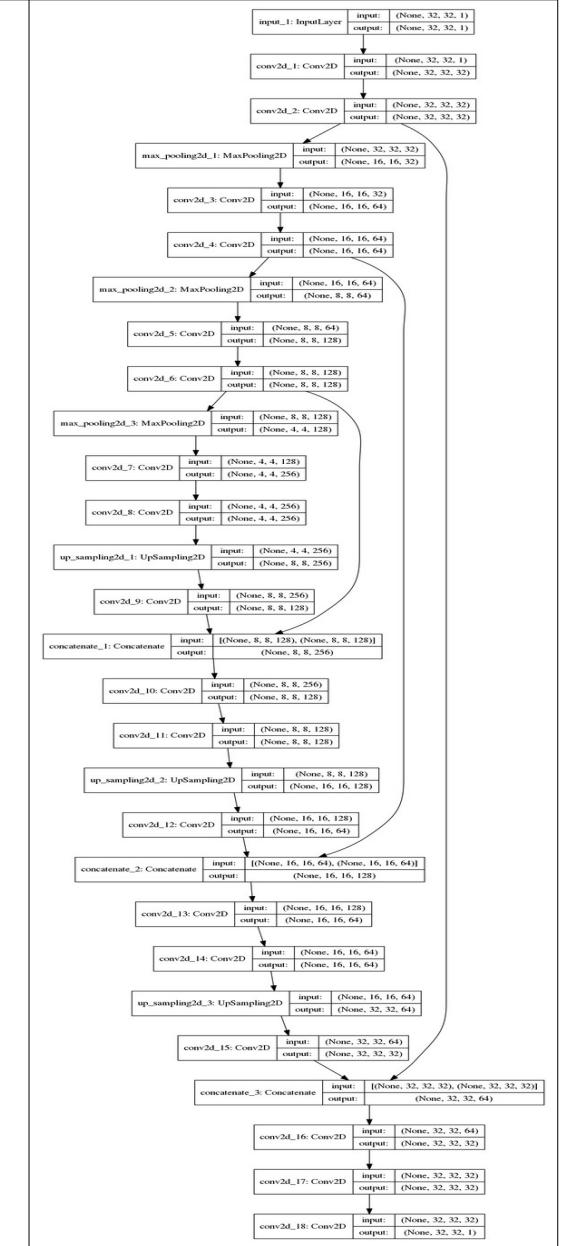


Figura 14: Estrutura da rede U-Net para segmentar elipses e retângulos.

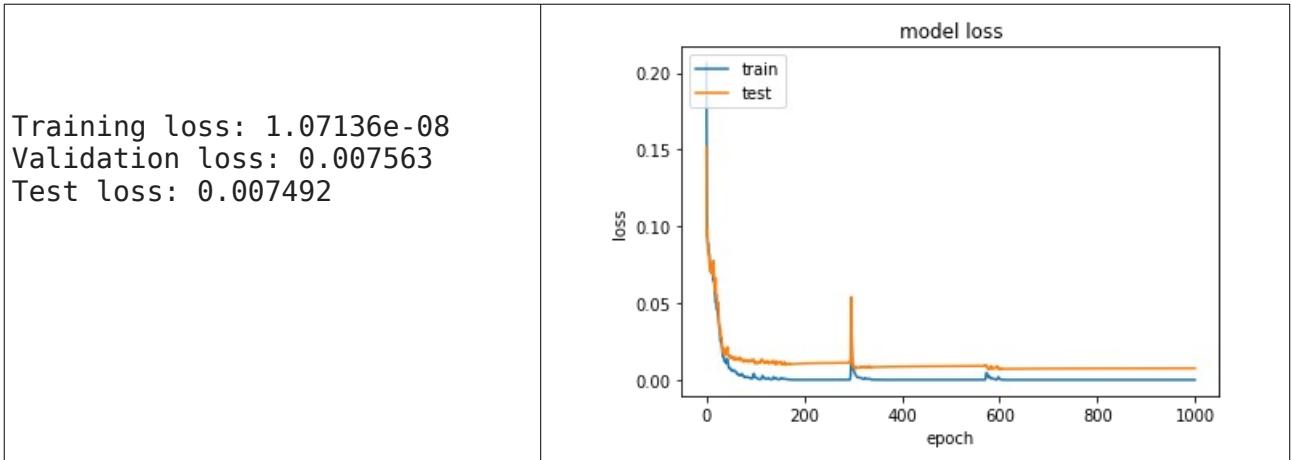


Figura 15: Os erros obtidos durante 1000 épocas de treino.

Os erros de validação e de teste (0.0075) são aproximadamente a metade do que tínhamos obtido com FCN (0.015). Além disso, o erro de treino é praticamente zero, muito menor do que os erros de validação e teste (da ordem de 10^{-8} versus 10^{-2}), significando que há *overfitting*.

Exercício: Altere o programa 4 colocando dropout, regularização L2 e outras técnicas para diminuir overfitting.

No *unet-train1.py* (programa 5), a função *leCsv* (linhas 17-32) é idêntica à mesma função do *fcn-train1.py* (programa 1). Também as leituras (nas linhas 80-84) são idênticas ao programa 1. Da mesma forma que no programa 1, aqui também normalizamos a entrada para 0 a +1.

Nota: é impossível treinar esta rede se normalizar a entrada para -1 a +1 ou para -0.5 a +0.5. Quase certamente, será possível treinar a rede com entrada [-1, +1] ou [-0.5, +0.5] se colocar camadas BatchNormalization no modelo. Experiemente.

A linha 87 chama a função *unet* para construir a rede. Nas linhas 86-88, dependendo de qual linha é deixada como comentário, o programa pode começar a treinar uma nova rede ou continuar o treino de onde parou na última execução do programa. Para isso, o programa pode carregar a rede gerada pela execução anterior (linha 91). As linhas 90-91 treinam a rede e salvam. As linhas 93-95 imprimem os erros obtidos.

Na definição da rede (linhas 34-77) não podemos usar a API (Application Programming Interface) sequencial para descrever a rede, pois U-Net não é sequencial. Em U-Net, uma camada não depende apenas da camada anterior. Precisamos usar a API funcional que permite descrever redes mais complexas.

A rede que vamos construir está na figura 13 e a estrutura da rede está desenhada na figura 14. Note que, diferentemente de FCN, estamos usando MaxPooling2D para diminuir resolução e UpSampling2D para aumentar resolução.

```

1 #unet-train1.py
2 #Treina rede unet para segmentacao semanticas de eliret
3 import os; os.environ['TF_CPP_MIN_LOG_LEVEL']= '3'
4 os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
5 import cv2; import numpy as np; np.random.seed(7); import sys
6 import tensorflow.keras as keras; from tensorflow.keras.models import *
7 from tensorflow.keras.layers import *; from tensorflow.keras.optimizers import *
8 import matplotlib.pyplot as plt
9
10 def impHistoria(history):
11     print(history.history.keys())
12     plt.plot(history.history['loss']); plt.plot(history.history['val_loss'])
13     plt.title('model loss'); plt.ylabel('loss'); plt.xlabel('epoch')
14     plt.legend(['train', 'test'], loc='upper left')
15     plt.show()
16
17 def leCsv(nomeDir,nomeArq):
18     print("Lendo: ", nomeArq); arq=open(os.path.join(nomeDir,nomeArq), "r")
19     lines=arq.readlines(); arq.close(); n=len(lines)
20
21     nl,nc = 32,32
22     AX=np.empty((n,nl,nc),dtype='uint8'); AY=np.empty((n,nl,nc),dtype='uint8')
23     i=0
24     for linha in lines:
25         linha=linha.strip('\n'); linha=linha.split(';')
26         AX[i]=cv2.imread(os.path.join(nomeDir,linha[0]),0)
27         AY[i]=cv2.imread(os.path.join(nomeDir,linha[1]),0)
28         i=i+1
29
30     ax= np.float32(AX)/255.0; ay= np.float32(AY)/255.0 #Entre 0 e +1
31     ax = ax.reshape(n, nl, nc, 1); ay = ay.reshape(n, nl, nc, 1)
32     return ax, ay
33
34 def unet(input_size = (32,32,1)):
35     n=32
36     inputs = Input(input_size) #32x32
37     conv2 = Conv2D(n, 3, activation = 'relu', padding = 'same')(inputs)
38     conv2 = Conv2D(n, 3, activation = 'relu', padding = 'same')(conv2)
39     pool2 = MaxPooling2D(pool_size=(2, 2))(conv2) #16x16
40
41     conv3 = Conv2D(2*n, 3, activation = 'relu', padding = 'same')(pool2) #16x16
42     conv3 = Conv2D(2*n, 3, activation = 'relu', padding = 'same')(conv3)
43     pool3 = MaxPooling2D(pool_size=(2, 2))(conv3) #8x8
44
45     conv4 = Conv2D(4*n, 3, activation = 'relu', padding = 'same')(pool3) #8x8
46     conv4 = Conv2D(4*n, 3, activation = 'relu', padding = 'same')(conv4)
47     pool4 = MaxPooling2D(pool_size=(2, 2))(conv4) #4x4
48
49     conv5 = Conv2D(8*n, 3, activation = 'relu', padding = 'same')(pool4) #4x4
50     conv5 = Conv2D(8*n, 3, activation = 'relu', padding = 'same')(conv5)
51
52     up6 = Conv2D(4*n, 2, activation = 'relu', padding = 'same')
53         )(UpSampling2D(size = (2,2))(conv5)) #8x8
54     merge6 = concatenate([conv4,up6], axis = 3) #8x8
55     conv6 = Conv2D(4*n, 3, activation = 'relu', padding = 'same')(merge6)
56     conv6 = Conv2D(4*n, 3, activation = 'relu', padding = 'same')(conv6) #8x8
57
58     up7 = Conv2D(2*n, 2, activation = 'relu', padding = 'same')
59         )(UpSampling2D(size = (2,2))(conv6)) #16x16
60     merge7 = concatenate([conv3,up7], axis = 3)
61     conv7 = Conv2D(2*n, 3, activation = 'relu', padding = 'same')(merge7)
62     conv7 = Conv2D(2*n, 3, activation = 'relu', padding = 'same')(conv7) #16x16
63
64     up8 = Conv2D(n, 2, activation = 'relu', padding = 'same')
65         )(UpSampling2D(size = (2,2))(conv7)) #32x32
66     merge8 = concatenate([conv2,up8], axis = 3)
67     conv8 = Conv2D(n, 3, activation = 'relu', padding = 'same')(merge8)
68     conv8 = Conv2D(n, 3, activation = 'relu', padding = 'same')(conv8) #32x32
69
70     conv9 = Conv2D(1, 1, activation = 'sigmoid', padding = 'same')(conv8) #32x32
71
72     model = Model(inputs = inputs, outputs = conv9)
73     model.compile(optimizer = Adam(learning_rate=1e-3), loss = 'mean_squared_error')
74     from tensorflow.keras.utils import plot_model
75     plot_model(model, to_file='unet-train1.png', show_shapes=True)
76     model.summary()
77     return model
78
79 ##### main #####
80 bdDir = "."
81 ax, ay = leCsv(bdDir, "treino.csv")
82 vx, vy = leCsv(bdDir, "valida.csv")
83 qx, qy = leCsv(bdDir, "teste.csv")
84 outDir = "."; os.chdir(outDir)
85
86 #Escolha entre comecar treino do zero ou continuar o treino de onde parou
87 model=unet()
88 #model = load_model("unet1.h5");
89
90 history=model.fit(ax, ay, batch_size=10, epochs=1000, verbose=2, validation_data=(vx,vy));
91 impHistoria(history); model.save("unet1.h5");
92
93 score = model.evaluate(ax, ay, verbose=0); print('Training loss:', score)
94 score = model.evaluate(vx, vy, verbose=0); print('Validation loss:', score)
95 score = model.evaluate(qx, qy, verbose=0); print('Test loss:', score)

```

Programa 5: Treina U-Net para segmentar elipses e retângulos.

https://colab.research.google.com/drive/1DfixQxjDNaSQkQvihHA4v_UBSjeZG8H?usp=sharing

Exercício: Coloque camadas “batch normalization” na U-Net do programa 5 e verifique se a rede converge mais rapidamente.

Nota: Há um exemplo de U-Net com Dropout e BatchNormalization em:

<https://towardsdatascience.com/understanding-semantic-segmentation-with-unet-6be4f42d4b47>

Exercício: Experimente normalizar a entrada do programa 5 usando camada de Keras *Normalization*.

Exercício: É possível usar a propriedade das redes completamente convolucionais na U-Net para segmentar imagens maiores do que as imagens de treino? Se for possível, implemente esta ideia.

O programa 6 (unet-pred1.py) utiliza rede U-Net já treinada pelo programa 5 (unet-train1.py) para segmentar elipse ou retângulo. Na figura 9, as colunas *e* e *f* (azul) mostram as saídas da U-Net. A segmentação não está perfeita, mas é possível diferenciar claramente as elipses dos retângulos. Os retângulos não têm mais cantos arredondados. Além disso, a saída em níveis de cinza já é praticamente uma imagem binária, indicando que a rede decidiu o valor da saída com maior confiança. Isto não acontece com as saídas da FCN (colunas *c* e *d* em amarelo).

Nota: Preste atenção na linha 32:

`QPB[qp>=0.5] = 255`

A condição “`qp>=0.5`” constrói uma matriz booleana do mesmo tamanho que *qp* contendo *True* nos elementos onde a condição é satisfeita e *False* onde a condição não é satisfeita. O comando acima coloca 255 nos elementos da matriz QPB indexadas por True (que satisfazem a condição “`qp>=0.5`”).

```
1 #unet-pred1.py
2 #Faz segmentacao de elipses e retangulos usando rede gerada pelo unet-train1.py
3 import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
4 os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
5 import cv2; import numpy as np; np.random.seed(7)
6 import tensorflow.keras as keras
7 from tensorflow.keras.models import load_model
8 from tensorflow.keras.layers import Dropout, Conv2D, Conv2DTranspose
9 from tensorflow.keras import optimizers
10 import sys; from sys import argv
11
12 ##### main #####
13 bdDir = "."
14 outDir = "."; os.chdir(outDir)
15 nome="077"
16 inImgX = nome+"x.png"; inImgY = nome+"y.png"
17 outImgG = nome+"g.png"; outImgB = nome+"b.png"
18 arquivoRede = "unet1.h5"
19
20 model = load_model(os.path.join(outDir,arquivoRede))
21 QX=cv2.imread(os.path.join(bdDir,inImgX),0)
22 QY=cv2.imread(os.path.join(bdDir,inImgY),0)
23 nl=QX.shape[0]; nc=QX.shape[1]
24 qx=np.float32(QX)/255.0 #Entre 0 e +1
25 qx=qx.reshape(1, nl, nc, 1)
26
27 qp=model.predict(qx); qp=qp.reshape(nl,nc) # entre 0 e +1
28
29 QPG=255.0*qp; QPG=np.clip(QPG,0,255) # Entre 0 e 255
30 QPG=np.uint8(QPG); cv2.imwrite(os.path.join(outDir,outImgG),QPG)
31
32 QPB=np.zeros((nl,nc),dtype= uint8'); QPB[ qp>=0.5 ] = 255
33 cv2.imwrite(os.path.join(outDir,outImgB),QPB)
34
35 from matplotlib import pyplot as plt
36 f = plt.figure()
37 f.add_subplot(1,4,1); plt.imshow(QX,cmap="gray"); plt.axis('off')
38 f.add_subplot(1,4,2); plt.imshow(QY,cmap="gray"); plt.axis('off')
39 f.add_subplot(1,4,3); plt.imshow(QPG,cmap="gray"); plt.axis('off')
40 f.add_subplot(1,4,4); plt.imshow(QPB,cmap="gray"); plt.axis('off')
41 plt.show(block=True)
```

Programa 6: Segmenta elipses e retângulos usando a rede U-Net treinada.

https://colab.research.google.com/drive/1DfixQxjDNaSQkQvhHA4v_UBSjeZG8H?usp=sharing

Exercício recomendado: Modifique os programas 5 e 6 para que a rede receba imagens de entrada no intervalo de [-1, +1]. Acrescente camadas de BatchNormalization (sem isto, o treino não vai convergir).

Exercício: Reescreva o programa 1 (regression.py) da aula “densakeras-ead” usando API funcional.

Exercício: Reescreva o programa 2 (abc1.py) da aula “densakeras-ead” usando API funcional. Teste para verificar que está fazendo a mesma tarefa que fazia API sequencial.

Exercício: Reescreva o programa 7 (cnn1.py) da aula “convkeras-ead” utilizando API funcional.

[PSI5790 aula 9. Lição de casa #2/2, vale 5,0. Exercício extra vale +2] O objetivo deste exercício é verificar quais alterações devemos introduzir na FCN (programas 1 e 2, MSE de validação/teste 0.016) para obter uma qualidade de segmentação semelhante a U-Net (sem trocar as convoluções e convoluções transpostas por maxpooling2d e upsampling2d). Ou seja, vamos construir U-Net usando convolução e convolução transposta com strides=2.

a) Reescreva os programas 1 e 2 desta apostila (*fcn-train1.py* e *fcn-pred1.py*) usando API funcional. Verifique se continua funcionando da mesma forma que antes (precisa obter MSE de validação/teste de aproximadamente 0.016). Segmente alguma imagem teste e mostre.

Solução privada em [~/deep/algpi/segment/licao/fcn-train2.py](#) e [fcn-pred2.py](#).

b - vale 5) Acrescente as ligações que atravessam os dois braços de “U” da rede FCN (como na U-Net). Fazendo isso e ajustando os números de convoluções das camadas como na U-Net (programa 4), consegui chegar a MSE de validação/teste 0.012. Segmente alguma imagem teste e mostre.

Solução privada em [~/deep/algpi/segment/licao/unet-train2.py](#).

c – exercício extra. vale +2) Para o modelo ficar mais parecido com U-Net, substitua todas convoluções e convoluções transpostas 5×5 por sequências de duas convoluções 3×3 . Por exemplo, substitua:

```
x2 = Conv2D(2*n, kernel_size=(5,5), strides=2, activation='relu', padding='same')(x2)  
por:
```

```
x2 = Conv2D(2*n, kernel_size=(3,3), strides=1, activation='relu', padding='same')(x2)  
x2 = Conv2D(2*n, kernel_size=(3,3), strides=2, activation='relu', padding='same')(x2)
```

Consegui chegar a taxa de erro de validação/teste 0.0073, a mesma da U-Net. Segmente alguma imagem teste e mostre.

Solução privada em [~/deep/algpi/segment/licao/unet-train2.py](#) e [unet-pred3.py](#).

Exercício: Altere a solução (c) do exercício anterior com data augmentation para segmentar imagens de tamanhos diferentes de 32×32 (isto é, 72×88 ou 80×88). Segmente as seis imagens maior? x.png (que estão em [segment.zip](#)).

Veja na figura 16 que a qualidade de segmentação é melhor do que usando FCN (figura 10). Os retângulos não aparecem mais com cantos arredondados.

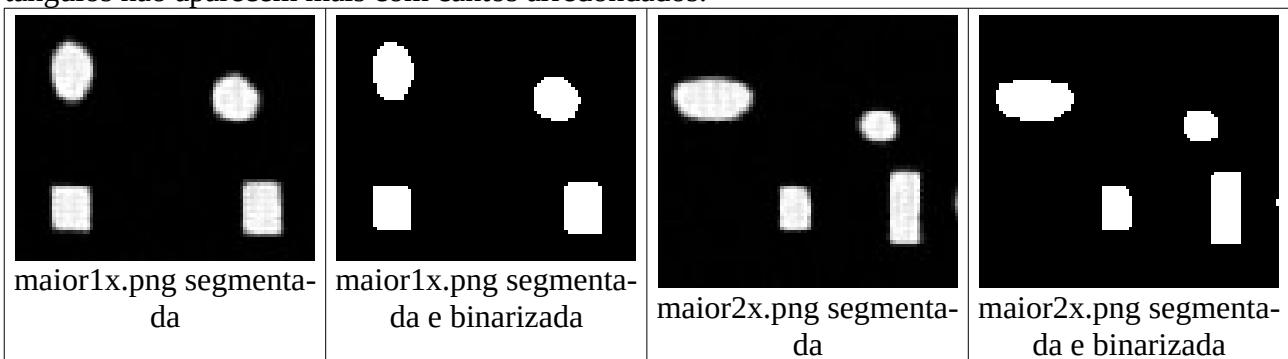


Figura 16: Maior1x e maior2x segmentadas pela U-Net. Compare com figura 10, segmentada pela FCN.

4. Segmentação de membrana celular

Nota para mim: Substituir este problema pela segmentação dos vasos sanguíneos da retina ou de covid.

https://keras.io/examples/vision/oxford_pets_image_segmentation/

<https://www.tensorflow.org/tutorials/images/segmentation>

<https://pyimagesearch.com/2022/02/21/u-net-image-segmentation-in-keras/>

<https://towardsdatascience.com/vessel-segmentation-with-python-and-keras-722f9fb71b21>

Até aqui, usamos FCN (fully convolutional network) e U-net num problema muito simples: segmentar elipses e retângulos. Concluímos que U-Net funciona melhor que FCN para segmentação. Agora, vamos testar U-Net numa aplicação mais útil do que segmentar elipses e retângulos. Vamos tentar resolver o problema de segmentar as membranas das células.

O site abaixo fornece 30 fatias sequenciais de um volume 3D das células de Drosophila. Vamos considerá-las como 30 imagens independentes de treinamento (entrada-saída, 512×512).

<https://github.com/zhixuhao/unet>

Esse site, por sua vez, pegou as imagens originais de :

http://brainiac2.mit.edu/isbi_challenge

Há um outro conjunto de 30 imagens para teste 512×512 , só que infelizmente não fornece as saídas verdadeiras, impossibilitando calcular o erro de teste. O conjunto de imagens está em:

(1) <http://www.lps.usp.br/hae/apostila/membrane.zip> OU

(2) https://drive.google.com/drive/folders/1KnXok4Ab5dGHs-Xc7yMaP6h_wJxzp4dZ?usp=sharing

A célula de programa abaixo baixa membrane.zip e descompacta no diretório default.

```
1 #membrane_leitura1.py: Descarregar membrane.zip
2 url='http://www.lps.usp.br/hae/apostila/membrane.zip'
3 import os; nomeArq=os.path.split(url)[1]
4 if not os.path.exists(nomeArq):
5     print("Baixando o arquivo",nomeArq,"para diretorio default",os.getcwd())
6     os.system("wget -nc -U 'Firefox/50.0' "+url)
7 else:
8     print("O arquivo",nomeArq,"ja existe no diretorio default",os.getcwd())
9 print("Descompactando arquivos novos de",nomeArq)
10 os.system("unzip -u "+nomeArq)
```

<https://colab.research.google.com/drive/1r0rIgQty8lPKnwGG3QCw2WO7PZa7ibQJ?usp=sharing>

A figura 17 mostra algumas imagens desse conjunto de dados.

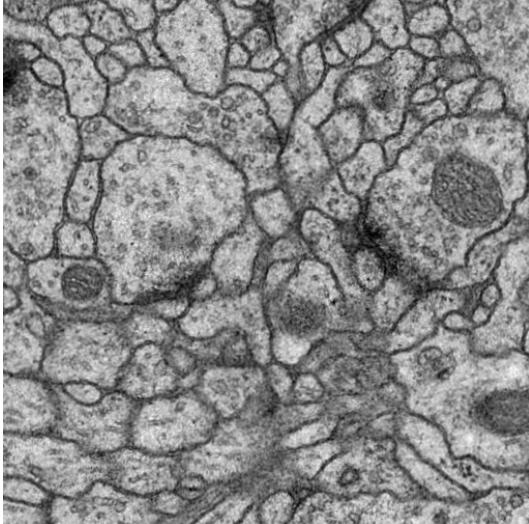
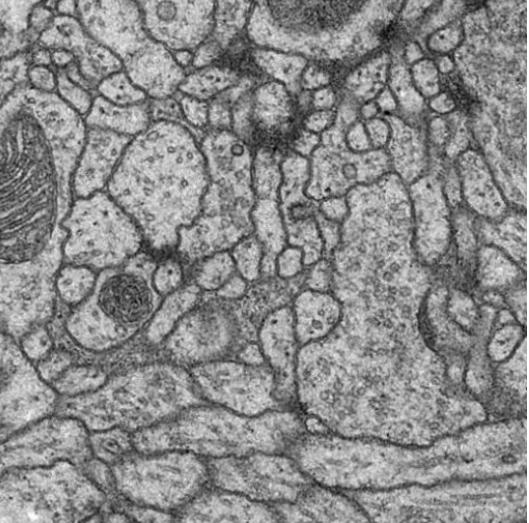
| image (AX) | label ou mask (AY) |
|---|--|
|  0.png |  0.png |
|  10.png |  10.png |

Figura 17: Algumas das 30 imagens de treinamento do BD *membrane*.

Para estudarmos os conceitos, sem perdermos muito tempo treinando, vamos trabalhar com as imagens redimensionadas para 128×128 (as originais são 512×512).

Nota: Este redimensionamento será feita pela função `flow_from_directory` de Keras.

Como temos somente 30 pares de imagens de treino, é essencial fazer *data augmentation*. Neste problema, *data augmentation* deve distorcer da mesma forma as imagens de entrada e saída. O programa 6 (`gera1.py`) testa *data augmentation* em tempo real a ser usado durante o treino. Para isso, o programa aplica, nas imagens *image* (AX) e *mask* (AY), distorções usando a mesma semente pseudo-aleatória. Algumas versões reduzidas e distorcidas de quatro imagens (entrada-original, saída-original, entrada-distorcida e saída-distorcida) estão mostradas na figura 18.

As linhas 14-20 especificam as distorções que podem ser introduzidas. Por exemplo, `rotation_range=10` indica que a imagem de entrada pode sofrer rotação aleatória no intervalo de $\pm 10^\circ$, `width_shift_range=0.05` indica que a imagem pode sofrer deslocamento horizontal de $\pm 5\%$ da largura da imagem, e assim por diante. Veja o manual de Keras para maiores detalhes:

<https://keras.io/api/preprocessing/image/#imagedatagenerator-class>

A estrutura “dict” do Python (linha 14) é uma coleção não-ordenada, mutável e indexada [<https://docs.python.org/3/tutorial/datastructures.html>, https://www.w3schools.com/python/python_dictionaries.asp]. Ela é provavelmente implementada usando tabela hash. Ela pode ser utilizada para especificar parâmetros de uma função, juntamente com operador ** (linhas 39 e 45). Exemplo [<https://softwareengineering.stackexchange.com/questions/131403/what-is-the-name-of-in-python>]:

```
kwargs = {"a": 1, "b": 2, "c": 3} OU kwargs = dict(a=1, b=2, c=3)
print(kwargs["a"]) imprime 1
kwargs["a"] = 7; print(kwargs["a"]) imprime 7
f(**kwargs) faz o mesmo que f(a=7, b=2, c=3)
```

Aqui, esta técnica é usada para especificar uma única vez os parâmetros das distorções.

Veja também [<https://treyhunner.com/2018/10/asterisks-in-python-what-they-are-and-how-to-use-them/>]

Neste programa, as imagens de treino e máscara estão respectivamente nos diretórios ‘./membrane/train/image’ e ‘./membrane/train/label’. As imagens e as máscaras correspondentes têm o mesmo nome. As imagens são carregadas diretamente dos diretórios usando o método `flow_from_directory` (linhas 26, 31, 36 e 41). Isto é útil quando o BD é grande e não cabe inteiramente na memória.

O comando:

`generator.next()`

embaralha as imagens do diretório especificado, pegam `batch_size` imagens, distorcem-nas e devolvem essas imagens distorcidas. Usando o mesmo `seed`, os 4 *generators* irão pegar as mesmas imagens nos diretórios. Os 2 primeiros *generators* não irão distorcer as imagens, pois foram chamados sem parâmetros (linhas 27 e 33). Os 2 últimos irão distorcer as imagens usando os mesmos parâmetros (linhas 39 e 45).

Exercício: Modifique o programa 6 para que este introduza um único tipo de distorção de cada vez e observe as saídas geradas. As saídas estão distorcidas corretamente? Por exemplo, modifique o programa 6 para que faça somente a rotação e verifique que as imagens de saída são todas rotações das imagens de entrada.

Exercício: Modifique o programa 6 para que leia todas as imagens na memória e trabalhe com todas elas na memória.

Exercício: Faça *data augmentation* usando *preprocessing layer* (em vez de *ImageDataGenerator*).

Programa 6: Gera pares de imagens 128×128 distorcidas.

<https://colab.research.google.com/drive/1r0rIgQty8lPKnwGG3QCw2WO7PZa7ibQJ?usp=sharing>

Nota: Pode ser que o programa acima não rode corretamente em Tensorflow 2.17 (por causa de ImageDataGenerator). Experimente rodar em Tensorflow 2.15.

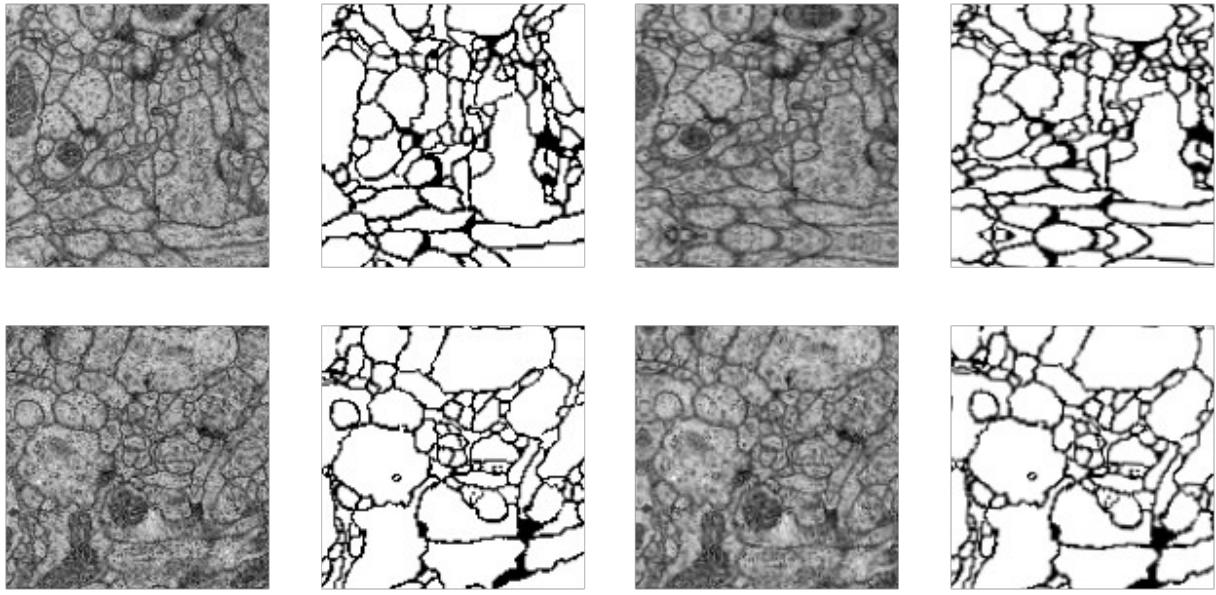


Figura 18: Versões reduzidas e distorcidas de duas imagens. Da esquerda para direita: (a) imagem original, (b) máscara original, (c) imagem distorcida, (d) máscara distorcida.

O programa 7 treina U-Net usando data augmentation em tempo real e lendo as imagens de disco. É possível continuar o treino de onde parou, bastando comentar/descomentar as linhas 109 e 110:

```
model = unet();
#model = load_model(nomeprog+".h5");
```

A função *trainGenerator* gera um batch de imagens de entrada-saída e devolve batches de pares de imagens à medida em que for solicitado, usando *yield* (linhas 101-105). Você pode pegar batch de imagens usando:

```
image,mask=next(trainGenerator())
```

O comando *yield* é semelhante ao *return* em Python. A diferença é que *yield* não destrói as variáveis locais, de forma que na próxima chamada, a função (neste caso, chamado de generator) recomeça de onde parou.

<https://www.geeksforgeeks.org/difference-between-yield-and-return-in-python/#:~:text=Return%20is%20generally%20used%20to,result%20to%20the%20caller%20statement.&text=It%20replaces%20the%20return%20of,execution%20without%20destroying%20local%20variables>

O programa 7 possui camadas de batch normalization para facilitar a sua convergência. Pode ser que acrescentar regularização L2 e outras técnicas ajudem a melhorar ainda mais a acuracidade.

O programa 8 faz predição usando a rede gerada pelo programa 7.

Nota: Há um exemplo de U-Net com Dropout e BatchNormalization em

<https://towardsdatascience.com/understanding-semantic-segmentation-with-unet-6be4f42d4b47>

Exercício: Altere o programa 7 fazer treino e teste usando imagens tamanho 256×256 . É necessário acrescentar mais camadas para obter uma boa segmentação?

Exercício: Elimine as ligações entre os dois braços do programa 7 e verifique a qualidade da segmentação observando: (1) Função de perda e acuracidade de treino. (2) Imagens segmentadas. Mostre no vídeo algumas imagens segmentadas com e sem as ligações entre os dois braços. Tire conclusões a respeito da necessidade (ou não) das ligações entre os dois braços.

Nota: Fazendo isso, obtive loss=0.072 e accuracy=0.902, bastante pior do que loss=0.032 e accuracy: 0.956 da UNet.

```

1 #membrane_unet1.py
2 import os; os.environ['TF_CPP_MIN_LOG_LEVEL']=3'
3 os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
4 import cv2; import numpy as np; np.random.seed(7); import sys;
5 import tensorflow.keras as keras; import tensorflow.keras.backend as K
6 from tensorflow.keras.models import *; from tensorflow.keras.layers import *
7 from tensorflow.keras.optimizers import *
8 from tensorflow.keras.callbacks import ModelCheckpoint, LearningRateScheduler
9 from tensorflow.keras.preprocessing.image import ImageDataGenerator
10 import matplotlib.pyplot as plt
11
12 def impHistoria(history):
13     print(history.history.keys())
14     plt.plot(history.history['loss'])
15     plt.title('train loss'); plt.ylabel('MSE loss'); plt.xlabel('epoch')
16     plt.legend(['train loss'], loc='upper left')
17     plt.show()
18     plt.plot(history.history['accuracy'])
19     plt.title('train accuracy'); plt.ylabel('accuracy'); plt.xlabel('epoch')
20     plt.legend(['train accuracy'], loc='upper left')
21     plt.show()
22
23 def unet(input_size = (128,128,1)):
24     inputs = Input(input_size) #128x128
25     conv2 = Conv2D(64, 3, activation = 'relu', padding = 'same' )(inputs)
26     conv2 = Conv2D(64, 3, activation = 'relu', padding = 'same' )(conv2)
27     pool2 = MaxPooling2D(pool_size=(2, 2))(conv2) #64x64
28     pool2 = BatchNormalization()(pool2)
29
30     conv3 = Conv2D(128, 3, activation = 'relu', padding = 'same' )(pool2)
31     conv3 = Conv2D(128, 3, activation = 'relu', padding = 'same' )(conv3)
32     pool3 = MaxPooling2D(pool_size=(2, 2))(conv3) #32x32
33     pool3 = BatchNormalization()(pool3)
34
35     conv4 = Conv2D(256, 3, activation = 'relu', padding = 'same' )(pool3)
36     conv4 = Conv2D(256, 3, activation = 'relu', padding = 'same' )(conv4)
37     drop4 = Dropout(0.5)(conv4) #32x32
38     pool4 = MaxPooling2D(pool_size=(2, 2))(drop4) #16x16
39     pool4 = BatchNormalization()(pool4)
40
41     conv5 = Conv2D(512, 3, activation = 'relu', padding = 'same' )(pool4)
42     conv5 = Conv2D(512, 3, activation = 'relu', padding = 'same' )(conv5)
43     drop5 = Dropout(0.5)(conv5) #16x16
44
45     up6 = Conv2D(256, 2, activation = 'relu', padding = 'same' )(UpSampling2D(size = (2,2))(drop5)) #32x32
46     merge6 = concatenate([drop4,up6], axis = 3)#32x32
47     conv6 = Conv2D(256, 3, activation = 'relu', padding = 'same' )(merge6)
48     conv6 = Conv2D(256, 3, activation = 'relu', padding = 'same' )(conv6)
49     conv6 = BatchNormalization()(conv6)
50
51     up7 = Conv2D(128, 2, activation = 'relu', padding = 'same' )(UpSampling2D(size = (2,2))(conv6)) #64x64
52     merge7 = concatenate([conv3,up7], axis = 3)
53     conv7 = Conv2D(128, 3, activation = 'relu', padding = 'same' )(merge7)
54     conv7 = Conv2D(128, 3, activation = 'relu', padding = 'same' )(conv7)
55     conv7 = BatchNormalization()(conv7)
56
57     up8 = Conv2D(164, 2, activation = 'relu', padding = 'same' )(UpSampling2D(size = (2,2))(conv7)) #128x128
58     merge8 = concatenate([conv2,up8], axis = 3)
59     conv8 = Conv2D(64, 3, activation = 'relu', padding = 'same' )(merge8)
60     conv8 = Conv2D(64, 3, activation = 'relu', padding = 'same' )(conv8)
61     conv8 = BatchNormalization()(conv8)
62
63     conv8 = Conv2D(2, 3, activation = 'relu', padding = 'same' )(conv8)
64     conv9 = Conv2D(1, 1, activation = 'sigmoid', padding = 'same',
65                   bias_initializer=keras.initializers.Constant(value=1.5))(conv8)
66     model = Model(inputs = inputs, outputs = conv9)
67     model.compile(optimizer = Adam(learning_rate = 1e-3),
68                   loss = 'mean_squared_error', metrics = ['accuracy'])
69
70     model.summary()
71     return model
72
73 #####
74 #####
75 #####
76 ##### main #####
77 nomeprog="membrane_unet"; train_path='./membrane/train'; outDir = "."; os.chdir(outDir)
78
79 aug_dict = dict(rotation_range=10, #Int. Degree range for random rotations.
80                  width_shift_range=0.05, #float: fraction of total width, if < 1, or pixels if >= 1.
81                  height_shift_range=0.05, #float: fraction of total height, if < 1, or pixels if >= 1.
82                  shear_range=0, #Float. Shear Intensity (Shear angle in counter-clockwise direction in degrees)
83                  zoom_range=0.2, #Range for random zoom. If a float, [lower, upper] = [1-zoom_range, 1+zoom_range].
84                  horizontal_flip=False, #Boolean. Randomly flip inputs horizontally.
85                  fill_mode='reflect'); #One of {"constant", "nearest", "reflect" or "wrap"}.
86
87 image_folder='image'; mask_folder= 'label';
88 target_size = (128,128); batch_size=10; seed = 7; save_to_dir = None;
89
90 image_datagen = ImageDataGenerator(**aug_dict);
91 mask_datagen = ImageDataGenerator(**aug_dict);
92
93 image_generator = image_datagen.flow_from_directory(
94     train_path, classes = [image_folder], class_mode = None, color_mode = "grayscale",
95     target_size = target_size, batch_size = batch_size, seed = seed);
96
97 mask_generator = mask_datagen.flow_from_directory(
98     train_path, classes = [mask_folder], class_mode = None, color_mode = "grayscale",
99     target_size = target_size, batch_size = batch_size, seed = seed);
100
101 def trainGenerator():
102     train_generator=zip(image_generator,mask_generator)
103     for (img,mask) in train_generator:
104         img=img/255; mask=mask/255; mask[mask > 0.5] = 1; mask[mask <= 0.5] = 0
105         yield(img,mask)

```

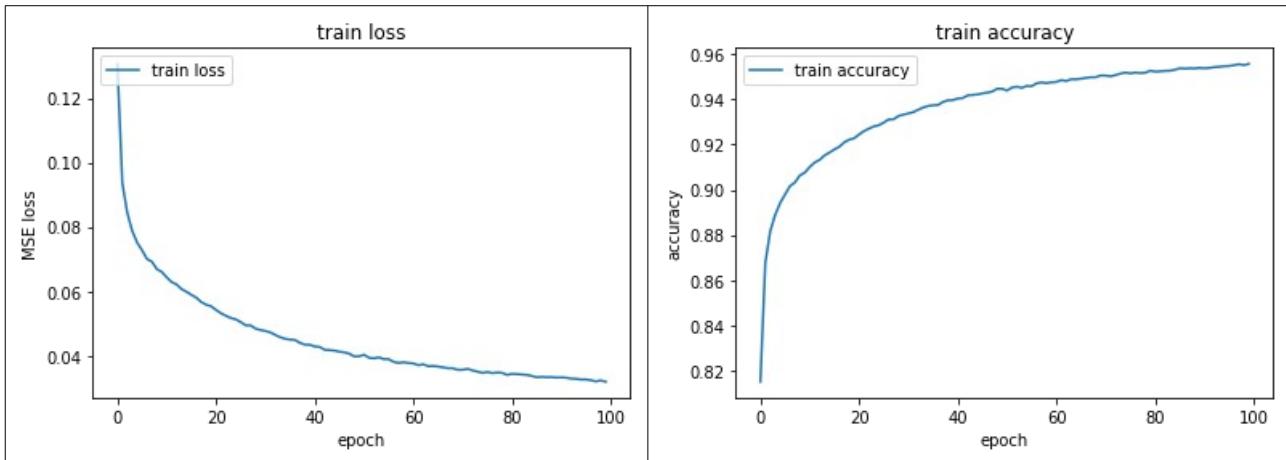
```

107 #<<<<<<<<<< main <<<<<<<<<<<<<
108 #Escolha entre comecar treino do zero ou continuar o treino de onde parou
109 model = unet();
110 #model = load_model(nomeprog+".h5");
111 from tensorflow.keras.utils import plot_model;
112 plot_model(model, to_file=nomeprog+'.png')
113
114 history=model.fit(trainGenerator(),steps_per_epoch=30,epochs=100,verbose=2);
115 impHistoria(history); model.save(nomeprog+".h5");

```

Programa 7: Treina U-Net para segmentar as imagens das células.

<https://colab.research.google.com/drive/1PoviHGMVM-m9SbKqUieL4VjR-JnEZuoq?usp=sharing>



```

Epoch  1/100 - 9s - loss: 0.1306 - accuracy: 0.8149
Epoch  10/100 - 3s - loss: 0.0643 - accuracy: 0.9100
Epoch  20/100 - 3s - loss: 0.0553 - accuracy: 0.9228
Epoch  30/100 - 3s - loss: 0.0479 - accuracy: 0.9333
Epoch  40/100 - 3s - loss: 0.0434 - accuracy: 0.9397
Epoch  50/100 - 3s - loss: 0.0398 - accuracy: 0.9447
Epoch  60/100 - 3s - loss: 0.0377 - accuracy: 0.9476
Epoch  70/100 - 3s - loss: 0.0356 - accuracy: 0.9506
Epoch  80/100 - 3s - loss: 0.0340 - accuracy: 0.9527
Epoch  90/100 - 3s - loss: 0.0332 - accuracy: 0.9539
Epoch  100/100 - 3s - loss: 0.0319 - accuracy: 0.9558

```

Programa 8: Segmenta células usando a rede treinada no programa 7.

<https://colab.research.google.com/drive/1PoviHGMVM-m9SbKqUieL4VjR-JnEZuo?usp=sharing>

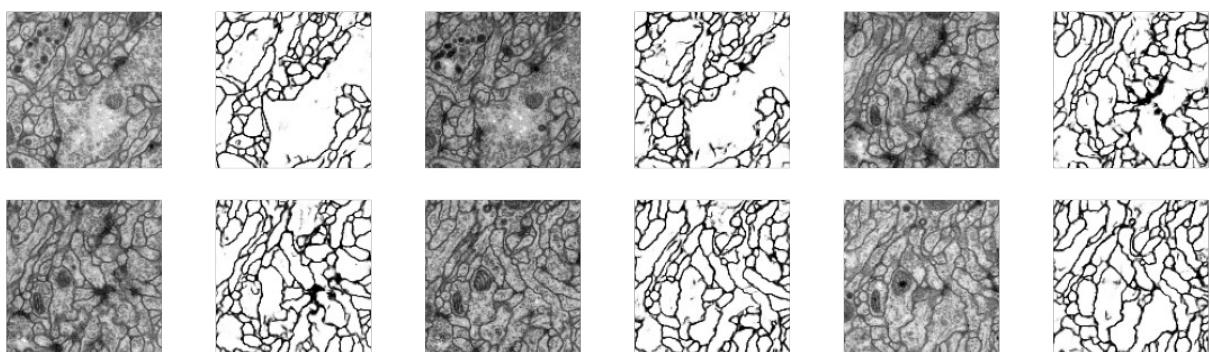


Figura 19: Algumas células (imagens da esquerda) e as membranas segmentadas por U-net (imagens da direita).

5. PSPNet

Artigo original PSPNet: <https://arxiv.org/abs/1612.01105>

5.1 Introdução

Estudamos FCN e U-Net. Para segmentar imagens médicas, de microscopia e outras assemelhadas, U-Net parece ser a técnica mais usada, pois esses tipos de imagens costumam estar numa única escala (não há células vistas de perto e outras vistas de longe). Porém, para imagens adquiridas do mundo real 3D, onde objetos podem aparecer em diferentes escalas, PSPNet (Pyramid Scene Parsing Network) é uma das mais usadas.

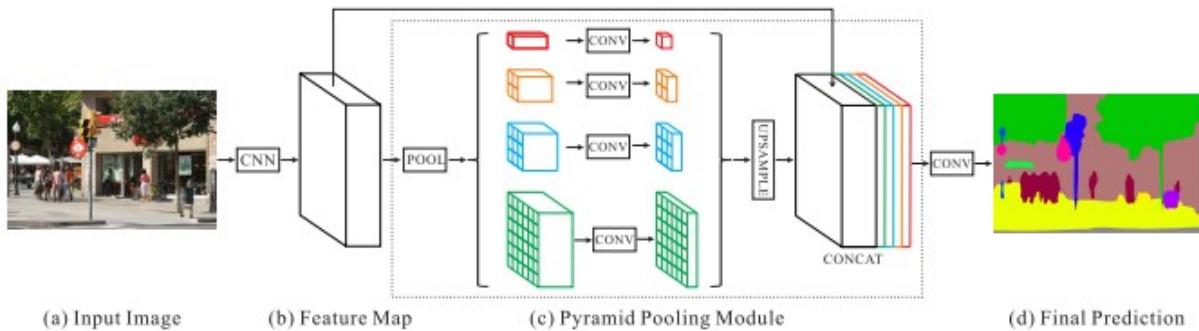


Figura 20: Estrutura de PSPNet [<https://arxiv.org/pdf/1612.01105.pdf>].

Nessa rede (figura 20), a imagem primeiro passa por camadas convolucionais, muitas vezes usando “dilated convolutions” (figura 21) para extrair mapa de atributos com campo de visão ampla. Convolução dilatada é uma técnica que expande o kernel inserindo furos entre seus elementos consecutivos, de modo a cobrir uma área maior da entrada sem aumentar o número de parâmetros. Em Keras, é possível especificar o parâmetro *dilation_rate* na camada Conv2D para fazer convolução dilatada:

```
out = Conv2D(10, (3, 3), dilation_rate=2)(input_tensor)
```

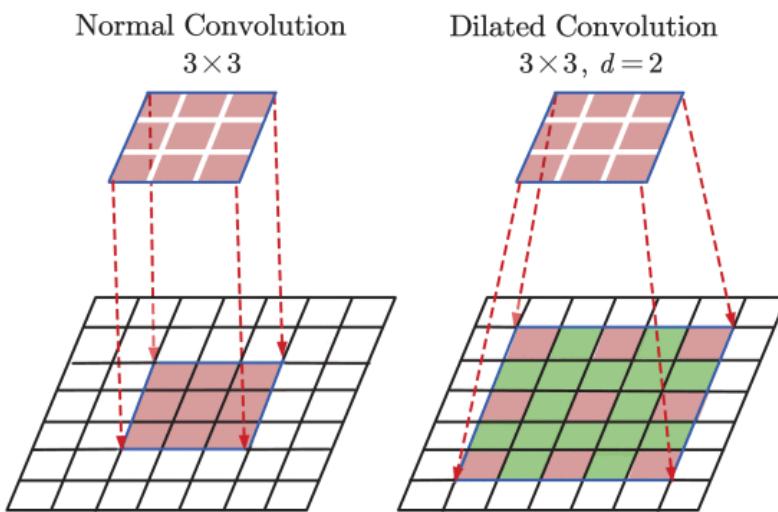


Figura 21: Convolução dilatada [<https://ieeexplore.ieee.org/document/8967055>]

Depois, os mapas de atributos resultantes passam por “average poolings” que diminuem as resoluções de mapas de atributos para diferentes tamanhos, formando uma estrutura em pirâmide tipicamente usada para processamento multi-escala. Convoluções são aplicadas nesses mapas redimensionados. Depois, os mapas da pirâmide aumentam de tamanho (usando upsampling2d), para que todos voltem a ficar com o mesmo tamanho original, antes de concatená-los junto com os atributos

nas escalas originais. Estes mapas concatenados passam por mais convoluções para gerar a saída segmentada final.

Neste modelo, os objetos pequenos são detectados pelas características em alta resolução, enquanto que os objetos grandes são detectados pelas características em baixa resolução.

5.2 PSPNet pré-treinada

O blog abaixo apresenta PSPNet implementada “do zero” em Keras:

<https://medium.com/analytics-vidhya/semantic-segmentation-in-pspnet-with-implementation-in-keras-4843d05fc025>

Em vez de estudar detalhadamente uma implementação “do zero”, vamos usar uma PSPNet pré-treinada. A biblioteca “keras-segmentation” possui PSPNet pré-treinada pronta para o uso:

<https://divamgupta.com/image-segmentation/2019/06/06/deep-learning-semantic-segmentation-keras.html>

<https://github.com/divamgupta/image-segmentation-keras>

Não é possível executar a biblioteca original em versões recentes do TensorFlow/Keras, incluindo o Google Colab, por problemas de compatibilidade das versões. Fiz pequenas alterações para que a biblioteca possa ser executada em versões recentes do TensorFlow/Keras e em Colab (17/09/2023).

A versão modificada está em:

<https://github.com/haeyongkimbr-usp/image-segmentation-keras/blob/master/README.md>

Essa versão modificada pode ser baixada e instalada com o comando:

\$ pip3 install --upgrade git+<https://github.com/haeyongkimbr-usp/image-segmentation-keras.git>

Deixei o programa-exemplo que usa essa biblioteca em:

https://colab.research.google.com/drive/1HQctTuFoUN5lhW_3A2ESq7BTurvr2K7a?usp=chrome_ntp#scrollTo=RpBOJYC9Oc1N



```
1 # pspnet2.py
2 # !pip3 install --upgrade git+https://github.com/haeyongkimbr-usp/image-segmentation-keras.git
3
4 url='http://www.lps.usp.br/hae/apostila/segment.zip'
5 import os; nomeArq=os.path.split(url)[1]
6 if not os.path.exists(nomeArq):
7     print("Baixando o arquivo",nomeArq,"para diretorio default",os.getcwd())
8     os.system("wget -nc -U 'Firefox/50.0' "+url)
9 else:
10    print("O arquivo",nomeArq,"ja existe no diretorio default",os.getcwd())
11 print("Descompactando arquivos novos de",nomeArq)
12 os.system("unzip -u "+nomeArq)
```

Célula para baixar as imagens

https://colab.research.google.com/drive/1HQctTuFoUN5lhW_3A2ESq7BTurvr2K7a?usp=chrome_ntp#scrollTo=EaCjmn6YEfcn

```

1 # pspnet2.py
2 # !pip3 install --upgrade git+https://github.com/haeyongkimbr-usp/image-segmentation-keras.git
3
4 url='http://www.lps.usp.br/hae/apostila/segment.zip'
5 import os; nomeArq=os.path.split(url)[1]
6 if not os.path.exists(nomeArq):
7     print("Baixando o arquivo",nomeArq,"para diretorio default",os.getcwd())
8     os.system("wget -nc -U 'Firefox/50.0' "+url)
9 else:
10    print("O arquivo",nomeArq,"ja existe no diretorio default",os.getcwd())
11 print("Descompactando arquivos novos de",nomeArq)
12 os.system("unzip -u "+nomeArq)
13
14 import os; os.environ['TF_CPP_MIN_LOG_LEVEL']= '3'
15 os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
16 from keras_segmentation.pretrained import pspnet_101_voc12
17 from matplotlib import pyplot as plt
18 import matplotlib
19 import cv2
20
21 model = pspnet_101_voc12() # load the pretrained model trained on Pascal VOC 2012 dataset
22
23 nome="chimpanze"
24 out = model.predict_segmentation(
25     inp=nome+".jpg",
26     out_fname=nome+"-voc.png"
27 )
28
29 im=cv2.imread(nome+".jpg",1); im=cv2.cvtColor(im, cv2.COLOR_BGR2RGB)
30 out=cv2.imread(nome+"-voc.png",1); out=cv2.cvtColor(out, cv2.COLOR_BGR2RGB)
31 print(im.shape); print(out.shape)
32
33 f = matplotlib.pyplot.gcf(); f.set_size_inches(6, 12)
34 f.add_subplot(1,2,1); plt.imshow(im); plt.axis("off");
35 f.add_subplot(1,2,2); plt.imshow(out); plt.axis("off");
36
37 plt.show()

```

Programa 9: Aplica PSPNet pré-treinada na imagem “chimpanze.jpg”, gerando “chimpanze-voc.png”.

https://colab.research.google.com/drive/1HQctTuFoUN5lhW_3A2ESq7BTurvr2K7a?usp=chrome_ntp#scrollTo=EaCjmn6YEfcn

Algumas imagens segmentadas com esse programa (disponíveis em: <http://www.lps.usp.br/hae/apostila/segment.zip>) para diferentes finalidades:

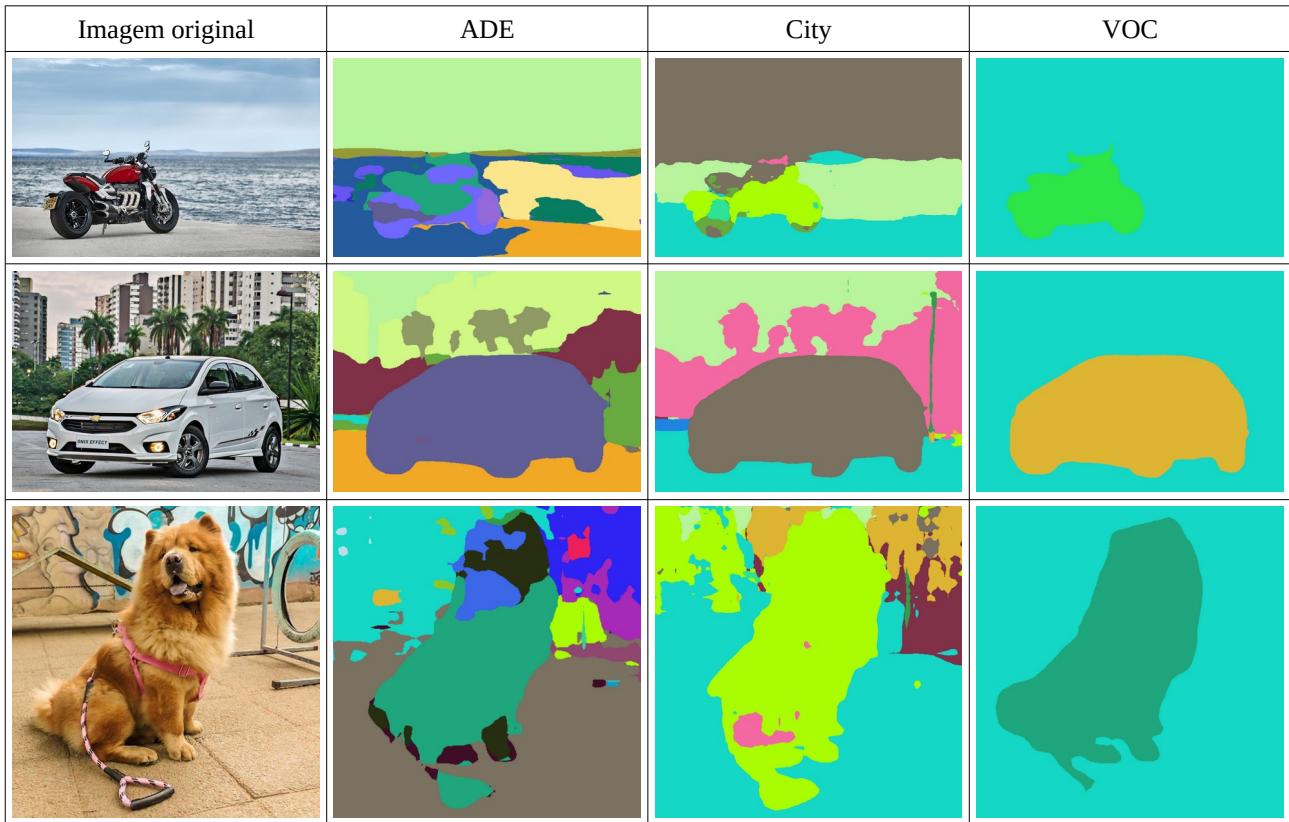


Figura 22: Algumas imagens segmentadas usando PSPNet pré-treinada com diferentes bancos de dados.

Veja como VOC conseguiu segmentar com sucesso os animais/objetos principais das imagens. A biblioteca traz 3 modelos treinados nos seguintes conjuntos de imagens:

ADE20K: <https://groups.csail.mit.edu/vision/datasets/ADE20K/>

```
model = pspnet_50_ADE_20K() # load the pretrained model trained on ADE20k dataset
```

Conjunto de imagens para segmentar objetos comuns do dia a dia.

Cityscapes: <https://www.cityscapes-dataset.com/>

```
model = pspnet_101_cityscapes() # load the pretrained model trained on Cityscapes dataset
```

Conjunto de imagens para segmentar cenas de cidades.

Pascal VOC data sets: <http://host.robots.ox.ac.uk/pascal/VOC/>

```
model = pspnet_101_voc12() # load the pretrained model trained on Pascal VOC 2012 dataset
```

Conjunto de imagens para classificar imagens em categorias, juntamente com segmentação do objeto principal.

Exercício: Os exemplos da figura 22 ilustram bem como funciona o modelo de segmentação VOC, mas não mostram muito claramente as propriedades dos modelos treinados em ADE20K e Cityscapes. Escolha 2 imagens apropriadas para serem segmentadas pelo modelo ADE20K e 2 imagens apropriadas para o modelo Cityscapes e segmente-as, para caracterizar bem esses dois modelos. Coloque essas imagens em edisciplinas, para eu poder usá-las como exemplos nas aulas dos próximos anos.

6. Referências

Vale a pena testar o site “segment anything”:

<https://segment-anything.com/>

permite segmentação iterativa via browser.

Blogs que resumem as principais técnicas de segmentação:

<https://neptune.ai/blog/image-segmentation>

<https://www.v7labs.com/blog/semantic-segmentation-guide>

Tutoriais de segmentação em Keras:

<https://keras.io/examples/vision/segformer/>

https://keras.io/examples/vision/deeplabv3_plus/

<https://www.tensorflow.org/tutorials/images/segmentation>

https://www.tensorflow.org/tfmodels/vision/semantic_segmentation

<https://www.tensorflow.org/lite/examples/segmentation/overview>

[WikiImageNet] <https://en.wikipedia.org/wiki/ImageNet>

[Li2017-lecture11] http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture11.pdf

[Ronneberger2015] Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. "U-net: Convolutional networks for biomedical image segmentation." *International Conference on Medical image computing and computer-assisted intervention*. Springer, Cham, 2015. <https://arxiv.org/abs/1505.04597>

[Zeger2021] I. Žege, S. Grgic, J. Vuković and G. Šišul, "Grayscale Image Colorization Methods: Overview and Evaluation," in IEEE Access, vol. 9, pp. 113326-113346, 2021, doi: 10.1109/ACCESS.2021.3104515. <https://ieeexplore.ieee.org/document/9512069>

[PSI5790 aula 9 parte 2. Fim]