[PSI3472-2023. 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 2. 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.

Redes semelhantes às utilizadas para fazer segmentação semântica podem ser usadas para colorir imagens em níveis de cinza [Zeger2021].

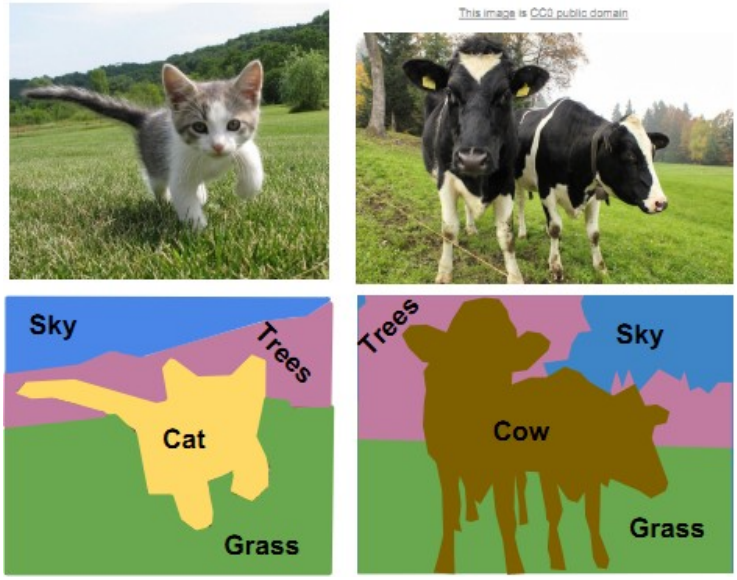


Figura 2: Segmentação semântica (de [Li2017-lecture11]).

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 3), sempre centralizados no centro da imagem. Todas as imagens têm dimensão 32×32 e vem em pares (*X*, *Y*): imagem de entrada e respectiva saída com 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 OU
https://drive.google.com/drive/folders/1qtOVdWNaxnro-SDUBpjiFJFB-PcPop7t?usp=sharing

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.

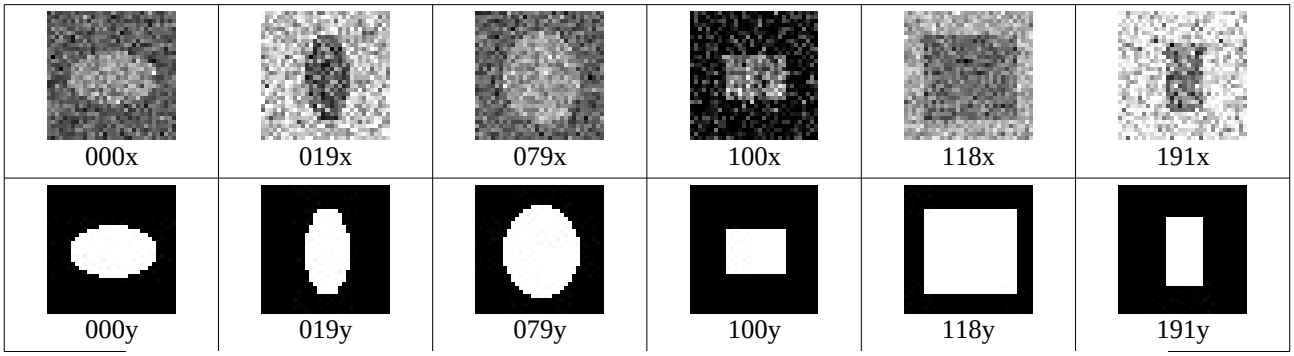


Figura 3: Banco de imagens *eliret* para testar técnicas de segmentação semântica.

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 4). 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 "grama", 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.

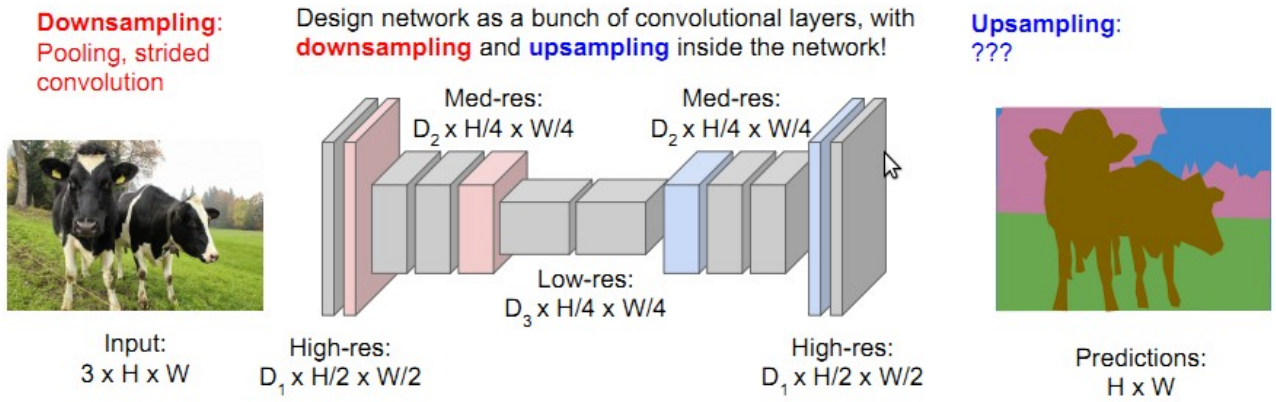


Figura 4: Estrutura de uma rede FCN para segmentação semântica (de [Li2017-lecture11]).

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

Em Keras, (aparentemente) só há um tipo de camada “unpooling” denominado *UpSampling2D* (figura U). 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/

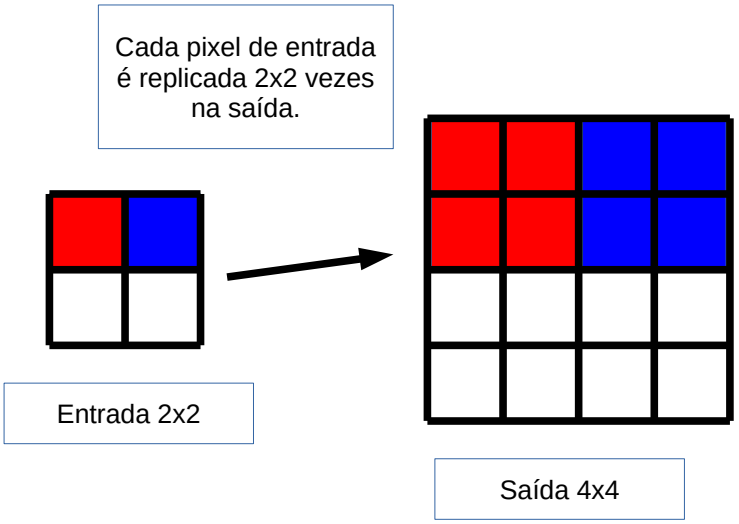


Figura U: UpSampling2D de Keras.

2) 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 (se usar “padding same”). Aumentando o passo para 2, a camada convolucional irá diminuir a resolução da imagem por 2 (usando “padding 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 (usando “padding 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). Como disse, 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.

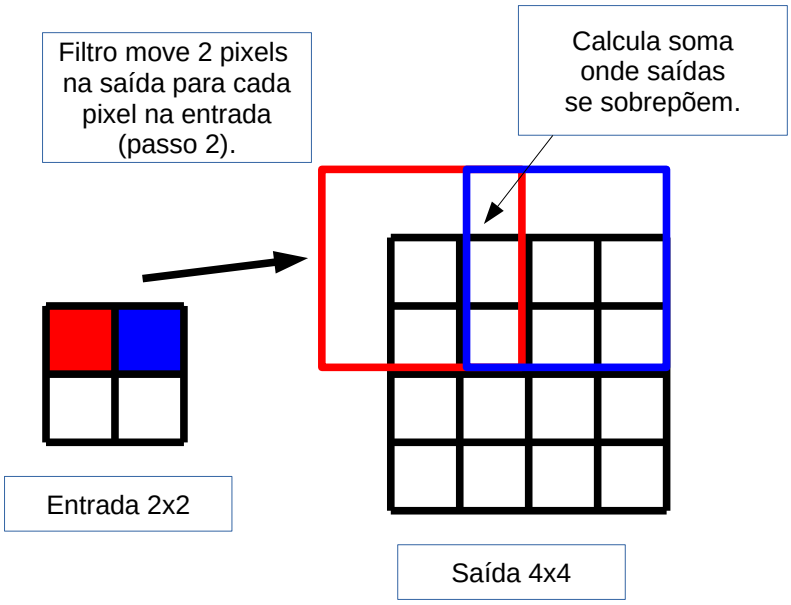


Figura 5: Convolução transposta 3×3, passo 2.

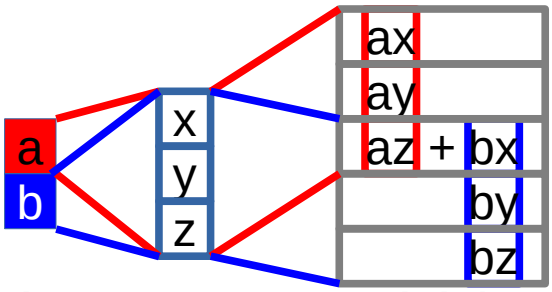


Figura 6: Convolução transposta 1D com janela de dimensão 3, passo 2.

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

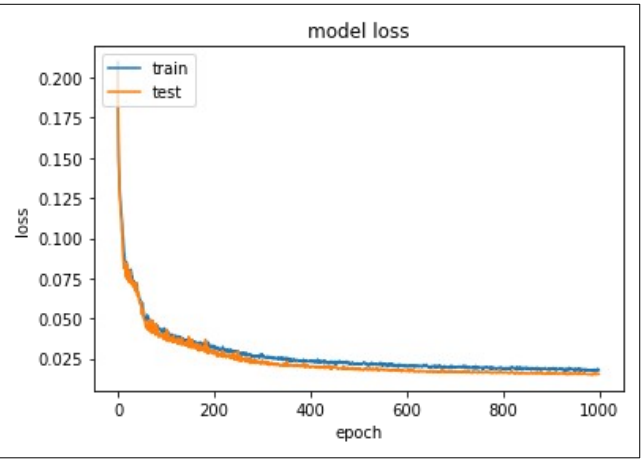
1.3 Implementação de FCN para segmentar elipses e retângulos

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 2) treina FCN para segmentar elipses e retângulos ruidosos. A estrutura da rede está mostrada na figura 7. O erro MSE obtido, após 1000 épocas de treinamento, é:

```
Training loss: 0.009817
Validation loss: 0.015401
Test loss: 0.015854
```



No programa 2, 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×32×32×1, 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.

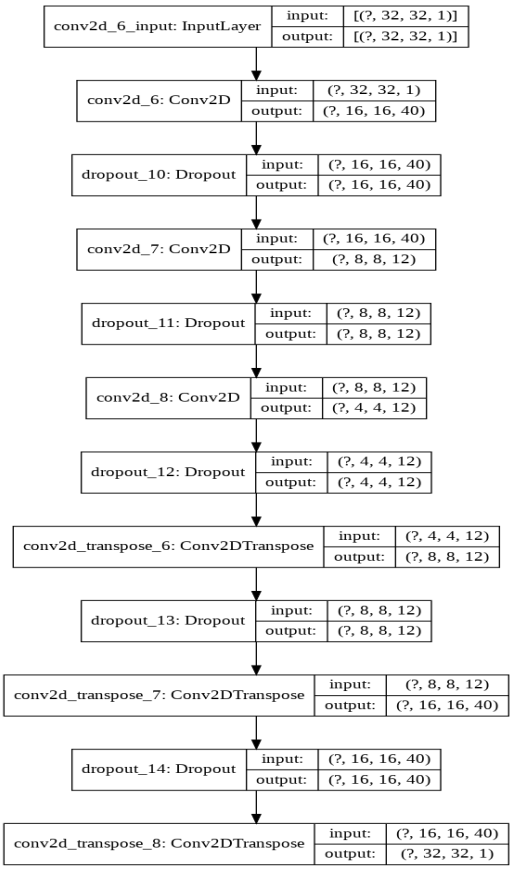
Veja a figura 8, onde as colunas *a* e *b* mostram imagem de entrada e saída ideal e as colunas *c* e *d* mostram as saídas de FCN. A saída original de FCN é imagem em níveis de cinza (coluna *c*). Ela foi limiarizada (coluna *d*) para facilitar a comparação com a saída ideal. Pelas saídas, pode-se afirmar 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 figura que está tentando segmentar é elipse ou retângulo, o que faz com que elipse fique retangular e retângulo fique elipsoidal.

Layer (type)	Output Shape	Param #
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

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.

```python
#fcn-train1.py
#Treina rede fcn para segmentacao semantica de eliret
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 Sequential
from tensorflow.keras.layers import Dropout, Conv2D, Conv2DTranspose
from tensorflow.keras import optimizers
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

#<<<<<<<<<<<< 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); batch_size = 20; epochs = 1000

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(Dropout(0.25))
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(Dropout(0.25))
model.add(Conv2DTranspose(12, kernel_size=(5,5), strides=2, activation='relu', padding='same')) #saida 8*8*12
model.add(Dropout(0.25))
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-train1.png', show_shapes=True)
model.summary()

opt=optimizers.Adam()
model.compile(optimizer=opt, loss='mean_squared_error')
history=model.fit(ax, ay, batch_size=batch_size, epochs=epochs, verbose=0, 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-train1.h5')
```

Programa 2: Segmentação de elipses e retângulos usando FCN.
https://colab.research.google.com/drive/1RUc1fE1daB1zBox7pe8-of2os_wKHqCv?usp=sharing

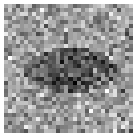
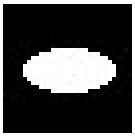
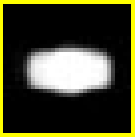
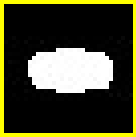
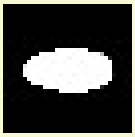
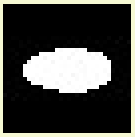
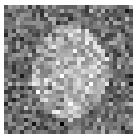
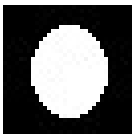
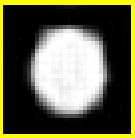
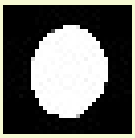
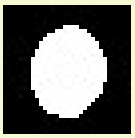
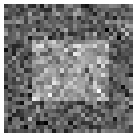
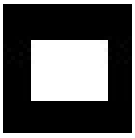
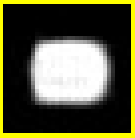
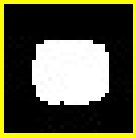
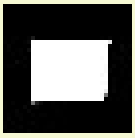
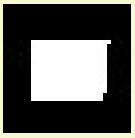
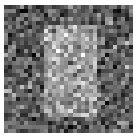
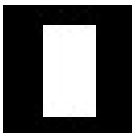
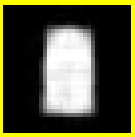
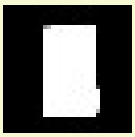
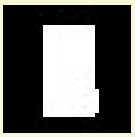
	a	b	c	d	e	f
	entrada teste QX	saída teste ideal QY	saída FCN níveis de cinza	saída FCN binari-zada	saída UNET níveis de cinza	saída UNET binari-zada
076						
078						
176						
178						

Figura 8: Algumas imagens de teste segmentadas pela FCN (amarelo) e U-net (verde).

O programa fcn-pred1.py (programa 3) usa a rede gerada pelo programa fcn-train1.py (programa 2) para segmentar a imagem de elipse ou retângulo ruidosa.

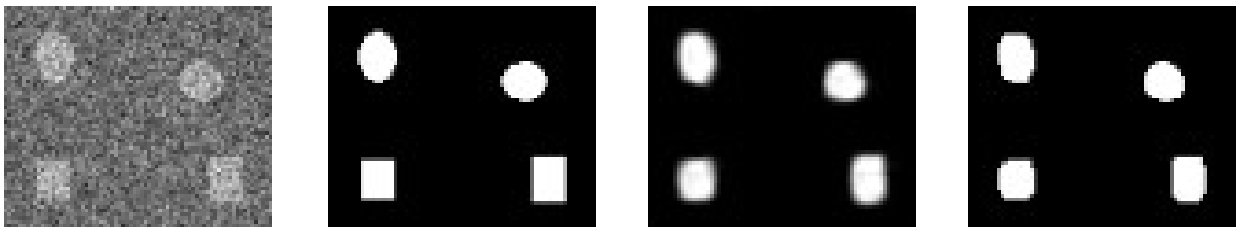
A variável "***nome***" especifica a imagem que se deseja segmentar ("***nome***x.png").

```python
#fcn-pred1.py
#Faz segmentacao semantica usando rede gerada pelo fcn-train1.py
import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
import cv2; import numpy as np
import tensorflow.keras as keras
from tensorflow.keras.models import load_model
from tensorflow.keras.layers import Dropout, Conv2D, Conv2DTranspose
from tensorflow.keras import optimizers
import sys; from sys import argv

#<<<<<<<<<<<<<<<<<<< main <<<<<<<<<<<<<<<<<<<<<<
bdDir = "."
outDir = "."; os.chdir(outDir)
nome="077"
inImgX = nome+"x.png"; inImgY = nome+"y.png"
outImgG =nome+"g.png"; outImgB =nome+"b.png"
arquivoRede = "fcn-train1.h5"

model = load_model(os.path.join(outDir,arquivoRede))
QX=cv2.imread(os.path.join(bdDir,inImgX),0)
QY=cv2.imread(os.path.join(bdDir,inImgY),0)
nl=QX.shape[0]; nc=QX.shape[1]
qx=np.float32(QX)/255.0 #Entre 0 e +1
qx=qx.reshape(1, nl, nc, 1)

qp=model.predict(qx); qp=qp.reshape(nl,nc) # entre 0 e +1

QPG=255.0*qp; QPG=np.clip(QPG,0,255) # Entre 0 e 255
QPG=np.uint8(QPG); cv2.imwrite(os.path.join(outDir,outImgG),QPG)

QPB=np.zeros((nl,nc),dtype='uint8'); QPB[ qp>=0.5 ] = 255
cv2.imwrite(os.path.join(outDir,outImgB),QPB)

from matplotlib import pyplot as plt
f = plt.figure()
f.add_subplot(1,4,1); plt.imshow(QX,cmap="gray"); plt.axis('off')
f.add_subplot(1,4,2); plt.imshow(QY,cmap="gray"); plt.axis('off')
f.add_subplot(1,4,3); plt.imshow(QPG,cmap="gray"); plt.axis('off')
f.add_subplot(1,4,4); plt.imshow(QPB,cmap="gray"); plt.axis('off')
plt.show(block=True)
```

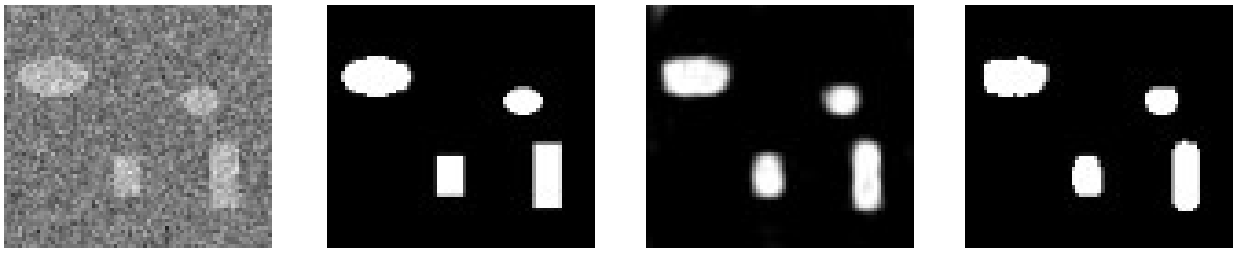
Programa 3: Segmenta elipse/retângulo usando FCN e mostra imagem segmentada.
https://colab.research.google.com/drive/1RUc1fE1daB1zBox7pe8-of2os_wKHqCv?usp=sharing

Propriedade das redes completamente convolucionais

Vimos (apostila cifar-ead) que as redes convolucionais "modernas" conseguem classificar imagens de qualquer tamanho, 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. Veja a figura X que mostra segmentação das imagens *maior1x.png* e *maior2x.png* com resolução de 72×88 pi 80×88 pixels usando a rede neural treinada em imagens 32×32. Como é possível segmentar imagens maiores do que a resolução para a qual a rede foi treinada?



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

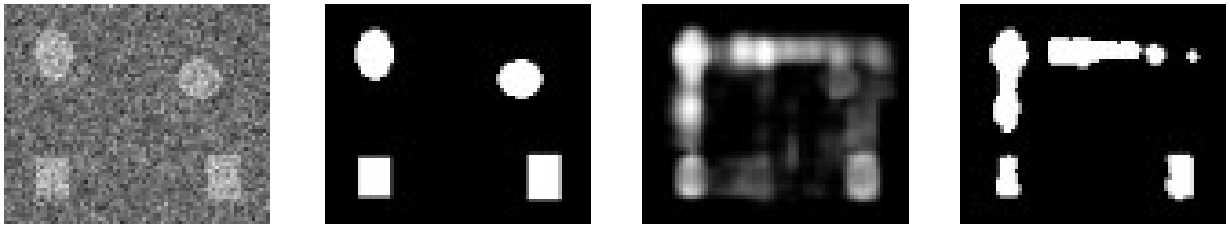
Note que uma convolução (por exemplo, 5×5) pode ser aplicada a qualquer imagem com tamanho maior igual ao tamanho do núcleo (no exemplo, ≥5×5). O mesmo vale para convolução transposta e para dropout. Assim, uma rede convolucional que não tem camadas densas pode receber como entrada imagens de qualquer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

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*×*nc*=72×88 de forma que *nl* e *nc* sejam divisíveis por 2^3=8. Para poder aplicar o modelo *fcn-train1.h5* 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 vieses exatamente igual à rede original. Depois, devemos carregar os pesos da rede treinada *fcn-train1.h5* no novo modelo usando o método:

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

Fazendo isso e processando a imagem *maior1x.png*, obtemos a seguinte saída:



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

Figura Y: Segmentação de *maior1x.png* usando o modelo *fcn-train1.h5* gerado pelo programa 2.

O programa foi executado sem erro, mas a segmentação da figura Y 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*.

Os programas X e Y fazem data augmentation usando *ImageDataGenerator* com deslocamento horizontal e vertical aleatório de ±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"}.
```

Com isso, foi possível obter as segmentações corretas mostradas na figura X (apesar de que 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.

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

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

```
#fcn_da1.py
#Treina rede fcn para segmentacao semantica de eliret
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 X: https://colab.research.google.com/drive/1PWyJ4tKHls5WliQxv-0iT0UaS4gSPrti?usp=sharing

```
#fcn_da1_pred.py
#Faz segmentacao semantica usando rede gerada pelo fcn-train1.py
import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
import cv2; import numpy as np
import tensorflow.keras as keras
from tensorflow.keras.models import load_model
from tensorflow.keras.layers import Dropout, Conv2D, Conv2DTranspose
from tensorflow.keras import optimizers
import sys; from sys import argv

#<<<<<<<<<<<<<<<<<<< main <<<<<<<<<<<<<<<<<<<<<<
bdDir = "."
outDir = "."; os.chdir(outDir)
nome="maior2"
inImgX = nome+"x.png"; inImgY = nome+"y.png"
outImgG =nome+"g.png"; outImgB =nome+"b.png"
arquivoRede = "fcn_da1.h5"

QX=cv2.imread(os.path.join(bdDir,inImgX),0)
QY=cv2.imread(os.path.join(bdDir,inImgY),0)
nl=QX.shape[0]; nc=QX.shape[1]
qx=np.float32(QX)/255.0 #Entre 0 e +1
qx=qx.reshape(1, nl, nc, 1)

model = fcn((nl,nc,1))
model.load_weights(os.path.join(outDir,arquivoRede))

qp=model.predict(qx)
qp=qp.reshape(nl,nc) # entre 0 e +1

QPG=255.0*qp; QPG=np.clip(QPG,0,255) # Entre 0 e 255
QPG=np.uint8(QPG);
cv2.imwrite(os.path.join(outDir,outImgG),QPG)

QPB=np.zeros((nl,nc),dtype='uint8'); QPB[ qp>=0.5 ] = 255
cv2.imwrite(os.path.join(outDir,outImgB),QPB)

from matplotlib import pyplot as plt
f = plt.figure()
f.add_subplot(1,4,1); plt.imshow(QX,cmap="gray"); plt.axis('off')
f.add_subplot(1,4,2); plt.imshow(QY,cmap="gray"); plt.axis('off')
f.add_subplot(1,4,3); plt.imshow(QPG,cmap="gray"); plt.axis('off')
f.add_subplot(1,4,4); plt.imshow(QPB,cmap="gray"); plt.axis('off')
plt.show(block=True)
```

Programa Y: https://colab.research.google.com/drive/1PWyJ4tKHls5WliQxv-0iT0UaS4gSPrti?usp=sharing

[PSI3472-2023. Aula 9. Fim.]

[PSI3472-2023. Aula 10. Início.]

1.4 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

Vimos acima que FCN não conseguiu localizar de forma precisa as fronteiras entre as regiões. Note que, para fazer upsampling, FCN utiliza 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 [Ronnenberger2015] resolve este problema passando para as camadas de *upsampling* as informações das camadas de *downsampling* com a mesma resolução (figuras 9 e 10). 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*.

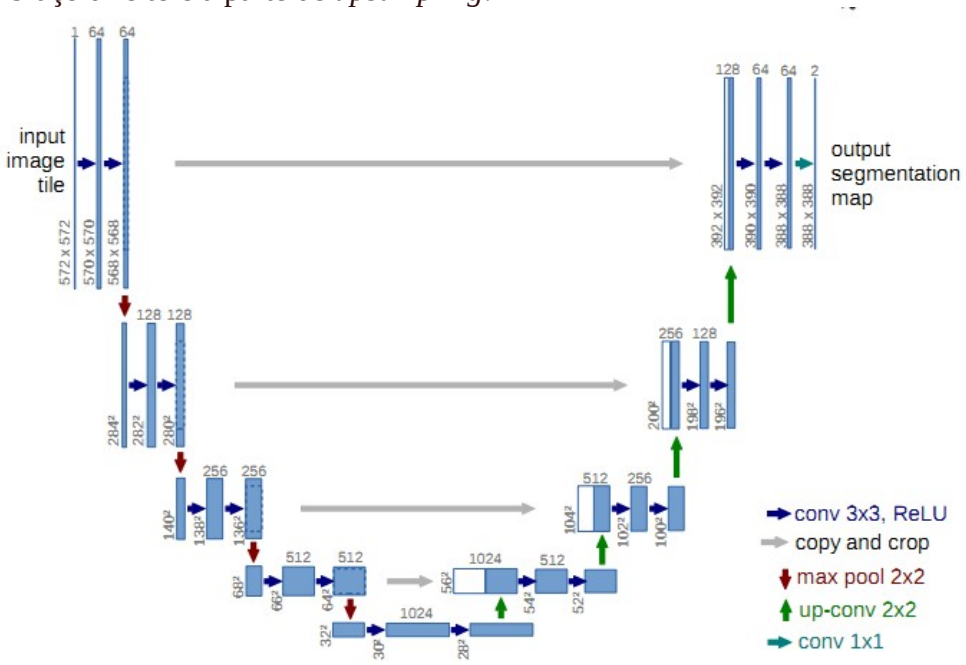


Figura 9: U-Net original.

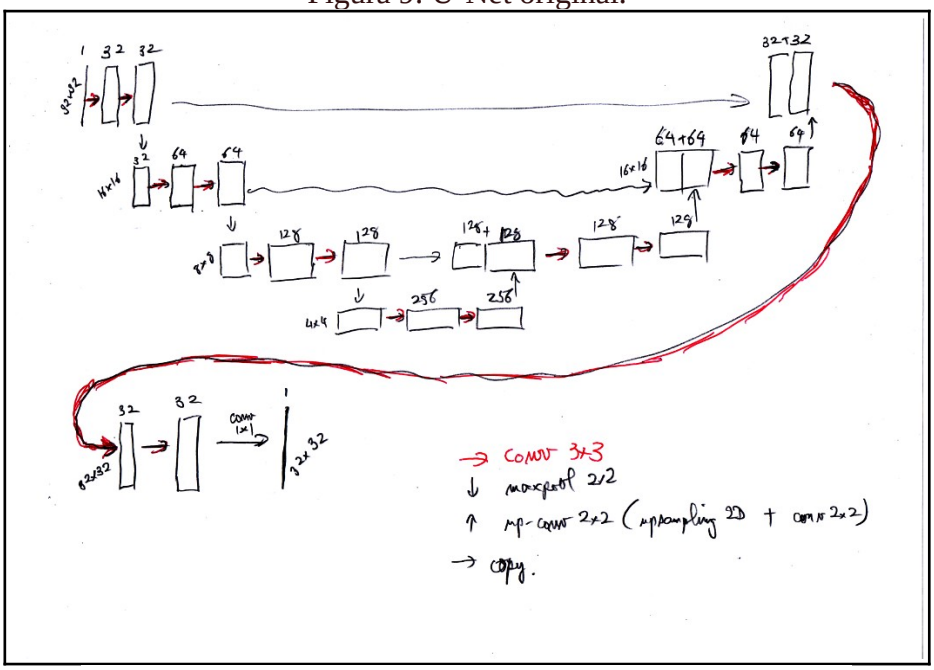


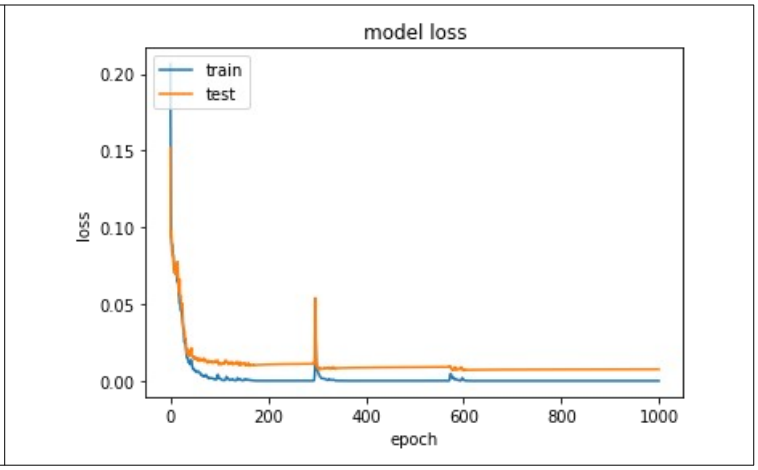
Figura 10: U-Net usada para segmentar elipses e retângulos.

U-Nets das figuras 9 e 10 possuem semelhanças com a rede FCN (figura 4): 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.

1.5 U-Net para segmentar elipses e retângulos

O programa unet-train1 (programa 4) treina U-Net para segmentar elipses e retângulos ruidosos. A estrutura da rede está mostrada na figura 11. O erro MSE obtido, após 1000 épocas de treinamento, é:

```
Training loss: 1.07136e-08
Validation loss: 0.007563
Test loss: 0.007492
```



Isto é, os erros de validação e de teste (0.0075) são aproximadamente a metade do que obtivemos 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*.
Nota: Talvez seja interessante colocar dropout ou regularização L2 na rede para diminuir overfitting.

No *unet-train1.py* (programa 4), a função *leCsv* (linhas 17-32) é idêntica à mesma função do programa 2. Também as leituras (nas linhas 80-84) são idênticas ao programa 2. Da mesma forma que no programa 2 (*fcn-train1.py*), aqui também normalizamos a entrada para 0 a 1. Por algum motivo desconhecido, é impossível treinar esta rede se normalizar a entrada para -1 a +1.

Exercício: Teste inserir BatchNormalization após camadas convolucionais. Possivelmente, a rede irá convergir mesmo com entrada de -1 a +1.

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 10 e a estrutura da rede está desenhada na figura 11. Note que, diferentemente de FCN, estamos usando MaxPooling2D para diminuir resolução e UpSampling2D para aumentar resolução.

A figura 8, colunas *e* e *f*, 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 consegue decidir a saída com maior confiança. Isto não acontece com as saídas da FCN.

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]

Total params: 1,925,025
Trainable params: 1,925,025
Non-trainable params: 0

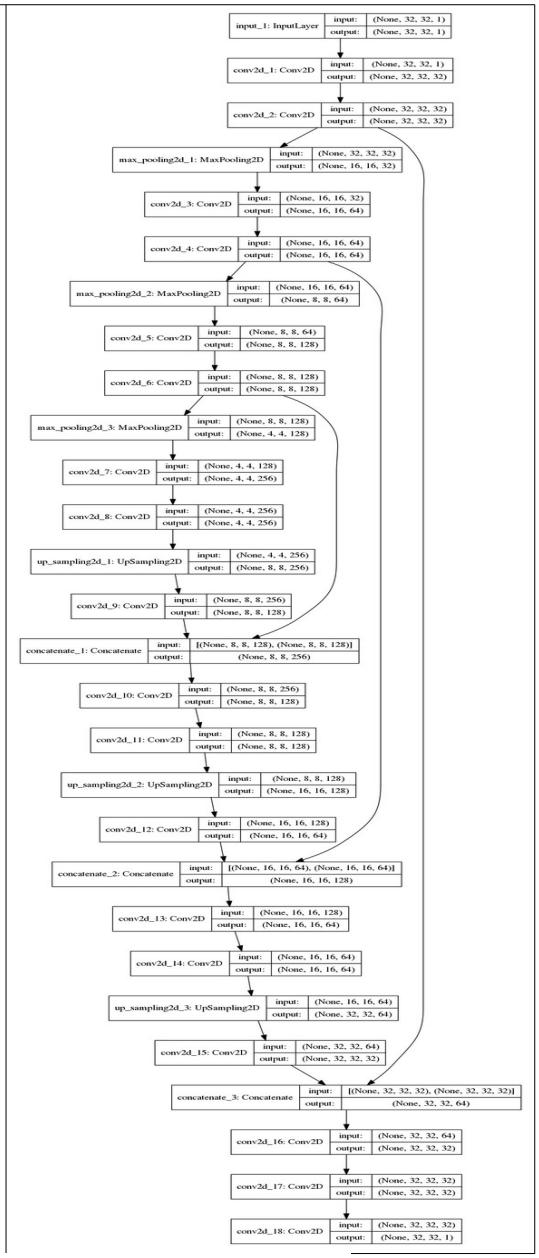


Figura 11: Estrutura da rede U-Net para segmentar elipses e retângulos.

```python
#unet-train1.py
#Treina rede unet para segmentacao semantica de eliret
import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
import cv2; import numpy as np; np.random.seed(7); import sys
import tensorflow.keras as keras; from tensorflow.keras.models import *
from tensorflow.keras.layers import *; from tensorflow.keras.optimizers import *
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 unet(input_size = (32,32,1)):
  n=32
  inputs = Input(input_size) #32x32
  conv2 = Conv2D(n, 3, activation = 'relu', padding = 'same' )(inputs)
  conv2 = Conv2D(n, 3, activation = 'relu', padding = 'same' )(conv2)
  pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)  #16x16

  conv3 = Conv2D(2*n, 3, activation = 'relu', padding = 'same' )(pool2) #16x16
  conv3 = Conv2D(2*n, 3, activation = 'relu', padding = 'same' )(conv3)
  pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)  #8x8

  conv4 = Conv2D(4*n, 3, activation = 'relu', padding = 'same' )(pool3) #8x8
  conv4 = Conv2D(4*n, 3, activation = 'relu', padding = 'same' )(conv4)
  pool4 = MaxPooling2D(pool_size=(2, 2))(conv4)  #4x4

  conv5 = Conv2D(8*n, 3, activation = 'relu', padding = 'same' )(pool4) #4x4
  conv5 = Conv2D(8*n, 3, activation = 'relu', padding = 'same' )(conv5) #4x4

  up6 = Conv2D(4*n, 2, activation = 'relu', padding = 'same'
               )(UpSampling2D(size = (2,2))(conv5)) #8x8
  merge6 = concatenate([conv4,up6], axis = 3) #8x8
  conv6 = Conv2D(4*n, 3, activation = 'relu', padding = 'same' )(merge6)
  conv6 = Conv2D(4*n, 3, activation = 'relu', padding = 'same' )(conv6) #8x8

  up7 = Conv2D(2*n, 2, activation = 'relu', padding = 'same'
               )(UpSampling2D(size = (2,2))(conv6)) #16x16
  merge7 = concatenate([conv3,up7], axis = 3)
  conv7 = Conv2D(2*n, 3, activation = 'relu', padding = 'same' )(merge7)
  conv7 = Conv2D(2*n, 3, activation = 'relu', padding = 'same' )(conv7) #16x16

  up8 = Conv2D(n, 2, activation = 'relu', padding = 'same'
               )(UpSampling2D(size = (2,2))(conv7)) #32x32
  merge8 = concatenate([conv2,up8], axis = 3)
  conv8 = Conv2D(n, 3, activation = 'relu', padding = 'same' )(merge8)
  conv8 = Conv2D(n, 3, activation = 'relu', padding = 'same' )(conv8) #32x32

  conv9 = Conv2D(1, 1, activation = 'sigmoid', padding = 'same' )(conv8) #32x32

  model = Model(inputs = inputs, outputs = conv9)
  model.compile(optimizer = Adam(learning_rate=1e-3), loss = 'mean_squared_error')
  from tensorflow.keras.utils import plot_model
  plot_model(model, to_file='unet-train1.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)

#Escolha entre comecar treino do zero ou continuar o treino de onde parou
model=unet()
#model = load_model("unet1.h5");

history=model.fit(ax, ay, batch_size=10, epochs=1000, verbose=2, validation_data=(vx,vy));
impHistoria(history); model.save("unet1.h5");

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)
```

Programa 4: U-Net usada para segmentar elipses e retângulos.
https://colab.research.google.com/drive/1DfixQxjDNaSQkQvihHA4v_UBSjjeZG8H?usp=sharing

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.

[PSI3472-2023. Lição de casa aulas 9/10.]
O objetivo deste exercício é verificar quais alterações devemos introduzir na FCN (programas 2 e 3) para que fique com a qualidade de segmentação semelhante a U-Net, sem trocar as convoluções e convoluções transpostas (com strides=2) por maxpooling2d e upsampling2d. Ou seja, vamos construir U-Net usando convolução e convolução transposta com strides=2.

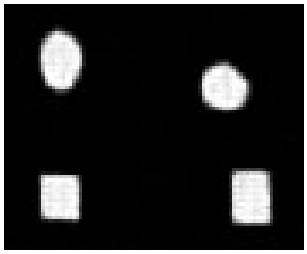
a) Reescreva os programas 2 e 3 desta apostila (*fcn-train1.py* e *fcn-pred1.py*) usando API funcional. Verifique se continua funcionando da mesma forma que antes (precisa obter MSE de validação/teste de aproximadamente 0.016). Segmente alguma imagem teste e mostre.
Solução privada em ~/deep/algpi/segment/licao/fcn-train2.py e fcn-pred2.py.

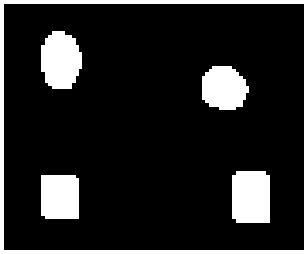
b - ex1 vale 5) Acrescente as ligações que atravessam os dois braços de "U" da rede FCN (como na U-Net). Fazendo isso e ajustando os números de convoluções das camadas como na U-Net (programa 4), consegui chegar a MSE de validação/teste 0.012. Segmente alguma imagem teste e mostre.
Solução privada em ~/deep/algpi/segment/licao/unet-train2.py.

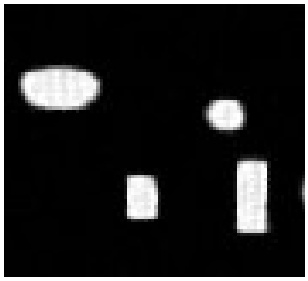
c -ex2 vale 5) Para o modelo ficar mais parecido com U-Net, substitua todas convoluções e convoluções transpostas 5×5 por sequências de duas convoluções 3×3. Por exemplo, substitua:
```
x2 = Conv2D(2*n, kernel_size=(5,5), strides=2, activation='relu', padding='same')(x2)
```
por:
```
x2 = Conv2D(2*n, 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.

[PSI3472-2023. Lição de casa extra aulas 9/10. Vale +2.]
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 Z que a qualidade de segmentação é melhor do que usando FCN (figura X). Os retângulos não aparecem mais com cantos arredondados.


maior1x.png segmentada


maior1x.png segmentada e binarizada


maior2x.png segmentada

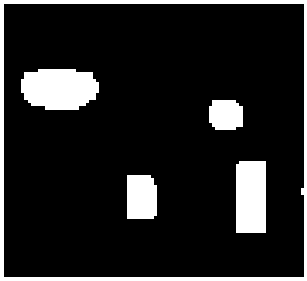

maior2x.png segmentada e binarizada

Figura Z: Maior1x e maior2x segmentadas pela U-Net. Compare com figura X.

Exercício: Coloque camadas "dropout" e/ou regularização L2 na U-Net do programa 4 para diminuir overfitting.

Exercício: Coloque camadas “batch normalization” na U-Net do programa 4 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: 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 5 (unet-pred1.py) utiliza rede U-Net já treinada pelo programa 5 (unet-train1.py) para segmentar elipse ou retângulo.

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 5: Gera imagem de saída da rede U-Net.
https://colab.research.google.com/drive/1DfixQxjDNaSQkQvihHA4v_UBSjjeZG8H?usp=sharing

[PSI3472-2023. Aula 10. Fim.]

2. 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`
Este site, por sua vez, pegou as imagens originais de :
`http://brainiac2.mit.edu/isbi_challenge`

O banco de dados possui 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 banco 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.

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

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

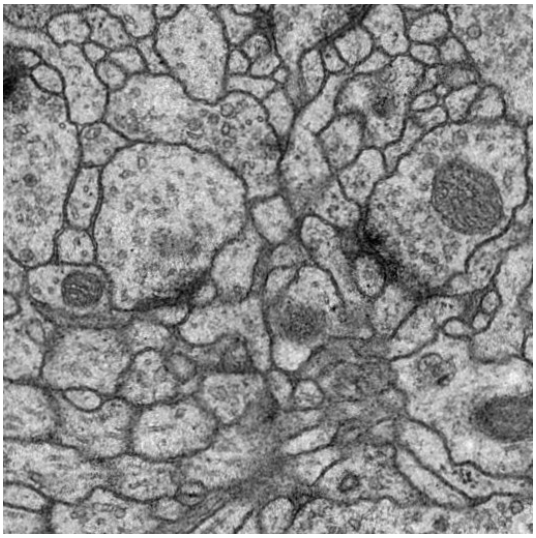
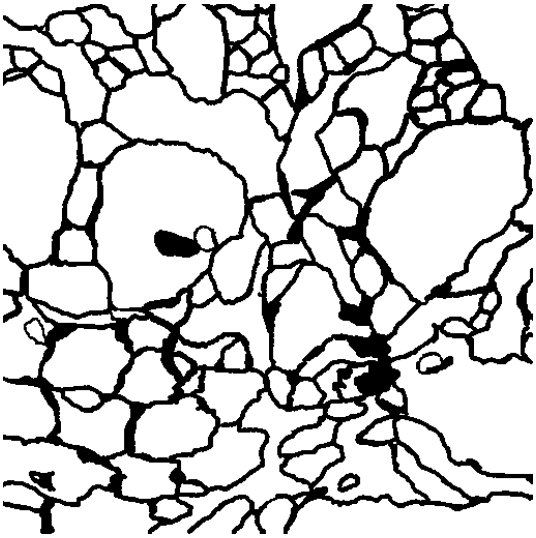
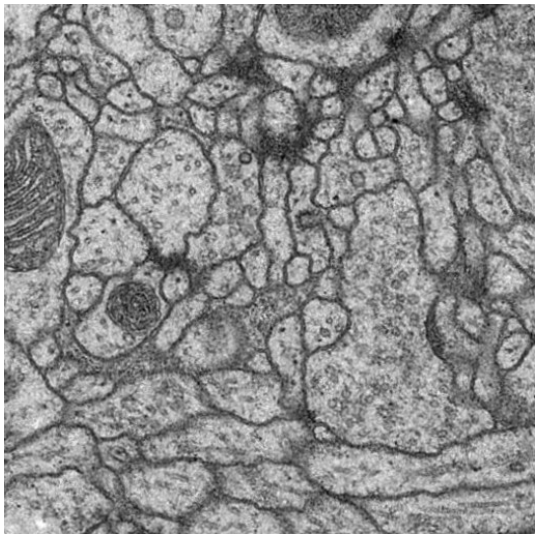
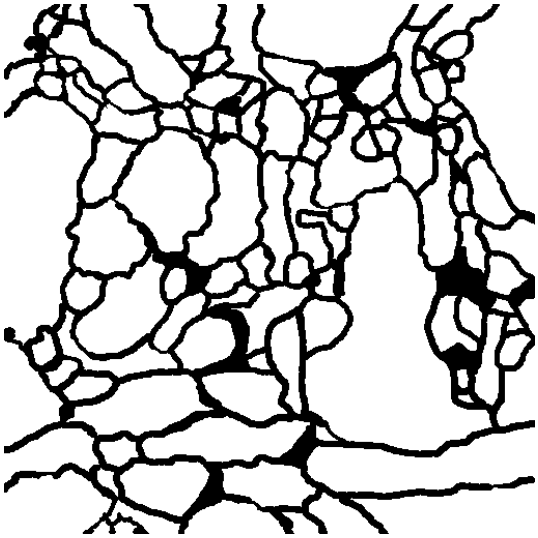
image (AX)	label ou mask (AY)
 0.png	 0.png
 10.png	 10.png

Figura 12: Algumas das 30 imagens de treinamento do BD *membrane*.

2.1 Data augmentation nas imagens de entrada e saída

Para estudarmos os conceitos, sem perdermos muito tempo treinando, vamos trabalhar com as imagens redimensionadas para 128×128 (as originais são 512×512). 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* não deve distorcer somente as imagens de entrada, mas 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 (isto é, efetuam distorções com os mesmos parâmetros). Algumas versões reduzidas e distorcidas de quatro imagens (entrada-original, saída-original, entrada-distorcida e saída-distorcida) estão mostradas na figura 13.

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 ±10°, width_shift_range=0.05 indica que a imagem pode sofrer deslocamento horizontal de ±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).

```python
#membrane_gera1.py
#Reduz imagem de 512x512 para 128x128
#Gera 10 imagens distorcidas para cada imagem de entrada
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.preprocessing.image import ImageDataGenerator;
import numpy as np; import cv2;

#<<<<<<<<<<<<<<< main <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
train_path='./membrane/train';
outDir = "."; os.chdir(outDir)

aug_dict = dict(rotation_range=10, #Int. Degree range for random rotations.
  width_shift_range=0.05, #float: fraction of total width, if < 1, or pixels if >= 1.
  height_shift_range=0.05, #float: fraction of total height, if < 1, or pixels if >= 1.
  shear_range=10, #Float. Shear Intensity (Shear angle in counter-clockwise direction in degrees)
  zoom_range=0.2, #Range for random zoom. If a float, [lower, upper] = [1-zoom_range, 1+zoom_range].
  horizontal_flip=False, #Boolean. Randomly flip inputs horizontally.
  fill_mode='reflect'); #One of {"constant", "nearest", "reflect" or "wrap"}.

image_folder='image'; mask_folder= 'label';
seed = 7; target_size = (128,128); batch_size=2; #Pega aleatoriamente 2 de 30 imagens do diretorio

ori_datagen = ImageDataGenerator() #Imagem sem distorcao
ori_generator = ori_datagen.flow_from_directory(
  train_path, classes = [image_folder], class_mode = None, color_mode = "grayscale",
  target_size = target_size, batch_size = batch_size, seed = seed);

mori_datagen = ImageDataGenerator() #Mascara sem distorcao
mori_generator = mori_datagen.flow_from_directory(
  train_path, classes = [mask_folder], class_mode = None, color_mode = "grayscale",
  target_size = target_size, batch_size = batch_size, seed = seed);

image_datagen = ImageDataGenerator(**aug_dict) #Imagem distorcida com parametros aug_dict
image_generator = image_datagen.flow_from_directory(
  train_path, classes = [image_folder], class_mode = None, color_mode = "grayscale",
  target_size = target_size, batch_size = batch_size, seed = seed);

mask_datagen = ImageDataGenerator(**aug_dict) #Mascara distorcida com parametros aug_dict
mask_generator = mask_datagen.flow_from_directory(
  train_path, classes = [mask_folder], class_mode = None, color_mode = "grayscale",
  target_size = target_size, batch_size = batch_size, seed = seed);

from matplotlib import pyplot as plt
for i in range(1):
  ori=ori_generator.next();   #2 imagens originais
  mori=mori_generator.next(); #2 mascaras originais
  img=image_generator.next(); #gera distorcao nas 2 imagens
  mask=mask_generator.next(); #gera distorcao nas 2 masks
  ori=ori.squeeze(); mori=mori.squeeze(); img=img.squeeze(); mask=mask.squeeze()
  f = plt.figure(figsize=[4,2],dpi=200)
  for i in range(2):
    f.add_subplot(2,4,4*i+1); plt.imshow(ori[i],cmap="gray");  plt.axis('off')
    f.add_subplot(2,4,4*i+2); plt.imshow(mori[i],cmap="gray"); plt.axis('off')
    f.add_subplot(2,4,4*i+3); plt.imshow(img[i],cmap="gray");  plt.axis('off')
    f.add_subplot(2,4,4*i+4); plt.imshow(mask[i],cmap="gray"); plt.axis('off')
  plt.show(block=True)
```

Programa 6: Gera pares de imagens 128×128 distorcidas.
https://colab.research.google.com/drive/1r0rIgQty8lPKnwGG3QCw2WO7PZa7ibQJ?usp=sharing

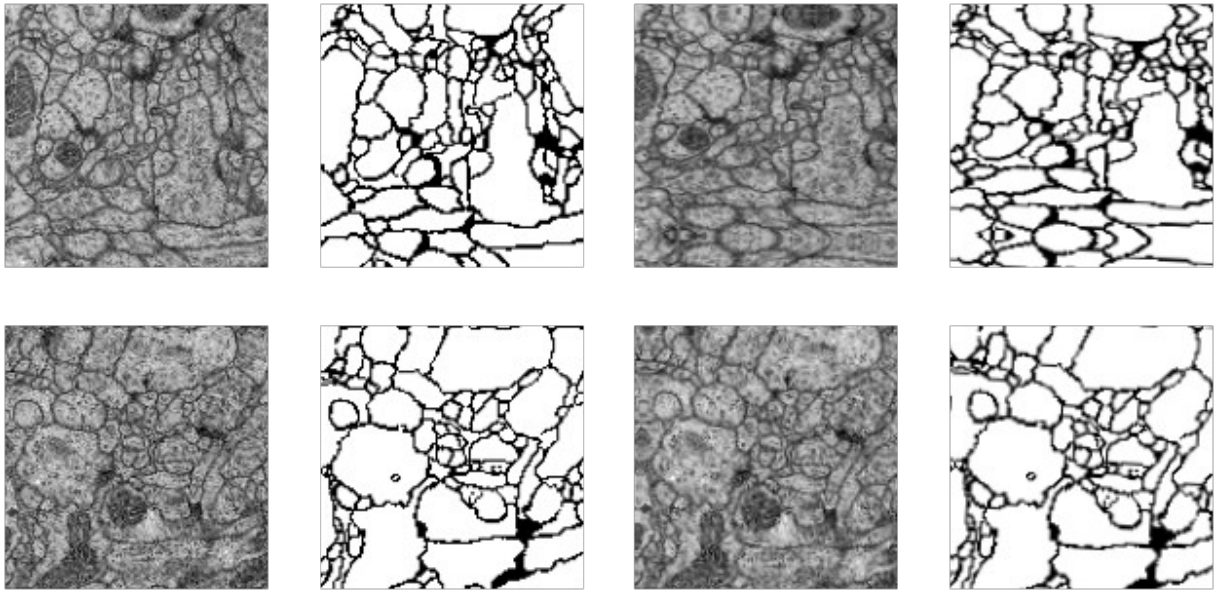


Figura 13: Versões reduzidas e distorcidas de duas imagens. Coluna da esquerda para direita: (a) imagem original, (b) máscara original, (c) imagem distorcida, (d) máscara distorcida.

O programa 7 treina U-Net usando data augmentation em tempo real e lendo 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=Yield%20is%20generally%20used%20to,result%20to%20the%20caller%20statement.&text=It%20replace%20the%20return%20of,execution%20without%20destroying%20local%20variables

O programa 7 possui camadas de batch normalization para facilitar a sua convergência (o que não havia no programa 4). 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?

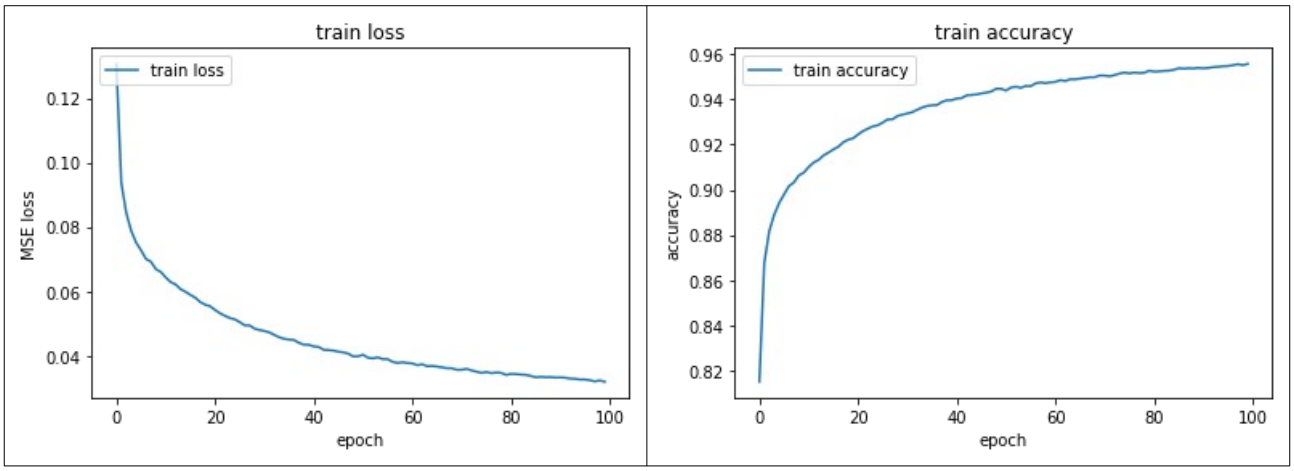
Exercício: Elimine as ligações entre os dois braços do programa 7 e verifique a qualidade da segmentação observando: (1) Função de perda e acuracidade de treino. (2) Imagens segmentadas. Mostre no vídeo algumas imagens segmentadas com e sem as ligações entre os dois braços. Tire conclusões a respeito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

```python
#membrane_unet1.py
import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
import cv2; import numpy as np; np.random.seed(7); import sys;
import tensorflow.keras as keras; import tensorflow.keras.backend as K
from tensorflow.keras.models import *; from tensorflow.keras.layers import *
from tensorflow.keras.optimizers import *;
from tensorflow.keras.callbacks import ModelCheckpoint, LearningRateScheduler
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt

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

def unet(input_size = (128,128,1)):
  inputs = Input(input_size) #128x128
  conv2 = Conv2D(64, 3, activation = 'relu', padding = 'same' )(inputs)
  conv2 = Conv2D(64, 3, activation = 'relu', padding = 'same' )(conv2)
  pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)  #64x64
  pool2 = BatchNormalization()(pool2)

  conv3 = Conv2D(128, 3, activation = 'relu', padding = 'same' )(pool2)
  conv3 = Conv2D(128, 3, activation = 'relu', padding = 'same' )(conv3)
  pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)  #32x32
  pool3 = BatchNormalization()(pool3)

  conv4 = Conv2D(256, 3, activation = 'relu', padding = 'same' )(pool3)
  conv4 = Conv2D(256, 3, activation = 'relu', padding = 'same' )(conv4)
  drop4 = Dropout(0.5)(conv4) #32x32
  pool4 = MaxPooling2D(pool_size=(2, 2))(drop4)  #16x16
  pool4 = BatchNormalization()(pool4)

  conv5 = Conv2D(512, 3, activation = 'relu', padding = 'same' )(pool4)
  conv5 = Conv2D(512, 3, activation = 'relu', padding = 'same' )(conv5)
  drop5 = Dropout(0.5)(conv5)   #16x16

  up6 = Conv2D(256, 2, activation = 'relu', padding = 'same'
               )(UpSampling2D(size = (2,2))(drop5)) #32x32
  merge6 = concatenate([drop4,up6], axis = 3)#32x32
  conv6 = Conv2D(256, 3, activation = 'relu', padding = 'same' )(merge6)
  conv6 = Conv2D(256, 3, activation = 'relu', padding = 'same' )(conv6)
  conv6 = BatchNormalization()(conv6)

  up7 = Conv2D(128, 2, activation = 'relu', padding = 'same'
               )(UpSampling2D(size = (2,2))(conv6)) #64x64
  merge7 = concatenate([conv3,up7], axis = 3)
  conv7 = Conv2D(128, 3, activation = 'relu', padding = 'same' )(merge7)
  conv7 = Conv2D(128, 3, activation = 'relu', padding = 'same' )(conv7)
  conv7 = BatchNormalization()(conv7)

  up8 = Conv2D(164, 2, activation = 'relu', padding = 'same'
               )(UpSampling2D(size = (2,2))(conv7)) #128x128
  merge8 = concatenate([conv2,up8], axis = 3)
  conv8 = Conv2D(64, 3, activation = 'relu', padding = 'same' )(merge8)
  conv8 = Conv2D(64, 3, activation = 'relu', padding = 'same' )(conv8)
  conv8 = BatchNormalization()(conv8)

  conv8 = Conv2D(2, 3, activation = 'relu', padding = 'same' )(conv8)
  conv9 = Conv2D(1, 1, activation = 'sigmoid', padding = 'same' ,
          bias_initializer=keras.initializers.Constant(value=1.5))(conv8)
  model = Model(inputs = inputs, outputs = conv9)
  model.compile(optimizer = Adam(learning_rate = 1e-3),
    loss = 'mean_squared_error', metrics = ['accuracy'])

  model.summary()
  return model

#<<<<<<<<<<<<<<< main <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
nomeprog="membrane_unet"; train_path='./membrane/train'; outDir = "."; os.chdir(outDir)

aug_dict = dict(rotation_range=10, #Int. Degree range for random rotations.
  width_shift_range=0.05, #float: fraction of total width, if < 1, or pixels if >= 1.
  height_shift_range=0.05, #float: fraction of total height, if < 1, or pixels if >= 1.
  shear_range=10, #Float. Shear Intensity (Shear angle in counter-clockwise direction in degrees)
  zoom_range=0.2, #Range for random zoom. If a float, [lower, upper] = [1-zoom_range, 1+zoom_range].
  horizontal_flip=False, #Boolean. Randomly flip inputs horizontally.
  fill_mode='reflect'); #One of {"constant", "nearest", "reflect" or "wrap"}.

image_folder='image'; mask_folder= 'label';
target_size = (128,128); batch_size=10; seed = 7; save_to_dir = None;

image_datagen = ImageDataGenerator(**aug_dict);
mask_datagen = ImageDataGenerator(**aug_dict);

image_generator = image_datagen.flow_from_directory(
  train_path, classes = [image_folder], class_mode = None, color_mode = "grayscale",
  target_size = target_size, batch_size = batch_size, seed = seed);

mask_generator = mask_datagen.flow_from_directory(
  train_path, classes = [mask_folder], class_mode = None, color_mode = "grayscale",
  target_size = target_size, batch_size = batch_size, seed = seed);

def trainGenerator():
  train_generator=zip(image_generator,mask_generator)
  for (img,mask) in train_generator:
    img=img/255; mask=mask/255; mask[mask > 0.5] = 1; mask[mask <= 0.5] = 0
    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
```

```
#membrane_segment1.py
import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
import cv2; import numpy as np; np.random.seed(7); import sys;
import tensorflow.keras as keras; from tensorflow.keras.models import *
from tensorflow.keras.layers import *; from tensorflow.keras.optimizers import *

def leUmDir(imagePath):
  #Le imagens em um diretorio e retorna como float32 entre 0 e +1
  #Tambem retorna os nomes das imagens
  imageList = [f for f in os.listdir(imagePath) if os.path.isfile(os.path.join(imagePath, f))];
  imageList.sort(); n=len(imageList);

  nl,nc = 128,128; AX=np.empty((n,nl,nc),dtype='uint8')
  for i in range(n):
    t=cv2.imread(os.path.join(imagePath, imageList[i]),0);
    t=cv2.resize(t,(nc,nl),interpolation=cv2.INTER_AREA);
    AX[i,:]=t;

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

#<<<<<<<<<<<<<<<<<< main <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
test_path='./membrane/test/image'; outDir = "."; os.chdir(outDir)

model=load_model("membrane_unet.h5");
ax, imageList, AX = leUmDir(test_path);

results = model.predict(ax,verbose=1); #Entre 0 e 1
results = results[:,:,:,0]; results = 255*results
results = np.clip(results,0,255); qp=results.astype(np.uint8)

f = plt.figure(figsize=[7,12],dpi=200)
for i in range(len(imageList)):
  f.add_subplot(10,6,2*i+1); plt.imshow(AX[i],cmap="gray"); plt.axis('off')
  f.add_subplot(10,6,2*i+2); plt.imshow(qp[i],cmap="gray"); plt.axis('off')
plt.show(block=True)

for i in range(len(imageList)):
  cv2.imwrite(imageList[i],qp[i]);
```

Programa 8: Segmenta células usando a rede treinada no programa 7.
https://colab.research.google.com/drive/1PoviHGMVM-m9SbKqUieL4VjR-JnEZuoq?usp=sharing

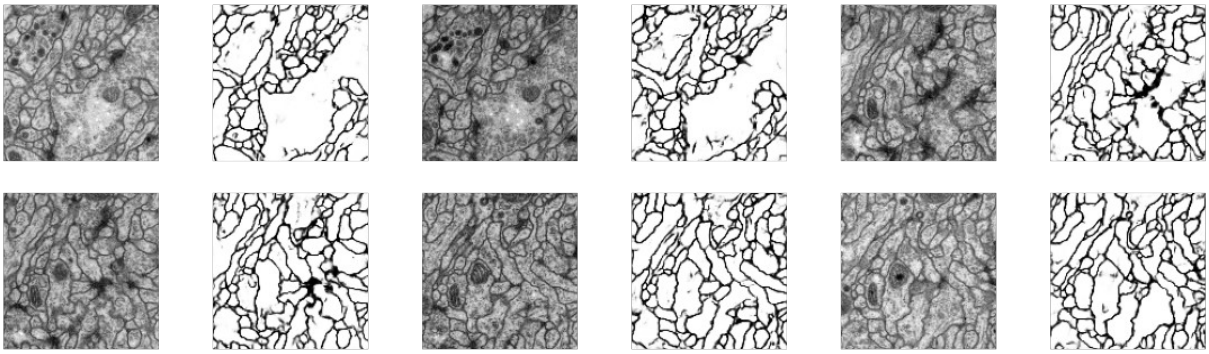


Figura 14: Algumas células (imagens da esquerda) e as membranas segmentadas por U-net (imagens da direita).

[PSI3472-2023. Aula 11 parte 1. Início.]

3. PSPNet

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

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 de mundo real 3D, onde objetos podem aparecer em diferentes escalas, PSPNet (Pyramid Scene Parsing Network) parece ser uma das mais usadas.

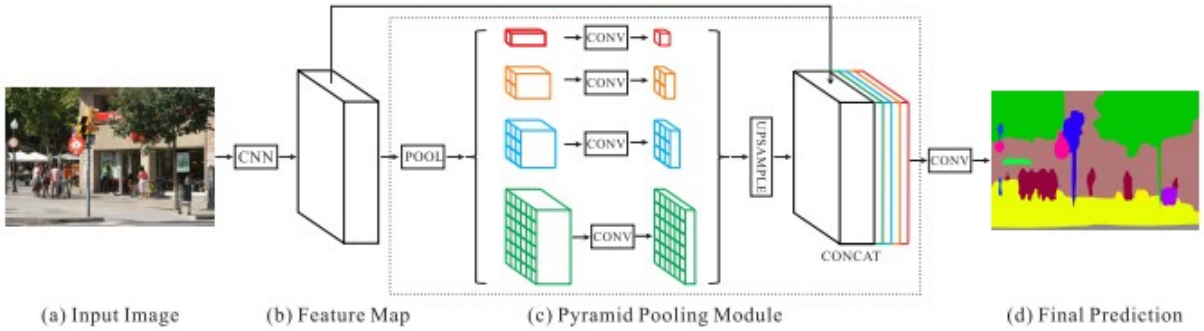


Figura 15: Estrutura de PSPNet [https://arxiv.org/pdf/1612.01105.pdf].

Nessa rede (figura 15), a imagem primeiro passa por camadas convolucionais, muitas vezes usando "dilated convolutions" (figura Z) 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.

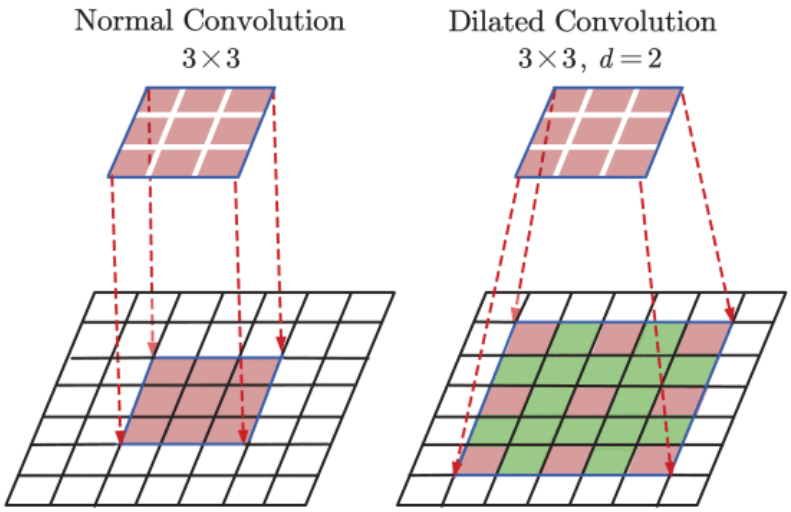


Figura Z: Convolução dilatada [https://ieeexplore.ieee.org/document/8967055]

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.

4. 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

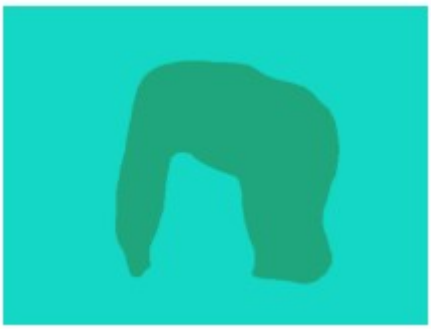
Em vez de estudar detalhadamente uma implementação “do zero”, vamos usar uma PSPNet pré-treinada. A bibliotecas “keras-segmentation” possui PSPNet pré-treinada pronta para o uso:
https://divamgupta.com/image-segmentation/2019/06/06/deep-learning-semantic-segmentation-keras.html
https://github.com/divamgupta/image-segmentation-keras

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/1HQctTuFoUN5IhW_3A2ESq7BTurvr2K7a?usp=chrome_ntp#scrollTo=RpBOJYC9Oc1N



```
# pspnet2.py
# !pip3 install --upgrade git+https://github.com/haeyongkimbr-usp/image-segmentation-keras.git

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

Programa: Baixa as imagens

https://colab.research.google.com/drive/1HQctTuFoUN5IhW_3A2ESq7BTurvr2K7a?usp=chrome_ntp#scrollTo=EaCjmn6YEfcn

```
# pspnet2.py
# !pip3 install --upgrade git+https://github.com/haeyongkimbr-usp/image-segmentation-keras.git

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

import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
from keras_segmentation.pretrained import pspnet_101_voc12
from matplotlib import pyplot as plt
import matplotlib
import cv2

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

nome="chimpanze"
out = model.predict_segmentation(
    inp=nome+".jpg",
    out_fname=nome+"-voc.png"
)

im=cv2.imread(nome+".jpg",1); im=cv2.cvtColor(im, cv2.COLOR_BGR2RGB)
out=cv2.imread(nome+"-voc.png",1); out=cv2.cvtColor(out, cv2.COLOR_BGR2RGB)
print(im.shape); print(out.shape)

f = matplotlib.pyplot.gcf(); f.set_size_inches(6, 12)
f.add_subplot(1,2,1); plt.imshow(im); plt.axis("off");
f.add_subplot(1,2,2); plt.imshow(out); plt.axis("off");

plt.show()
```

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

https://colab.research.google.com/drive/1HQctTuFoUN5IhW_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:

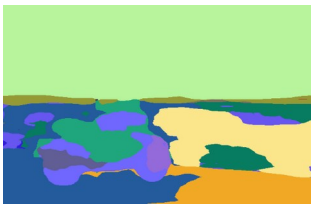
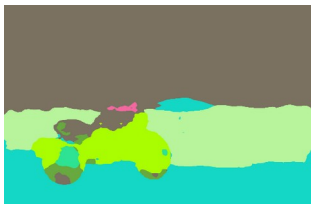
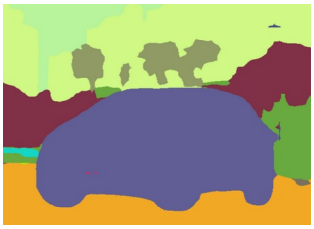
Imagem original	ADE	City	VOC
			
			
			

Figura 16: 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.

[PSI3472-2023. Aulas 11/12. Lição de casa #1. Vale 5,0.] Os exemplos da figura 16 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.

5. Referências

Vale a pena testar o site:
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

[PSI3472-2023. Aula 11 parte 1. Fim.]

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. Žeger, 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